Functional Ownership through Fractional Uniqueness

DANIEL MARSHALL, University of Kent, United Kingdom
DOMINIC ORCHARD, University of Kent, United Kingdom and University of Cambridge, United Kingdom

The Rust programming language, which is experiencing a surge in popularity in recent years, is known for its intricate system of ownership and borrowing, designed to enforce safe memory management. Rust also aims to bring some of the guarantees offered by functional programming languages into the realm of performant system code. As things stand, this is largely separate from the ownership model, with type and borrow checking happening in separate compilation phases. Recent models such as RustBelt and Oxide aim to formalise Rust's memory management in a type-theoretic way; this opens the door for functional languages to borrow some of the ideas originated by Rust, so that more programmers can experience their benefits.

One strategy for dealing with memory usage in a functional setting is through the notion of *uniqueness types*, but these offer a fairly coarse-grained view: either a value has exactly one reference, and can be mutated safely, or it cannot, since other references may exist. Recent work has demonstrated that *linear* types and *uniqueness* types can be used within a single system to offer both restrictions on program behaviour and guarantees about memory usage. This paper extends this connection further; we show that just as *graded* type systems like those of Granule and Idris build upon the idea of linearity, Rust's *ownership* model builds on the idea of uniqueness. We develop a combined calculus based on ideas from fractional permissions and graded types, and implement this in the Granule language. This represents the first account of Rust-style ownership that can smoothly integrate into a standard functional type system.

ACM Reference Format:
Daniel Marshall and Dominic Orchard. 2023. Functional Ownership through Fractional Uniqueness. 1, 1 (July 2023), 83 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Rust programming language has dramatically grown in popularity in recent years, having been adopted as the second official language of the Linux kernel,[1] and also deployed in production code by companies including AWS, Huawei, Google, Microsoft and Mozilla, all of whom are founding members of the Rust Foundation.[2] This is in large part due to its focus on memory safety; Rust finds a happy medium between systems programming languages like C which offer precise control but little in the way of safety guarantees and the contrasting approach of higher-level languages like Java or Go where memory is managed automatically through garbage collection.

The intricate ownership system which characterises Rust's approach to memory management is inspired by the literature on using *linear types* for tracking resource usage. These were originally based on Girard's linear logic [Girard 1987] but have been expanded upon from many angles due to their applications in the setting of type theory [Wadler 1990]. Linear types have also been brought into the realm of practical programming in recent years, particularly by the advent of a linear types extension to Haskell [Bernardy et al. 2017]. The precise theoretical relationship between purely linear types and the properties that can be enforced by Rust's borrow checker remains unclear,

[1] https://www.zdnet.com/article/rust-takes-a-major-step-forward-as-linuxs-second-official-language/
[2] https://foundation.rust-lang.org/news/2021-02-08-hello-world/

Authors' addresses: Daniel Marshall, School of Computing, University of Kent, United Kingdom, dm635@kent.ac.uk; Dominic Orchard, School of Computing, University of Kent, United Kingdom , Department of Computer Science and Technology and University of Cambridge, United Kingdom, D.A.Orchard@kent.ac.uk.

however, particularly since Rust lies outside the traditional functional paradigm to which linearity is most suited and opts for a more mixed imperative and functional approach.

Modern resourceful type systems often allow for going beyond the coarse restrictions of linearity, enabling usage to be classified more finely. Quantitative types, inspired by bounded linear logic [Girard et al. 1992], capture an upper bound on the precise usage of a value by indexing types via natural numbers or polynomial terms. This can be generalised further to *graded* types as exemplified by the Granule programming language [Orchard et al. 2019], which allows various properties of data use and data flow to be tracked smoothly by a single type system. It is natural to wonder whether the complex ownership properties specifiable in Rust can be represented in some way directly inside a resource-based type system through an application of the more precise grading expressible in Granule, and that is our aim in this work.

Many existing models of ownership from a type-theoretic perspective already exist. For example, λ_{Rust} developed as part of the RustBelt framework provides a low-level model of Rust's ownership and borrowing systems suitable for formally verifying properties about Rust programs [Jung et al. 2017], while Oxide [Weiss et al. 2019] and FR [Pearce 2021] represent more high-level theoretical models. However, we offer a different but complementary perspective, providing a system that both explains how to relate ownership and borrowing ideas with existing linear type systems and also allows for their integration in a single setting, offering possibilities for representation of complex type and memory safety properties all at the type level.

A natural place to start is with *uniqueness types*, which already represent the idea that a uniquely typed value has exactly one reference, and therefore it is safe to mutate [Barendsen and Smetsers 1996; de Vries et al. 2008; Smetsers et al. 1994]. This represents a simplistic model of ownership without borrowing—a resource with a unique owner may be modified by that owner, but a broken guarantee of uniqueness can never be recovered. Indeed, recently a type system has been developed that captures the relationship between linear and unique types: linear values are restricted from being copied or discarded in the future, whilst unique ones are guaranteed to have never been copied in the past [Marshall et al. 2022]. We build upon this work, which combines linearity and uniqueness within Granule, as our foundation here.

Just as the non-linearity modality ! can be generalised to a graded modality $!_r$ for fine-grained reasoning about resource usage via the index r and its algebraic structure [Gaboardi et al. 2016; Orchard et al. 2019], we show that uniqueness types, represented modally, can be 'graded' to capture the idea that borrowing is a controlled relaxation of the uniqueness guarantee. This allows many ideas from Rust, such as immutable and mutable borrows, partial borrows and reborrowing to be represented in a functional setting, all through the application of an elegant form of grading based on Boyland's fractional permissions [Boyland 2003]. Our work thus integrates and relates within a unified framework the substructural type systems of linearity, uniqueness, grading, ownership and borrowing. This unified framework provides a path to a future where ownership properties can be expressed more simply in languages like Haskell and dually linear or graded types can be used to their best effect in languages like Rust.

The outline of the paper is as follows. First, in Section 2 we go over the key concepts of uniqueness and borrowing in the context of Rust, and discuss how various programs making use of these ideas will later be rewritten as Granule programs through our unified type system, in order to motivate the tools we will develop in further sections. We then recap Granule's pre-existing core calculus in Section 3, before moving on to the primary contributions of this paper:

- In Section 4, we connect Granule's unique and graded modal types with ideas from Rust, discussing how unique ownership allows for safe mutation. We generalise the connection between uniqueness and linearity, demonstrating that uniqueness and precise grading can

coexist within a single type system. We also introduce a notion of 'identifiers' for situations where multiple references point to the same value, making use of existential types.
- In Section 5, we extend this idea to allow for multiple immutably borrowed references at a time, by carefully tracking references that exist, similarly to Boyland's fractional permissions. We show that this allows for a more fine-grained approach to tracking uniqueness in much the same way that grading increases expressivity over pure linearity.
  - In Section 5.1, we develop an equational theory for our extension to Granule's type system, showing that the modality for unique ownership induces a *relative functor* over the new modality for mutable and immutable borrowing.
  - In Section 5.2, we discuss how using distributive laws can allow for borrowing only part of a larger data structure while leaving the rest uniquely owned, enabling a much wider variety of practical programming patterns.
- In Section 6, we detail the semantics for the type system developed thus far, and prove various key properties–both standard notions like progress and preservation but also borrow safety and uniqueness. This goes beyond earlier work by incorporating borrowing but also by adapting the previous call-by-name semantics to a call-by-value setting.

Finally, in Sections 7 and 8 we discuss the many and varied areas of similar research that have inspired this paper, and look at some possible avenues for future work. All source code discussed and presented in this work will be made available alongside the artifact for this paper, along with the implementation of the type system that we will describe built on top of the Granule compiler.[3]

2 KEY RUST CONCEPTS

In order to develop a type system which integrates ownership and borrowing ideas from Rust with the linear and graded types already present in Granule, we will first need to understand the ideas in question. This section presents six key patterns which we aim to capture in the theory we discuss throughout this paper, along with simple Rust code examples to demonstrate the patterns in action. Each of these examples relies on a single base value of type `Colour(u32, u32, u32)` - a struct containing three unsigned integers, representing a colour with red, green and blue components.

2.1 Ownership

The first crucial concept is the notion of an *owned* value, where a variable is 'owned' by a particular identifier; here, the value `Colour(220, 20, 60)` is owned by the identifier `scarlet`. Each variable can only be owned by a single identifier at any given time. To enforce this, Rust uses *move semantics*, which are illustrated in the following example: on the second line, ownership of the value is *moved* to the identifier `x`. Now, the identifier `scarlet` no longer owns the value, so attempting to use it again on the third line gives an error.

```
1  let scarlet = Colour(220, 20, 60);
2  let x = scarlet;
3  let y = scarlet; // error: use of moved value: `scarlet`
```

Rust ✗

It is fairly clear that the idea here in some way relates to linearity, since linear values can only be used once which restricts them to being passed around sequentially; indeed, much of the literature on Rust makes mention of linear (or affine) types. We will demonstrate through our unified type system in Chapter 4 that in fact ownership can be better understood as an extension of *uniqueness*

[3]If the reader wishes to experiment with the Granule language, the latest releases are available from https://github.com/granule-project/granule/releases.

types, since each value having a single owner implies that the owner holds the unique reference to said value. these are similar to linear types in some ways but with important differences.

2.2 Immutable borrowing

Borrowing allows for generalising the concept of ownership, permitting multiple references to point to a single value simultaneously. Rust's *borrow checker* carefully manages all borrows that exist at a given time, ensuring that eventually values are returned to their owners and that in the meantime various memory safety properties are maintained. *Immutable* borrows are one flavour of reference; any number of these can exist at a given time, but this means that mutation of a value cannot be allowed through an immutable borrow, since this could result in data races. In the below example, two immutable borrows (`x` and `y`) both reference the original `persimmon` value.

```
let persimmon = Colour(252, 118, 5);
let x = &persimmon;
let y = &persimmon; // ok!
```

Rust ✓

2.3 Mutable borrowing

It is also possible to borrow values and retain the capacity to mutate them, but this comes at a price: as mentioned above, allowing mutation through multiple references simultaneously is harmful to memory safety, so in order to prevent this only a single mutable borrow is permitted at a time. Much like with owned values, mutably borrowing a value guarantees the sole capability for destructive access to the underlying data. This is demonstrated below, where attempting to create two *mutable* borrows following the pattern of the above example is disallowed.

```
let mut viridian = Colour(52, 161, 128);
let x = &mut viridian; // error: cannot borrow `viridian` as mutable
let y = &mut viridian; //                   more than once at a time
```

Rust ✗

Note, however, that the above example copied verbatim into a Rust file will in fact compile successfully unless additional code is introduced that makes use of the variables `x` and `y`; this is due to a feature called *non-lexical lifetimes*, through which the Rust compiler is able to *infer* that allowing two mutable borrows is safe as long as one of them is never used. We do not linger on this, since we will not be attempting to capture this behaviour in our type system; our goal is to embed some essential ownership and borrowing patterns *explicitly* at the type level. However, we will mention the possibility for extending our system with more advanced ideas in Section 8.

2.4 Mixing mutable and immutable borrows

Rust also disallows creating immutable borrows to values that are already borrowed mutably, since this invites similar problems; the value could update through the mutable borrow while it is being read through the immutable borrow, for instance. Hence, the following example wherein the second reference is borrowed immutably rather than mutably is also forbidden by the borrow checker. In Section 5, we will demonstrate that both of these notions can be represented through a generalisation of uniqueness typing, via annotations that represent the borrower's level of access.

```
let mut cerulean = Colour(3, 109, 230);
let x = &mut cerulean; // error: cannot borrow `cerulean` as immutable
let y = &cerulean;     //       because it is also borrowed as mutable
```

Rust ✗

2.5 Partial borrowing

One useful pattern for working with larger data structures is the ability to borrow only *part* of a structure, while the original owner retains access to what remains. This is valuable for allowing one part of your program to work with a particular piece of data without restricting access to the entire larger structure, enabling tasks to be carried out in parallel much more easily. In the example that follows, the red and green components of the `indigo` value are simultaneously borrowed mutably, but the Rust compiler allows this since the references point to disjoint parts of the original struct.

```
let mut indigo = Colour(32, 36, 209);
let r = &mut indigo.0;
let g = &mut indigo.1; // ok!
```

Rust ✓

Later, in Section 5.2, we will take advantage of Granule's functional nature to present a cohesive way of managing this kind of partial borrow at the type level, including the possibility for working with disjoint components of the original structure concurrently.

2.6 Reborrowing

The final pattern we aim to capture in our type system, also essential for practical Rust programming, is the notion of *reborrowing*, through which it is possible to create a borrow of an identifier that itself references another value. We will show in Section 5 that this behaviour falls out of our generalisation of uniqueness naturally, without the need for additional constructs. To illustrate the general idea, consider the following example; here, the red component of the `amethyst` value is borrowed mutably as `r`, then another mutable borrow `x` is created pointing to `r`. The value can now only be mutated through `x`–both `r` and `amethyst` are inaccessible until the borrow is complete.

```
let mut amethyst = Colour(98, 1, 170);
let r = &mut amethyst.0;
let x = &mut *r; // ok!
```

Rust ✓

3 CORE CALCULUS

So that we can later establish key properties for our overall system, we recap the well-trodden details of Granule's core type system, originally presented by Orchard et al. [2019], which incorporates graded modal types into a linear type theory. Later, we will build on this core calculus to develop our functional model of uniqueness and borrowing. In this section, we focus on the syntax and static semantics (type system) for the core of Granule; Section 6 will later give an operational semantics incorporating the various extensions discussed throughout this work.

The core type theory we describe extends the linear λ-calculus with products, a multiplicative unit, and a *semiring-graded necessity modality* $\Box_r A$, where for a pre-ordered semiring[4] $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ there exists a family of types $\{\Box_r A\}_{r \in \mathcal{R}}$. This language represents a simplified monomorphic subset of Granule [Orchard et al. 2019], but also closely resembles other graded systems from the literature [Abel and Bernardy 2020; Atkey 2018; Brunel et al. 2014; Choudhury et al. 2021; Gaboardi et al. 2016; McBride 2016; Moon et al. 2021; Petricek et al. 2014]; extensions for uniqueness and borrowing discussed here could be made compatible with such systems also, since they do not rely heavily on particular details of the core calculus in question.

[4]A pre-ordered semiring extends a semiring with a pre-order where $*$ and $+$ must then also be monotonic wrt. the ordering.

An additional extension we include is existential types and type variables (restricted to a particular kind), to be used for tracking type-level identifiers associated with resources (whose dataflow paths may fork and join due to the borrowing patterns we discuss later).

3.1 Syntax

Our syntax is that of the linear λ-calculus with multiplicative products and unit (first line of syntax below), with additional terms for introducing and eliminating the $\Box$ modality (second line) and existentially quantified identifiers (third line):

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t_1\ t_2 \mid (t_1, t_2) \mid \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 \mid () \mid \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 \\ & \mid [t] \mid \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 \\ & \mid \mathbf{pack}\ \langle id, t\rangle \mid \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 \end{aligned} \qquad \text{(terms)}$$

Following the syntax of variables, we group terms above into pairs of introduction and elimination forms, for functions, tensors, units, the graded modality, and existential types respectively. The meaning of these terms is explained in the next subsection with reference to their typing.

3.2 Type System

Typing judgments have the form $\Gamma \vdash t : A$, assigning type A to term t under context Γ. Types are:

$$A, B ::= A \multimap B \mid A \otimes B \mid \mathsf{unit} \mid \Box_r A \mid \exists id.A \qquad \text{(types)}$$

Hence, our type syntax comprises linear function types $A \multimap B$, linear multiplicative products $A \otimes B$, a linear multiplicative unit (unit), the graded modality $\Box_r A$ where $r \in \mathcal{R}$, and existentially quantified types where id : Name for an abstract kind of names Name.

Contexts Γ contain both linear assumptions $x : A$, graded assumptions $x : [A]_r$ which have originated from inside a graded modality, and type variables which we write as id due to their restricted purpose here, omitting the kind which is the abstract type Name:

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \mid \Gamma, id \qquad \text{(contexts)}$$

Syntax and typing are then given by the following rules:

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

The VAR, ABS, and APP rules are the standard rules of the linear λ-calculus, augmented with a notion of *contraction* captured by the + operation on contexts coming from multiple sub-terms, which is only defined when contexts are disjoint with respect to linear assumptions, and on overlapping graded assumptions we add their grades, e.g. $(\Gamma_1, x : [A]_r) + (\Gamma_2, x : [A]_s) = (\Gamma_1 + \Gamma_2), x : [A]_{r+s}$. More explicitly, context addition is specified as follows:

$$\begin{aligned} (\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| & \emptyset + \Gamma &= \Gamma \\ \Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma| & \Gamma + \emptyset &= \Gamma \\ (\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) &= (\Gamma + \Gamma'), x : [A]_{(r+s)} \\ (\Gamma, id) + (\Gamma', id) &= (\Gamma + \Gamma'), id \end{aligned}$$

(in the first two cases, x may be id with A = Name implicitly). This is a declarative specification of + rather than an algorithmic definition, as graded assumptions of the same variable may appear in different positions within the two contexts. The var rule also embeds the notion of *weakening*, allowing a context of variables graded by 0. $r \cdot \Gamma$ is partial, scaling each graded assumption in Γ by r (0 in this case), preserving id type variables, but undefined if Γ contains any linear assumptions.

The rules involving graded modalities are perhaps more interesting:

$$\frac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim} \qquad \frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

The PR rule (promotion) introduces a graded modality with grade r, implying that the result of t can be used in an 'r-like' way and thus all of the dependencies of t must be scaled by r to propagate usage to the dependencies, none of which are allowed to be linear. Promotion also contains an explicit restriction that the value to be promoted must not be a *resource allocator*, designed to avoid problems that otherwise arise when promoting values with certain behaviours in a call-by-value setting; we will discuss this in detail when we describe our interface for mutable arrays in Section 5.

The ELIM rule eliminates a graded modality, capturing the idea that a requirement for x to be used in an 'r-like' way in t_2 can be matched with the capability of t_1 described by its graded modal type. In Granule, this construct is folded into pattern matching: we can 'unbox' (eliminate) a graded modality to provide a graded variable in the body of the function (the analogue to t_2 in this rule).

The DER rule connects linear typing to graded typing, showing that a requirement for a linear assumption is satisfied by an assumption graded by 1. The last important rule is the APPROX rule which provides grade approximation; this allows a grade r to be converted to another grade s, provided that s *approximates* r where $\sqsubseteq$ is the pre-order of the semiring in question.

For example, one possible choice of semiring is the semiring of natural numbers, which we define as $(\mathbb{N}, +, 0, 1, \equiv)$. The *discrete ordering* $\equiv$ here means that this semiring has no approximation, and so we are tracking the exact number of times a term has been used. We could instead use the standard $\leq$ ordering on natural numbers which would permit approximation, allowing for an *upper bound* on a term's usage. Another useful semiring involves *intervals* of natural numbers. Here, we can use a grade such as $0..1$ to represent a value which can be used *either* zero or one times; this captures the notion of an *affine* value, often discussed in existing literature on Rust. Other interesting semirings include lattices for security levels [Abel and Bernardy 2020; Gaboardi et al. 2016], hardware schedules [Ghica and Smith 2014], and sets for abstract property tracking.

Finally, we have the rules for introducing and eliminating tensor products and the multiplicative unit, which are standard, though it is important to remember that products are *linear* and so it is not possible to freely discard either side: both elements must be used when consuming a product.

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

Example 3.1. The following gives an example derivation assuming the natural number semiring:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \otimes A \vdash x : A \otimes A}\ \textsc{var} \quad \dfrac{}{y : [A]_1 \vdash y : A}\ \textsc{var'}}{y : [A]_1, x : A \otimes A \vdash (x, y) : (A \otimes A) \otimes A}\ \otimes_I}{y : [A]_1 \vdash \lambda x.(x, y) : A \otimes A \multimap (A \otimes A) \otimes A}\ \textsc{abs} \quad \dfrac{\dfrac{}{\emptyset \vdash v : A} \quad \dfrac{}{y : [A]_1 \vdash y : A}\ \textsc{var'}}{y : [A]_1 \vdash (v, y) : A \otimes A}\ \otimes_I}{y : [A]_2 \vdash (\lambda x.(x, y))\,(v, y) : (A \otimes A) \otimes A}\ \textsc{app}$$

where (VAR') is a synonym for the (VAR) rule followed by dereliction (DER).

As a first taste of Granule syntax (which resembles Haskell, apart from the presence of the graded modal type constructs), the following captures the same idea as this example:

```
example : ∀ {a : Type} . a → a [2] → ((a, a), a)
example v yb = let [y] = yb in (λx → (x, y)) (v, y)
```
Granule

The type $\Box_r A$ is instead written postfix as `A [r]`, and Granule uses $\rightarrow$ for its linear function types.

Existential types have standard introduction and elimination typing forms, but restricted only to type variables of kind Name (whose kind is omitted here for simplicity due to this restriction):

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \textsc{pack} \qquad \frac{\Gamma_1 \vdash t_1 : \exists id.A \qquad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unpack}$$

Note on Linear Haskell. As mentioned at the start of this section, we aim for the extensions we develop throughout this work to be compatible with other graded type systems after some adaptation. One particular system of interest is the calculus underlying the recent extension which introduces linear types to Haskell [Bernardy et al. 2017]. This extension uses a graded type system below the surface in order to implement linearity; all function types are given a 'multiplicity' annotation r written `(a %r -> b)` akin to a type $\Box_r A \multimap B$ here, where the annotation can currently be either `'One` or `'Many` representing either linear or unrestricted usage respectively. Work on formalising the precise connection between systems such as this where all function types come with a grade (sometimes called 'graded base') and systems such as Granule's where values are linear by default and graded values are wrapped inside a modality (called 'linear base') is ongoing.

4 ONE OF A KIND: UNIQUENESS AND SHARING

The first extension we make to Granule's core type system is to represent *ownership*: values that have a unique owner who can mutate the value freely because they are in possession of the only reference that exists. It turns out this matches closely with the idea of *uniqueness* types; if it is possible to guarantee that a reference to a value is unique (i.e., it is the only reference that exists), then whichever part of the program (e.g., thread/process) holds that reference must be the owner.

Uniqueness types were introduced into Granule's core type system in previous work [Marshall et al. 2022], but we will extend this calculus in two ways - first, we generalise the rules of Marshall et al. [2022] to allow for a graded comonadic modality parameterised by an arbitrary semiring rather than the simple non-linear !, and second, we incorporate the identifiers described in Section 3 into the typing, which will become important when multiple references need to be tracked.

The crucial insight for integrating owned values with Granule's existing linear and graded values, as with more traditional uniqueness typing, is to note that we can consider any linear or graded value to have no ownership information attached; these values have their memory managed by a garbage collector in the runtime. We reintroduce Marshall et al.'s uniqueness modality, wrriten $*$, to represent values that are *owned*, meaning that we must take care to ensure that only one reference exists at any given time. We extend the syntax of terms and types as follows:

$$t ::= \ldots \mid \mathbf{share}\ t \mid \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \qquad A, B ::= \ldots \mid *A \qquad \text{(terms \& types, extended)}$$

The two constructs for the uniqueness modality $*$ provide sharing and cloning, with types:

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\ t : \Box_r A}\ \textsc{share} \qquad \frac{1 \sqsubseteq r \qquad \mathsf{noIDs}(\Gamma_1) \qquad \Gamma_1, \overline{id} \vdash t_1 : \Box_r A \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t_2 : \Box_r B}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone}$$

The SHARE construct allows a guarantee of unique ownership to be discarded, though this means that memory must now once again be managed automatically just as with any other Granule values where ownership is not tracked. Any grade r can be selected here, though the shared value must eventually be either discarded (requiring $0 \sqsubseteq r$) or fully consumed via pattern matching.

The CLONE construct makes a *deep copy* of the value t_1, where we take a value that does not necessarily have a unique owner and make a unique copy of it; we take ownership of the copy, binding it the scope of t_2. We can guarantee uniqueness since this is now the only reference that exists to the newly copied value. We must update any identifiers along the way, however, to ensure that this is genuinely understood as a separate value to the original, in case other references to the original still exist. This is mediated by existential types, where all identifiers $\overline{id}$ used in typing t_1 are bound under an existential quantifier (where the predicate noIDs(Γ_1) ensures there are no further identifiers in Γ_1). The side-condition $1 \sqsubseteq r$ explains that we must be able to accommodate a usage of the input t_1 since **clone** consumes the t_1 value once in copying it.

Our implementation of these ideas in Granule follows the above typing. We recap the initial ownership example from Section 2 here in our new extension of Granule using a data type `Colour` which acts as an alias for a triple of type `(Int, (Int, Int))`. The following code illustrates that the base linearity of Granule's core calculus requires us to obey the laws of "move semantics" when working with owned values; we must move ownership of `c` to `x` and then again to `y`.

```
scarlet : *Colour → *Colour
scarlet c = let x = c in
            let y = x in y
```
Granule

One way in which we can create multiple variables which all point to the initial value of type `Colour` is via the **share** operation; thus, we are explicitly exempting the value from being managed by the ownership system and moving it back into the domain of the garbage collector. Here, we use the interval grade `0..2`, capturing the idea that the value must be used somewhere between zero and two times (where here it happens to be used twice).

```
scarlet' : *Colour → Colour [0..2]
scarlet' c = let [s] = share c in
             let [x] = [s] in
             let [y] = [s] in [y]
```
Granule

The **share** and **clone** operations have the following equations showing their interaction:

$$\textbf{clone}\ (\textbf{share}\ v)\ \textbf{as}\ x\ \textbf{in}\ t \equiv [\textbf{pack}\ \langle \overline{id}, v\rangle / x]\, t \qquad \text{(unitL)}$$

$$\textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ (\textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3) \equiv \textbf{clone}\ (\textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2)\ \textbf{as}\ y\ \textbf{in}\ t_3 \quad (x \notin \mathsf{FV}(t_3)) \qquad \text{(assoc)}$$

The (unitL) axiom states that sharing an owned value v and cloning it to create a new owned x in the scope of t is equivalent to substituting the original value v for x in t (with its identifiers packed in an existential). The (assoc) axiom is associativity of cloning.

The presence of identifiers necessitates the existential typing. The need for identifiers is shown in the next section once we move to fractional uniqueness with the ability to separate *immutable borrows* from *mutable borrows*. In a sub-calculus *without* identifiers, the above rule can be simplified:

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \textbf{clone'}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2 : \Box_r B}\ \text{CLONE'}$$

The non-graded version of this rule (with ! instead of $\Box_r A$) was presented in Marshall et al. [2022]; in a setting without borrowing, the $*$ modality then acts as a functor over which each $\Box_r$ becomes a *relative monad*, much as with ! in prior work. A relative monad comprises a functor J and an object mapping T, along with an operation $\eta : JX \to TX$ and a mapping from $JX \to TY$ arrows to $TX \to TY$ with axioms analogous to the monad axioms [Altenkirch et al. 2010]. Thus, here J is the uniqueness modality $*$ and T the graded $\Box_r$ modality. In this setting an additional rule holds, which along with (unitL) and (assoc) completes the full set of relative monad operations:

$$\textbf{clone'}\ t\ \textbf{as}\ x\ \textbf{in}\ (\textbf{share}\ x) \equiv t \qquad \text{(unitR)}$$

$$\textbf{clone'}\ (\textbf{share}\ v)\ \textbf{as}\ x\ \textbf{in}\ t' \equiv [v/x]t' \qquad \text{(unitL)}$$

$$\textbf{clone'}\ t_1\ \textbf{as}\ x\ \textbf{in}\ (\textbf{clone'}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3) \equiv \textbf{clone'}\ (\textbf{clone'}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2)\ \textbf{as}\ y\ \textbf{in}\ t_3 \qquad \text{(assoc)}$$

The (unitR) axiom states that cloning a non-linear t to create a new owned value x and then immediately sharing said owned value is equivalent to just using the original term t.

In our extended calculus, (unitR) is however no longer possible, because we would first need to 'unpack' the existential, and then sharing would lead to a type in which the existentially quantified identifier would 'leak' out of the scope. The typing of **unpack** prevents this; a term of the form **clone'** t **as** x **in unpack** $\langle id, y\rangle = x$ **in share** y is not well-typed. The result is that we now have something close to a relative monad—but not quite. Instead, the loss of (unitR) comes with the gain of tighter control of the *lifetime* of unique values with identifiers.

5 IMMUTABLY BORROWED IS FRACTIONALLY UNIQUE

We now generalise from the system which incorporates uniqueness into the core linear and graded calculus to a system that allows for the uniqueness guarantees to be temporarily broken in controlled ways such that we can continue to ensure memory safety and more closely approximate Rust's more complex ownership and borrowing rules. The first concept we will introduce is that of a *mutable borrow*, which allows for temporary mutable access to a value while preserving the guarantee that it will be eventually returned to its original owner. In our calculus, this is represented through a new $\&_1 A$ modality (with the annotation 1 representing total unhindered access); as in the previous chapter, we need to augment our syntax for terms as follows:

$$t ::= \ldots \mid \textbf{withBorrow}\ t_1\ t_2 \mid \textbf{split}\ t \mid \textbf{join}\ t_1\ t_2 \qquad \text{(terms, extended)}$$

We also introduce a new graded modality into our syntax for types:

$$A, B ::= \ldots \mid \&_p A \qquad \text{(types, extended)}$$

where p represents a new form of grade called a *permission*, for tracking borrowing. Much as grading non-linearity gives a more precise account of resource usage, borrowing is a graded form of non-*uniqueness* that allows us to manage references more carefully. Permissions are either rational numbers between 0 and 1 or a special permission $*$ which is used to allow for polymorphism over owned and borrowed values. Their syntax is defined as follows:

$$p, q ::= f \mid * \qquad \text{(permissions)}$$

where $(f \in \mathbb{Q}, 0 < f \leq 1)$. Note that 0 is excluded: the typing rules we provide can never produce a value with permission 0. The creation of mutable borrows is mediated through the following WITH& rule:

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ t_1\ t_2 : *B}\ \text{WITH\&}$$

Here, we allow for a uniquely owned $*A$ value to be borrowed and manipulated in a mutable way by some function t_2 which expects a $\&_1 A$ as input, so long as it returns the mutable borrow as

a $\&_1 B$ in the output so that the original owner can reclaim this as a unique reference ($*B$). By encapsulating the mutable borrow behaviour inside a continuation, we ensure that it is impossible to construct a closed term with a borrowed type, which means borrowed references to values must always eventually return full access to the owner of said value.

In order to generalise this further to allow for the notion of *immutable borrows*, wherein multiple references to a value can exist at any given time so long as they are not able to mutate it (as this can interfere with memory safety and lead to problems such as data races), we take inspiration from Boyland's fractional permissions [Boyland 2003], which themselves served as partial inspiration for Rust's notion of ownership. This approach also takes note of recent literature on integrating linear types with fractional permissions [Makwana and Krishnaswami 2019].

Mutable borrows are already graded with permission 1, representing full unfettered access to a value, so both reads and writes are permissible. We provide SPLIT and JOIN rules to generalise this:

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \textsc{split} \qquad \frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

Here, the SPLIT rule allows a single borrowed reference to split into two new borrowed references to the same value, each annotated with half of the fraction of the original borrow. Note that SPLIT could be generalised to produce a vector with any number n of $\&_{\frac{p}{n}} A$, or even in theory to arbitrary permissions q and q' such that $q + q' = p$, but here we restrict to halving for simplicity. JOIN allows for two borrows to be recombined into one in much the same way, with their fractions being combined additively; note in particular here that the type contains identifier information from when the resource was created, so it is never possible to combine references to two different values.

Note that these rules certainly do not encompass every detail of Rust's intricate ownership system; they instead aim to represent the *core* concepts of ownership and borrowing in such a way that they can integrate into Granule's existing type system, leaving room for potential extensions. In particular, as was mentioned in Section 2 we only support *lexical* lifetimes, meaning that there is a class of programs which are accepted by Rust's compiler but cannot be represented in this calculus. We focus on being able to represent notions of ownership *explicitly* in the types, rather than using a complex static analysis like Rust's borrow checker to *infer* which programs are safe.

We also do not aim to capture any of the additional notions allowed by *unsafe* Rust (a superset of the safe portion of Rust we do consider) here. This is a natural choice, since while in Rust memory is managed through ownership and borrowing by default with memory for unsafe code needing to be managed manually, in languages like Granule managing memory automatically through a garbage collector is the default, while using our extension to obviate the need for garbage collection is a special case applying to some subset of a program. If the user does not wish for ownership to be taken into consideration, they can use Granule's pre-existing type system as they would before.

The following code demonstrates how all of these new primitives can be put together in a simple Granule program. This example makes use of a function called `observe` which operates over borrowed values; the precise behaviour of this function is elided, but it does not perform any unsafe behaviours such as mutation and so it can operate over values at any permission.

```
1  observe : ∀ {p : Permission} . & p Colour → & p Colour
2
3  persimmon : *Colour → *Colour
4  persimmon c = withBorrow (λb → let (x, y) = split b   in
5                                 let     x' = observe x in
6                                 let      b = join (x', y) in b) c
```

Granule

In `persimmon`, we *mutably* borrow the value `c` as `b`, split this into two *immutable* borrows `x` and `y`, and apply the `observe` function to `x` before rejoining the borrows and returning the value to its owner.

It should be clear that the unsafe patterns demonstrated by the `viridian` and `cerulean` examples in Section 2 are also unrepresentable in Granule. It is impossible for two mutable borrows or for a mutable borrow and an immutable borrow to coexist here, since splitting a borrow must reduce the permission in its type by the nature of the `split` primitive. We can, however, represent *reborrowing*; this simply involves continuing to split immutable borrows further, where recovering the original value now requires collecting all of the borrows once more. The following example illustrates this.

```
amethyst : *Colour → *Colour
amethyst c = withBorrow (λb → let (x, y) = split b in
                              let (l, r) = split x in
                              let     x' = join (l, r) in
                              let     b  = join (x', y) in b) c
```
Granule

When it comes to enforcing which behaviours should be allowed for different permissions on a given resource, this is mediated by said resource's interface. We will give one in-depth example in this paper, which is an interface for mutable arrays that can be created, read from, written to and deleted (each at differing levels of access); these will allow us to illustrate one of the key practical benefits of ownership, which is that unique access to an owned value allows for safe mutation (the original pun behind Wadler's "Linear Types can Change the World" [Wadler 1990]). Note, however, that it should be possible to define various other interfaces using the same system of ownership and borrowing that we describe. First, we extend our calculus with a primitive type of arrays:

$$A ::= \ldots \mid \mathsf{Array}_{id}\ A \mid \mathbb{N} \mid \mathbb{F} \qquad \text{(types, extended)}$$

where $\mathbb{N}$ are natural numbers used for sizes and indices and $\mathbb{F}$ are floating-point numbers, and arrays are parameterised by an identifier. Then, our interface for mutable arrays provides the following primitives (with built-in weakening):

$$\begin{array}{lll}
0 \cdot \Gamma \vdash & \textbf{newArray} & : \mathbb{N} \multimap \exists id.{*}(\mathsf{Array}_{id}\ \mathbb{F}) \\
0 \cdot \Gamma \vdash & \textbf{readArray} & : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\
0 \cdot \Gamma \vdash & \textbf{writeArray} & : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \quad (\textit{where } p \equiv 1 \ \vee \ p \equiv *) \\
0 \cdot \Gamma \vdash & \textbf{deleteArray} & : {*}(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}
\end{array}$$

In the above typing, *id* and *p* are metavariables for identifier types and permissions respectively. Thus, we treat these primitives as part of the typing rules rather than functions that are in scope.

This interface is similar to Granule's existing interface for array mutation involving only uniqueness types, but with some crucial differences owing to the introduction of borrowing. First, array types are now parameterised by an identifier pointing to a particular array value, which is generated during typechecking for each individual call to `newArray` (and within the scope of the CLONE rule). This allows us to keep track of which array is being referenced now that multiple borrows pointing to the same array are permitted to exist.

The other key difference is that reading and writing no longer require sole ownership. Reading can be carried out at any permission, since this is safe no matter how many references exist, but writing is restricted to either an owned array or a mutably borrowed array.

We then link with the development of Section 4 by the following type equality: $\&_* A \equiv {*}A$. This identity allows us to more easily define operations that operate over either owned or mutably borrowed values without needing multiple primitives. Note that addition and division are not defined at this permission, which prevents the use of SPLIT or JOIN.

Resource allocators. Here, the restriction introduced on the promotion rule in Section 3 meaning that 'resource allocators' are forbidden from being promoted becomes crucial for ensuring soundness in the call-by-value setting of this work. Consider the following snippet of Granule code (eliding packing and unpacking of existentials), which would be allowed given unrestricted promotion:

```
1  let [x] : ((*(Array id Float)) [2]) = [newArray 1] in
2  let                               () = deleteArray x in writeArray x 0 1.0
```

Granule ✗

On the first line, this program creates a reference `x` to a new array of size 1, but under a promotion, with the type explaining that we want to use the resulting value twice (given by the explicit type signature here). This promotion then allows two uses of the array on the second and third line.

Under a call-by-name semantics, as with the semantics used in previous work on embedding uniqueness types in Granule [Marshall et al. 2022] (and accessible in Granule via the extension `language CBN`), this program executes successfully and produces an array which contains the value written on the third line. The key is that call-by-name reduction substitutes the call to `newArray` into the two uses of the variable `x`, and so these point to two entirely separate arrays. However, under a call-by-value semantics like the semantics used in this paper (and Granule's default), the first line is fully evaluated, and so the reference `x` always points to the same array. This means that on the third line we are attempting to write to an array after it has been deleted, which will cause a runtime error. Thus, in a call-by-value setting this program must not be permitted.

The solution we apply here (originally developed in work on graded session types [Marshall and Orchard 2022b]) is that we syntactically restrict promotion to terms which do not allocate resources; here this includes terms which contain a call to `newArray` in a reduction position. The predicate resourceAllocator(t) classifies precisely these terms; note in particular that ¬resourceAllocator($\lambda x.$**newArray** t) since reduction does not happen underneath an abstraction (Section 6 defines the reduction semantics). The appendix includes the full inductive definition.

Linear Haskell. Other work resolves this same problem relating to promotion of resource allocators through different techniques. Originally, the linear types extension to Haskell got around this difficulty by only ever allocating resources inside a specialised continuation, which is passed around at every step until deallocation to prevent the allocator itself from ever being used non-linearly. More recently, this strategy has been generalised by introducing a "linear constraint" [Spiwack et al. 2022] called `Linearly`. This constraint, which must itself be used in a linear fashion, is assumed whenever a new resource is allocated. A continuation is still necessary for the initial assumption of `Linearly`, but the same qualification may now be used generically for varying resource types.

5.1 Equational theory

It is possible to examine the equational theory for the new $\&_1 A$ modality in much the same way as we earlier analysed the relationship between $*$ and $\Box_r$. Note that in particular, the WITH& rule suggests a functorial relationship between the two modalities involved; indeed, by analogy to the notion of a relative monad presented in Section 4, we can consider the idea that $*$ acts as a "relative functor" with regard to $\&_1 A$, where using WITH& equates to mapping a function involving mutable borrows onto an owned value. The following axioms for functors also hold here.

$$\textbf{withBorrow}\ (\lambda x.x)\ t \equiv t \qquad \text{(\&unit)}$$

$$\textbf{withBorrow}\ (\lambda x.f\ (g\ x))\ t \equiv \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t) \qquad \text{(\&assoc)}$$

The first axiom states that borrowing a reference to an owned value and simply returning it via the identity function without making use of it is equivalent to doing nothing at all, as one might

expect. The second axiom is an associativity axiom, giving us the result that borrowing a value to apply one function and then borrowing the value again to apply a second function is equivalent to simply borrowing the value once and applying the functions in sequence.

5.2 Divide and conquer: partial views via distributive laws

One particularly useful borrowing pattern when writing practical programs is the ability to take a composite data structure and borrow only *part* of it, such that the original owner retains access to the remaining structure. This introduces the possibility of multiple threads working with parts of a data structure in parallel, where it is no longer necessary to take ownership of the full data structure in order to mutate the only part for which access is required.

Our core calculus here provides some capacity for structured data in the form of product types; in order to allow for the pattern of partial borrowing inside a product, we introduce additional rules for distributing the borrow and uniqueness modalities into and out of pairs.[5]

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \textsc{push} \qquad \frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\ t : \&_p(A \otimes B)}\ \textsc{pull}$$

We can derive an operation akin to WITH& for borrowing one component of a product in the following way: use PUSH to move ownership onto the values inside the product, use WITH& to borrow the required component, and then once we are done, use PULL to recover the original pair.

To demonstrate how this concept works in action, consider the following example. Here, we use a simple function `transform` which operates over borrowed components of a `Colour`; again we elide the implementation but this time we assume the operation is mutating and so requires a whole permission. The strategy for partial borrowing here follows exactly the pattern described above.

```
1  transform : & 1 Int ⟶ & 1 Int
2
3  indigo : *Colour ⟶ *Colour
4  indigo c = let (r, p) = push c in
5             let     r' = withBorrow transform r in pull (r', p)
```

Granule

This notion extends gracefully to structures containing more than two components. Consider the case where we wish to borrow a single element from a unique tuple of three values, while leaving the remaining two values available for access by their original owner. First, note that in our core calculus the way to represent this is with a pair nested inside another pair; imagine that we are working with the type $*(A \otimes (B \otimes C))$ for the sake of example (the product could be associated in the opposite way, but these two types are isomorphic).

We can use the PUSH and PULL rules to achieve this result as before. Here, we may need to apply PUSH twice in order to distribute the modality over the entire product depending on which we are borrowing, but in the same way as above we can use WITH& to extract the desired data (for example, $\&_1 A$) while applying PULL to recover the remaining structure (in this case, $*(B \otimes C)$).

We now give another example to demonstrate this idea in practice. We also make use of Granule's pre-existing `par` combinator here, which takes two continuations and runs them concurrently in separate threads; this illustrates the idea that it is safe to simultaneously operate on disjoint components of a single data structure without risking a data race. The implementation of `par` involves making use of Granule's facility for linear session types [Marshall and Orchard 2022b]; this is not a focus of this paper and so we elide these details here.

[5] One might wonder whether the $\Box_r$ modality also distributes in this way; the answer is not in general, though some choices of semiring permit this behaviour. This question has been explored in depth in prior work [Hughes et al. 2021a].

```
par : ∀ {a : Type, b : Type} . (() → a) → (() → b) → (a, b)

indigo' : *Colour → *Colour
indigo' c = let (r, p) = push c in
        let   (g, b) = push p in
        let (r', b') = par (λ() → withBorrow transform r) (λ() → withBorrow transform b) in
        let       p' = pull (g, b') in pull (r', p')
```
Granule

Extending this core concept to the more expressive setting of Granule's full type system is future work, including ideas such as borrowing a value from within a structure of any algebraic data type, or borrowing 'slices' of multiple values simultaneously. One approach would be to extend the `push` and `pull` functionality in order to generically derive operations for pushing and pulling modalities over arbitrary data structures, following patterns described in prior work [Hughes et al. 2021b].

This general pattern for partial borrowing relates closely to McBride's notion of computing the *derivative* of a data type by finding its type of one-hole contexts [McBride 2001], where the derivative describes the structure that remains after borrowing one element. Under this interpretation, borrowing a slice of multiple values is akin to the idea of taking the derivative of a data type *with respect to* another type, which has recently become more salient [Marshall and Orchard 2022c].

6 SEMANTICS AND META-THEORY

We define an operational semantics here for the calculus which serves to further explain the details of arrays, mutation, copying, and borrowing. We adapt the approach of Choudhury et al. [2021] and Marshall et al. [2022] for giving an operational semantics to a graded system, with some degree of accounting for grades and references with the model in order to relate the dynamic semantics back to the static semantics of typing (Section 6.4). With the exception of Granule[6], much of the preceding work on operational models for graded systems is based on call-by-name. Here we opt for call-by-value, for the purpose of describing real-world practical functional languages; call-by-name is prohibitively expensive with unpredictable performance and poor interaction with side effects.

Before defining the operational semantics, we define notions of values and the runtime terms and typing required by the rules, as well as the syntax of heaps.

6.1 Preliminary definitions

Values. We first define the subset of terms that are *values* in the semantics, i.e., normal forms (terms that have no further reduction), via the grammar:

$$v ::= (v_1, v_2) \mid () \mid [v] \mid \lambda x.t \mid i \mid p \mid \mathbf{pack}\ \langle id, v \rangle \qquad \text{(value terms sub-grammar)}$$

including pairs of values, the unit value, boxed values, abstractions, natural numbers i, or primitives p which may be partially applied to other values, e.g., **newArray**, **readArray**, **readArray** v, etc.

Runtime terms and typing. We extend the syntax of terms with several runtime representations which appear only in the semantics (i.e., they cannot be written by users in programs):

$$t ::= \ldots \mid *t \mid \&t \mid \mathbf{unborrow}\ t \mid a \qquad \text{(runtime terms)}$$

where $*t$ represents unique terms, $\&t$ represents borrowed terms, **unborrow** t is the inverse to borrowed terms (used to implement withBorrow), and a are array references which are bound in the heap and act as a kind of abstract, opaque pointer.

[6]Granule can be configured to run with either a call-by-value or call-by-name semantics.

The syntactic category of values is extended to runtime values as:

$$v ::= \ldots \mid *v \mid \&v \mid \textbf{unborrow}\ v \mid a \qquad \text{(runtime values)}$$

In order to account for the runtime representation of arrays, the syntax of contexts is extended to include assumptions $a : \mathsf{Array}_{id}\ A$ which are treated as a different syntactic category of variables.

Context addition and scalar multiplication extend as follows:

$$\begin{aligned}(\Gamma, a : \mathsf{Array}_{id}\ A) + (\Gamma', a : \mathsf{Array}_{id}\ A) &= (\Gamma + \Gamma'), a : \mathsf{Array}_{id}\ A \\ r \cdot (\Gamma, a : \mathsf{Array}_{id}\ A) &= (r \cdot \Gamma), a : \mathsf{Array}_{id}\ A \qquad \text{(context operations)}\end{aligned}$$

A *runtime context* γ is a context containing only array references, i.e.:

$$\gamma ::= \emptyset \mid \gamma, a : \mathsf{Array}_{id}\ A \qquad \text{(runtime context)}$$

Runtime terms are typed by the following rules:

$$\frac{\gamma \vdash t : A}{0 \cdot \Gamma, \gamma \vdash *t : *A}\ \textsc{nec} \qquad \frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : \&_p A}\ \textsc{borrow} \qquad \frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \textbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash a : \mathsf{Array}_{id}\ A}\ \textsc{arr}$$

Configurations. In our model, a *configuration* comprises a pair of a *heap* H and a term t, written as $H \vdash t$ where the free variables $\mathsf{fv}(v) \in \mathsf{dom}(H)$, and $\mathsf{arrRefs}(v) \in \mathsf{dom}(H)$ where arrRefs extracts all array references a from the term.

Heaps. Heaps map program variables to values (our semantics does not use syntactic substitution), and also map array references to identifiers and identifiers to array values.

The syntax of heaps is given by:

$$H ::= \emptyset \mid H, x \overset{r}{\mapsto} v : A \mid H, a \overset{p}{\mapsto} id \mid H, id \mapsto \textbf{arr} \qquad \text{(heaps)}$$

Thus, a heap can be extended in three ways: (1) with an assignment of a program variable x to a value v, storing the grade of the variable (which comes from the typing) with the type of the value; (2) with an assignment of an array reference a to an identifier id with permission p; (3) with an assignment of an identifier id to an array value. Identifiers and array references are separated since we may need multiple array references to point to the same identifier, e.g., in the case of immutable borrows (one can think of identifiers like abstract memory addresses in the semantics).

Array terms in the heap are given by the following grammar, describing an empty array, or an array where value v is stored at position i:

$$\textbf{arr} ::= \mathsf{init} \mid \textbf{arr}[i] = v \qquad \text{(heap array terms)}$$

6.2 Single-step reduction

Single-step reductions in the operational semantics map source configurations to target configurations, with the judgment:

$$H_1 \vdash t_1 \rightsquigarrow_r H_2 \vdash t_2$$

where H_1 and H_2 are input and output heaps respectively, t_1 is the source term and t_2 the target. The grade r denotes the usage context of this rule.

We now go on to discuss the various reduction rules for our operational semantics in detail.

Lambda calculus. The λ-calculus core of the operational semantics has rules:

$$\frac{\exists r'.\ r' + s \sqsubseteq r}{H, x{\mapsto_r} v : A \vdash x \rightsquigarrow_s H, x{\mapsto_r} v : A \vdash v}\ \rightsquigarrow_{\text{VAR}} \qquad \frac{\Gamma \vdash v : A}{H \vdash (\lambda x.t)\ v \rightsquigarrow_s H, x{\mapsto_s} v : A \vdash t}\ \rightsquigarrow_{\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\text{APPL}} \qquad \frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\ t_2 \rightsquigarrow_s H' \vdash v\ t_2'}\ \rightsquigarrow_{\text{APPR}}$$

The rules for function application are fairly standard. In the $\rightsquigarrow_{\text{VAR}}$ rule, a variable x is reduced to a value v which was assigned to x in the heap. The annotation r is preserved in the output heap, with the side condition in the premise ensuring that the grade r will be enough to capture the usage s required by the reduction. In the $\rightsquigarrow_\beta$ rule, rather than using a substitution, the body term is the result under a heap extended with x assigned to the (typed) argument value v. This rule is where the grade s required for the reduction annotates x in the heap.

Existential types and names. The semantics of existentials is standard, with a beta rule:

$$\frac{}{H \vdash \mathbf{unpack}\ \langle id, x\rangle = \mathbf{pack}\ \langle id', v\rangle\ \mathbf{in}\ t \rightsquigarrow_s H[id'/id], x{\mapsto_r} v \vdash t}\ \rightsquigarrow_{\exists\beta}$$

and two standard congruence rules for pack and unpack (elided for brevity).

Tensors and units. Tensor products have the following rules for their introduction and elimination forms, with three congruence rules and one β-rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)}\ \rightsquigarrow_{\otimes\text{L}} \qquad \frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ (x,y) = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ (x,y) = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')}\ \rightsquigarrow_{\otimes\text{R}} \qquad \frac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{H \vdash \mathbf{let}\ (x,y) = (v_1, v_2)\ \mathbf{in}\ t \rightsquigarrow_s H, x{\mapsto_s} v_1 : A, y{\mapsto_s} v_2 : B \vdash t}\ \rightsquigarrow_{\otimes\beta}$$

Note that we evaluate pair constructors left-to-right. In the case of ($\rightsquigarrow_{\otimes\beta}$), we extend the heap with assignments for x and y to v_1 and v_2 respectively, continuing on with the body term t. We elide the rules for the unit type as they are similar.

Example 6.1. The following gives a reduction sequence for the term $(\lambda x.(x,y))\ (v,y)$ under a heap $H = y{\mapsto_2} v : A$, until a normal form is reached:

$$\begin{aligned}
& y{\mapsto_2} v : A \vdash (\lambda x.(x,y))\ (v,y) \\
(\rightsquigarrow_{APPL}, \rightsquigarrow_{\otimes R}, \rightsquigarrow_{VAR}) \rightsquigarrow_1\ & y{\mapsto_2} v : A \vdash (\lambda x.(x,y))\ (v,v) \\
(\rightsquigarrow_{\beta}) \rightsquigarrow_1\ & y{\mapsto_2} v : A, x{\mapsto_1} (v,v) : (A \otimes A) \vdash (x,y) \\
(\rightsquigarrow_{\text{APPL}}, \rightsquigarrow_{\otimes\text{L}}, \rightsquigarrow_{\text{VAR}}) \rightsquigarrow_1\ & y{\mapsto_2} v : A, x{\mapsto_1} (v,v) : (A \otimes A) \vdash ((v,v), y)
\end{aligned}$$

Graded modalities. The rules for graded modalities are structured similarly to the standard lambda calculus rules and rules for tensor products seen above, but we need to do some additional management of grades in the heap.

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ [x] = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

$$\frac{\begin{array}{l} H \vdash t \rightsquigarrow_{s*r} H' \vdash t' \\ \Gamma \vdash [t] : \Box_r A \end{array}}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']}\ \rightsquigarrow_{\Box} \qquad \frac{\Gamma \vdash [v] : \Box_r A}{H \vdash \mathbf{let}\ [x] = [v]\ \mathbf{in}\ t \rightsquigarrow_s H, x{\mapsto_{(s*r)}} v : A \vdash t}\ \rightsquigarrow_{\Box\beta}$$

In the $\rightsquigarrow_\Box$ rule, to construct a reduction with the required grade s then we need to be able to reduce inside the box at grade $s * r$, to account for the additional usage required by the modality's r grade.

In the $\leadsto_{\Box\beta}$ rule, the x in the heap is annotated not only with s as in the regular $\leadsto_\beta$ rule but with $s * r$, again to account for the additional usage the modality requires.

Arrays. New arrays are created by the **newArray** primitive, with reduction rule:

$$\frac{a\#H \qquad id\#H}{H \vdash \mathbf{newArray}\ n \leadsto_s H, a{\mapsto_1} id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\ \langle id, *a\rangle} \leadsto_{\text{NEWARRAY}}$$

where the premises state that a and id are fresh for the heap H. Note that the resulting array is initialised to the empty array (init), and the result is a unique array reference $*a$. Note that, in the heap, a is marked with the whole permission 1.

Arrays can be read in their unique or borrowed forms as:

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (*a)\ i \leadsto_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, *a)} \leadsto_{\text{READARRAY}}$$

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (\&(*a))\ i \leadsto_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, \&(*a))} \leadsto_{\text{READ\&ARRAY}}$$

Arrays can be written similarly in unique or borrowed forms. For brevity we show just the unique form, but the borrowed form is very similar to the unique form, with the difference being similar to the array reading rules seen above:

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (*a)\ i\ v \leadsto_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash *a} \leadsto_{\text{WRITEARRAY}}$$

(p here should always be 1 or $*$, but this is mediated by the type system rather than being enforced in the semantics; the following rule is similar in this regard.)

Unique arrays are deleted via the rule:

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\ (*a) \leadsto_s H \vdash ()} \leadsto_{\text{DELETEARRAY}}$$

Note that we do not explicitly track the sizes of arrays in our semantics, for simplicity.

Sharing and cloning.

$$\frac{H \vdash t \leadsto_s H' \vdash t'}{H \vdash \mathbf{share}\ t \leadsto_s H' \vdash \mathbf{share}\ t'} \leadsto_{\text{SHARE}} \qquad \frac{\mathrm{dom}(H) \equiv \mathrm{arrRefs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \leadsto_s ([H]_0), H' \vdash [v]} \leadsto_{\text{SHARE}\beta}$$

$$\frac{\Gamma \vdash [v] : \Box_r A \qquad \mathrm{dom}(H') \equiv \mathrm{arrRefs}(v) \qquad (H'', \theta, \overline{id}) = \mathrm{copy}(H')}{H, H' \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \leadsto_s H, H', H'', x{\mapsto_s} \mathbf{pack}\ \langle \overline{id}, *(\theta(v))\rangle : *A \vdash t_2} \leadsto_{\text{CLONE}\beta}$$

In the $\leadsto_{\text{SHARE}\beta}$ rule, the incoming heap is split into two parts, where H is such that it provides the allocations for all array references in v (enforced by the premise). The unique value $*v$ is wrapped in the graded box modality in the result as $[v]$, and thus all its array references are now annotated with 0 in the heap via $([H]_0)$, e.g.:

$$H', id \mapsto \mathbf{arr}, a{\mapsto_1} id \vdash \mathbf{share}\ (*a) \rightsquigarrow H', id \mapsto \mathbf{arr}, a{\mapsto_0} id \vdash [a] \mid \emptyset$$

The $\leadsto_{\text{CLONE}\beta}$ rule enacts a 'deep copy', where $\mathrm{dom}(H') \equiv \mathrm{arrRefs}(v)$ marks the part of the heap with array references coming from v. Then $\mathrm{copy}(H')$ copies the arrays in this part of the heap, creating a heap fragment H'' and a renaming operator θ which maps from old array references to new copied references. This renaming is applied to v in the freshly bound unique variable x, such that the value $*(\theta(v))$ refers to any newly copied arrays. Lastly, we pack the renamed unique value with new identifiers $\overline{id}$ generated by copy.

We elide the straightforward congruence rule for **clone**.

Borrowing. We elide the congruence rules for withBorrow which ensures that we reduce the two argument terms left to right until they are values, after which we can reduce as follows:

$$\frac{}{H \vdash \textbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H \vdash \textbf{unborrow}\ ([\&(*v)/x]t)}\ \rightsquigarrow_{\text{WITH\&}}$$

Here, the beta reduction that comes from applying the function $\lambda x.t$ to the value v is enacted as a substitution, with the resulting value being wrapped inside the & modality as a representation of the fact that within the context of withBorrow it is now a borrowed term.

For a term to escape withBorrow it must eventually be 'unborrowed'; the runtime term unborrow encapsulating the resulting term represents this idea, and obeys the following rules:

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \textbf{unborrow}\ t \rightsquigarrow_s H \vdash \textbf{unborrow}\ t'}\ \rightsquigarrow_{\text{UNBORROW}} \qquad \frac{}{H \vdash \textbf{unborrow}\ (\&(*v)) \rightsquigarrow_s H \vdash *v}\ \rightsquigarrow_{\text{UN\&}}$$

Aside from the congruence, which is standard, the $\rightsquigarrow_{\text{UN\&}}$ rule simply unwraps the value from the & modality, allowing the semantics to treat said value as a unique term once more.

Split and join.

$$\frac{\#a_1 \qquad \#a_2}{H, id \mapsto \textbf{arr}, a \mapsto_p id \vdash \textbf{split}\ (\&(*a)) \rightsquigarrow_s H, id \mapsto \textbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id \vdash (\&(*a_1), \&(*a_2))}\ \rightsquigarrow_{\text{SPLITARR}}$$

$$\frac{\begin{array}{c} H \vdash \textbf{split}\ (\&(*v)) \rightsquigarrow_s H' \vdash (\&(*v_1), \&(*v_2)) \\ H' \vdash \textbf{split}\ (\&(*w)) \rightsquigarrow_s H'' \vdash (\&(*w_1), \&(*w_2)) \end{array}}{H \vdash \textbf{split}\ (\&(*(v, w))) \rightsquigarrow_s H'' \vdash (\&(*(v_1, w_1)), \&(*(v_2, w_2)))}\ \rightsquigarrow_{\text{SPLIT}\otimes}$$

$$\frac{\#a}{H, id \mapsto \textbf{arr}, a_1 \mapsto_p id, a_2 \mapsto_q id \vdash \textbf{join}\ (\&(*a_1))\ (\&(*a_2)) \rightsquigarrow_s H, id \mapsto \textbf{arr}, a \mapsto_{(p+q)} id \vdash \&(*a)}\ \rightsquigarrow_{\text{JOINARR}}$$

$$\frac{\begin{array}{c} H \vdash \textbf{join}\ (\&(*v_1))\ (\&(*v_2)) \rightsquigarrow_s H' \vdash (\&(*v)) \\ H' \vdash \textbf{join}\ (\&(*w_1))\ (\&(*w_2)) \rightsquigarrow_s H'' \vdash (\&(*w)) \end{array}}{H \vdash \textbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) \rightsquigarrow_s H'' \vdash \&(*(v, w))}\ \rightsquigarrow_{\text{JOIN}\otimes}$$

Join and split also have congruence rules which are straightforward and elided.

There are two primary reduction rules for each ofsplit and join: one for the case where the terms are array references and one for the case where the terms are pairs of values.

In the array reference case, split removes the initial array reference from the heap and generates two fresh references pointing to the same identifier. These are each annotated with half of the permission belonging to the original reference, to match the typing. As one might expect, join for array references behaves dually; it deletes two existing references to an array from the heap, and generates one fresh reference, with its permission being the sum of the constituent parts.

When it comes to pairs, the reduction rules are defined in an inductive fashion; as long as we can split or join on the two components of the pair, we are allowed to reduce on the overall pair itself. In this way, we construct a reduction which ensures that all array references contained within the pair are split or joined as required, no matter how deeply nested the pair may be.

Push and pull. The reduction rules for push and pull are fairly simple; they simply distribute the modality into or out of the product term in a way that matches the typing of the given rule.

$$\frac{}{H \vdash \textbf{push}\ *(v_1, v_2) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\ \rightsquigarrow_{\text{PUSH}*} \qquad \frac{}{H \vdash \textbf{pull}\ (*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)}\ \rightsquigarrow_{\text{PULL}*}$$

The heap is left unchanged. The above rules are for unique terms; there are two equivalent rules for push and pull on borrowed terms, but these are identical aside from the modality on the term.

Push and pull also have congruence rules which are straightforward and elided.

6.3 Multi-step reductions

Lastly, as a convenience, we define a relation that composes single-step reductions into a sequence of reductions, called a multi-reduction, with the obvious rules for chaining together reductions:

$$\frac{}{H \vdash t \;\Rightarrow_s\; H \vdash t}\;\textsc{refl} \qquad \frac{H \vdash t_1 \;\rightsquigarrow_s\; H' \vdash t_2 \qquad H' \vdash t_2 \;\Rightarrow_s\; H'' \vdash t_3}{H \vdash t_1 \;\Rightarrow_s\; H'' \vdash t_3}\;\textsc{ext}$$

6.4 Theorems

We now move on to understanding the useful properties that hold for our type system, particularly with regard to how it interacts with the operational semantics we have described thus far. First, we must build a foundation via checking some preliminary results and noting some crucial definitions.

Substitution. Two useful results, which extend those found elsewhere in the literature, are that substitution is admissible for our calculus, coming in both linear and graded variants:

Lemma 6.2 (Linear substitution is admissible, extending [Orchard et al. 2019]). *If* $\Gamma_1 \vdash t_1 : A$ *and* $\Gamma_2, x : A \vdash t_2 : B$ *then* $\Gamma_2 + \Gamma_1 \vdash [t_1/x]t_2 : B$.

Lemma 6.3 (Graded substitution is admissible, extending [Orchard et al. 2019]). *If* $[\Gamma_1] \vdash t_1 : A$ *and* $\Gamma_2, x : [A]_r \vdash t_2 : B$ *and* $\neg\mathsf{resourceAllocator}(t_1)$ *then* $\Gamma_2 + r \cdot \Gamma_1 \vdash [t_1/x]t_2 : B$.

Type safety. Key to ensuring type safety is the notion of *heap compatibility with a typing context.*

Definition 6.4 (Heap compatibility). A heap H is compatible with free variable context Γ, denoted $H \bowtie \Gamma$, if the grades in the heap match those of the context, and any variables stored in the heap have their resources accounted for in the rest of the heap too. The relation is defined inductively over the syntax of heaps and contexts:

$$\frac{}{\emptyset \bowtie \emptyset}\;\textsc{empty} \qquad \frac{H, id \mapsto \mathbf{arr} \bowtie \Gamma}{H, a \mapsto_p id, id \mapsto \mathbf{arr} \bowtie (\Gamma, a : \mathsf{Array}_{id}\, A)}\;\textsc{extRef} \qquad \frac{H \bowtie \emptyset}{H, a \mapsto_p id \bowtie \emptyset}\;\textsc{GCArr}$$

$$\frac{H \bowtie \Gamma + r \cdot \Gamma' \qquad x \notin \mathrm{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\, r + r' \equiv s}{(H, x \mapsto_s v : A) \bowtie (\Gamma, x : [A]_r)}\;\textsc{ext}$$

$$\frac{H \bowtie \Gamma + \Gamma' \qquad x \notin \mathrm{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\, 1 + r' \equiv s}{(H, x \mapsto_s v : A) \bowtie (\Gamma, x : A)}\;\textsc{extLin}$$

Thus, a context extended with a runtime type of an array reference (extRef) is compatible with a heap which contains that array reference, pointing to some array with the corresponding identifier that the array reference points to. In the premise, the array (with its identifier) is preserved in the heap since there may be other references pointing to it (e.g., generated from a split). The (GCArr) rule then allows heap compatibility to 'garbage collect' any remaining arrays.

The (ext) rule says that a context with graded assumption $x : [A]_r$ is compatible with a heap as long as the heap contains a binding for x to some value v and as long as the heap grade s can accommodate the usage of r (via the constraint $\exists r'.\, r + r' \equiv s$) and as long as the free variables Γ' of v are also compatible with the heap (scaled by r to reflect the usage of x). The (extLin) rule is similar to (ext), but *effectively* where $r = 1$—the variable x is used in a linear fashion.

The following example aims to illustrate the idea of heap compatibility in more detail.

Example 6.5. The context $x : [A]_1, y : [B]_2$ is compatible with the heap $x{\mapsto_7} v_1 : A, y{\mapsto_2} v_2 : B$ assuming the typing $x : [A]_3 \vdash v_2 : B$, with heap compatibility derivation:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \bowtie \emptyset}\ \textsc{empty}}{\emptyset, x{\mapsto_7} v : A \bowtie x : [A]_7}\ \textsc{ext} \qquad x : [A]_3 \vdash v_2 : B}{(\emptyset, x{\mapsto_7} v_1 : A, y{\mapsto_2} v_2 : B) \bowtie (x : [A]_1, y : [B]_2)}\ \textsc{ext}$$

These definitions allow us to establish key properties for our calculus in conjunction with the operational semantics. We begin with syntactic type safety, by verifying progress and preservation. These are largely standard, though preservation links with heap compatibility in its second conjunct.

Theorem 6.6 (Progress). *Given $\Gamma \vdash t : A$, then t is either a value, or if $H \bowtie \Gamma_0 + \Gamma$ there exists a heap H', term t', grade s, such that $H \vdash t \rightsquigarrow_s H' \vdash t'$.*

Theorem 6.7 (Type Preservation). *For a well-typed term $\Gamma \vdash t : A$ and all s, Γ_0, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ and a reduction $H \vdash t \rightsquigarrow_s H' \vdash t'$ we have:*

$$\exists \Gamma', H'.\ \ \Gamma' \vdash t' : A \ \ \wedge \ \ H' \bowtie (\Gamma_0 + s \cdot \Gamma')$$

In addition to preservation, Marshall et al. [2022] also proved a *conservation* property, ensuring that resource usage is respected (in particular, that resource usage accrued in a given reduction plus remaining resources in the resulting heap are approximated by the resources in the original heap plus the specified resource usage from any variable bindings encountered along the way). Their semantics was call-by-name, which is more natural for conservation; checking that a modified form of conservation still holds in our setting is future work, though various other graded type systems with similar properties are also call-by-value [Bianchini et al. 2022; Orchard et al. 2019].

We now move on to establishing perhaps the most interesting properties here, which are related to the safety of ownership and borrowing and how these notions relate to the array references present in the heap. First, we ensure that taking a single step preserves the total of fractional annotations on array references (unless we have stopped tracking ownership information for the given reference, in which case it should be annotated with 0):

Lemma 6.8 (Borrow safety). *For a well-typed term $\Gamma \vdash t_1 : A$, consider all types $\&_p A' \in A$ (subformulas of A with the form $\&_p A'$ for some A'). Then for all $id \in A'$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$, and given a single-step reduction $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$ then for all $a \in \mathsf{arrRefs}(t_1)$ (array references in t_1) such that $H(a) = id$ we have:*

$$\sum_p a{\mapsto_p} id \in H = \sum_p a{\mapsto_p} id \in H' = 1 \quad \vee \quad \sum_p a{\mapsto_p} id \in H' = 0$$

i.e., either *any array references contributing to the final term have their total fraction of* 1 *preserved from the incoming heap to the resulting term,* or *the total fraction in the resulting term is* 0 *(we have stopped tracking ownership information for the given identifier).*

Also, for all $a \in \mathsf{arrRefs}(t_1')$ (array references in t_1') such that $a \notin \mathsf{dom}(H)$ we have:

$$\exists id'. \sum_q a{\mapsto_q} id' \in H' = 1$$

i.e., any new array references contributing to the final term also have total fractions summing to 1.

When we extend this result to multi-step reductions, we are able to restrict the result further, and prove that if the resulting term is of unique type, then not only is the sum of fractions preserved

throughout the overall reduction (which we verify inductively using the above result) but also that in the final term the reference to the resource we are considering must itself be unique.

Lemma 6.9 (Borrow safety over multi-reduction). *For a well-typed term* $\Gamma \vdash t_1 : *A$*, then for all* $id \in A$ *and all* Γ_0 *and heaps* H *such that* $H \bowtie (\Gamma_0 + \Gamma)$*, and given a multi-step reduction* $H \vdash t_1 \Rightarrow_s H' \vdash v$ *then for all* $a \in \mathsf{arrRefs}(t_1)$ *(array references in* t_1*) such that* $H(a) = id$ *we have:*

$$\sum_p a \mapsto_p id \in H = 1 \implies \exists a'.a' \mapsto_1 id \in H'$$

i.e., any array references contributing to the final term have their total fraction of 1 *preserved from the incoming heap to the resulting term, with this fraction now contained in a single reference.*

Also, for all $a \in \mathsf{arrRefs}(t_1')$ *(array references in* t_1'*) such that* $a \notin \mathsf{dom}(H)$ *we have:*

$$\exists id'.a \mapsto_1 id' \in H'$$

i.e., any new array references contributing to the final term uniquely point to their identifier, and thus are annotated with the fraction 1*.*

The uniqueness theorem presented by Marshall et al. [2022] now follows as a direct corollary of the multi-step borrow safety theorem we described above.

Corollary 6.10 (Uniqueness). *For a well-typed term* $\Gamma \vdash t : *A$ *and all* Γ_0 *and* H *such that* $H \bowtie (\Gamma_0 + \Gamma)$ *and given a multi-reduction to a value* $H \vdash t \Rightarrow_s H' \vdash *v$*, for all* $a \in \mathsf{arrRefs}(v)$ *(array references in* v*) we have:*

$$a \mapsto_1 id \in H \implies a \mapsto_1 id \in H' \quad \wedge \quad a \notin \mathsf{dom}(H) \implies \exists id'.a \mapsto_1 id' \in H'$$

i.e., any array references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new array references contributing to the final term are also unique.

Finally, we see that the operational semantics we have defined (extended to full β-reduction) supports the equational theory we have gradually developed throughout this work.

Theorem 6.11 (Soundness with respect to the equational theory). *For all* t_1, t_2 *such that* $\Gamma \vdash t_1 : A$ *and* $\Gamma \vdash t_2 : A$ *and* $t_1 \equiv t_2$ *and given* H *such that* $H \bowtie \Gamma$*, there exists a value (irreducible term)* v *such that there are full* β*-reductions to the same value*

$$H \vdash t_1 \Rightarrow_\beta H' \vdash v \quad \wedge \quad H \vdash t_2 \Rightarrow_\beta H'' \vdash v$$

where full β*-reduction includes all congruences to evaluate inside* λ*s, etc.*

Full proofs for the above theorems may be found in the appendices for this paper, provided as supplementary material, along with collected typing and reduction rules.

7 RELATED WORK

7.1 Linear and graded types

The notion of linearity in programming originated with Girard's linear logic [Girard 1987], which builds a foundation for treating information in a resourceful way by restricting the structural rules of standard intuitionistic logic. This was soon adopted by programming language researchers who were interested in representing the resourceful behaviour of data, rapidly developing into the concept of *linear types* [Wadler 1990, 1993], but it took a substantial amount of time for them to make their way into the realm of practical programming. Lately, linear types have seen a renaissance due to their adoption as an extension to the Haskell language [Bernardy et al. 2017]; other languages

incorporating pure linearity (or affine types, which are similar but allow weakening) include ATS [Zhu and Xi 2005] and Alms [Tov and Pucella 2011].

This binary view was later refined by the notion of *bounded* linear logic [Girard et al. 1992], which introduces a family of operators indexed by a polynomial giving an upper bound on the usage of a resource. Further generalisations of this idea to be able to track a broader range of increasingly fine-grained properties are what led to the introduction of *graded* types, which allow for annotating values with precise information about how they interact with their environment.

Granule's particular approach to graded modalities draws heavily from the literature on *coeffects*, which describe how programs *depend* on their context [Petricek et al. 2014], though it also incorporates ideas from the literature on effect systems which we focus on less in this paper. This was developed at a similar time to other work which approached the same target from a different angle–through attempting to find explicit generalisations for bounded linear logic [Ghica and Smith 2014]. Other significant instantiations of graded types include Quantitative Type Theory (QTT) [Atkey 2018] upon which the type system for Idris 2 is built [Brady 2021], the core calculus underlying Linear Haskell [Bernardy et al. 2017; Spiwack et al. 2022], and various others [Abel and Bernardy 2020; Gaboardi et al. 2016; Wood and Atkey 2022].

7.2 Uniqueness types

Uniqueness types are most well known for their appearance in the Clean language [Smetsers et al. 1994], where they are used in lieu of monadic computation and for the efficiency gains offered by in-place update. In Clean, computation is based on graph rewriting and reduction; constants such as numbers are graphs, and functions are graph rewriting formulas. This gives the type system a different feel to those offered by more recent functional programming languages. Various more recent work has attempted to capture the benefits of uniqueness while allowing a more modern style of programming; examples include Cogent [O'Connor et al. 2021], Mercury [Somogyi et al. 1996] and the prior iteration of uniqueness types in Granule [Marshall et al. 2022].

Theoretical work on understanding uniqueness began with Harrington's uniqueness logic in 2006 [Harrington 2006]; this was followed by a substantial amount of theoretical groundwork for Clean's particular strategy for uniqueness typing, in works by de Vries and others [de Vries et al. 2008]. These papers aim to clarify the distinction between Clean's type system and systems based on the λ-calculus. Further work made headway on the problem of distinguishing uniqueness from other substructural systems, allowing for applications such as polymorphic programming and concurrency [de Vries 2013; de Vries et al. 2009]. This work laid the groundwork for Granule's somewhat orthogonal approach to uniqueness, allowing for uniqueness and linearity to be integrated in a single system, which eventually lead to the extensions developed in the present work.

7.3 Region-based memory management

The notion of regions [Tofte et al. 2004] was conceived with the aim of bringing some of the benefits of traditional stack-based memory management into higher-order functional languages. Regions divide values based on lifetimes, and in a similar fashion to more modern ownership-based systems they eliminate the need for garbage collection by using region type information to allow for safe allocation and deallocation. Historically, region type systems have typically been used for effect systems [Jouvelot and Gifford 1991; Lucassen and Gifford 1988].

Later work on regions (for example, work on static capabilities [Walker et al. 2000]) extends this stack-based foundation by making use of uniqueness information; a unique reference to a region ensures that the region has no aliases, and as such it can be deallocated efficiently. Regions themselves, similarly to lifetimes in ownership-based systems, can be thought of as equivalence

classes for a "may alias" relation; values which do not share a region are not permitted to alias with one another, and so if a value does not share a region with any other then it can be safely mutated.

The most well-known concrete implementation of regions is Cyclone, a 'safe dialect' of C [Hicks et al. 2004]. Work on understanding the theory underpinning Cyclone demonstrated the relationship between references and regions, observing that "unique pointers are essentially lightweight, dynamic regions that hold exactly one object [Fluet et al. 2006]." Rust's lifetimes themselves were heavily inspired by work on regions. An extension of ML called Affe [Radanne et al. 2020] has also been developed which supports both linearity and borrowing using regions.

7.4 Ownership and borrowing

Ownership was first developed as a framework for understanding aliasing in object-oriented languages [Mycroft and Voigt 2013] receiving particular attention from around 1998 1998 [Clarke et al. 1998], with related work in this area having introduced concepts such as islands [Hogg 1991], balloon types [Almeida 1997] and external uniqueness [Clarke and Wrigstad 2003]. The intent of ownership systems is to give a high-level structural view of objects and references, in a similar fashion to the way that type systems allow for a high-level structural view of data.

More recently, ownership ideas are primarily known due to being pervasively used in the Rust programming language in order to help ensure memory safety. Multiple formalisations for Rust's intricate ownership and borrowing system have been attempted; RustBelt [Jung et al. 2017] gives a lower-level encoding of Rust intended for formal verification while Oxide [Weiss et al. 2019] is a higher-level encoding designed for more theoretical work, among others [Jung et al. 2019; Pearce 2021]. Rust is not the only modern language to make use of ownership, however; other languages such as Swift incorporate similar ideas, and work on incorporating ownership into existing languages with manual memory management is ongoing [Sammler et al. 2021]. Lorenzen et al. [2023] describe how to use ideas related to ownership and borrowing to determine which functional programs can be executed purely in-place without requiring allocation.

8 CONCLUSIONS AND FUTURE WORK

8.1 Future work

Performance improvements. This paper has generalised the connection between uniqueness and linearity in order to develop a unified framework for reasoning about ownership and grading. While we have an implementation of the groundwork for this unified calculus built atop the Granule language compiler, the practical benefits this implementation can offer for resourceful programming have not yet been fully explored. We hope to evaluate the potential **performance improvements** that introducing precise ownership tracking into a functional language can offer through reducing time that needs to be spent on garbage collection for some subset of a program, by implementing a set of examples involving ownership and borrowing and benchmarking them against equivalent code in the standard functional style.

Guarantees. Linearity is only one example of a property that can be tracked via *coeffects*, the flavour of grading we consider here; many others have been described in prior work [Petricek et al. 2014]. If uniqueness dualises linearity, it stands to reason that translating the same relationship onto other coeffects may result in other interesting properties that can be tracked in a similar manner to uniqueness and ownership; for example, it has already been noted that in the realm of information flow tracking for security, *confidentiality* can be understood as a coeffect with *integrity* as its respective dual [Marshall and Orchard 2022a]. It would be valuable to find an analogy for the ownership generalisation developed here in the context of security, and also to eventually go further and develop a general algebraic theory for **guarantees** about past program behaviour.

Counting permissions. Fractional permissions and the general notion of dividing a single mutable borrow into many immutable borrows is not the only model for representing borrowing that we considered. One interesting path for future research would be to explore an alternative strategy for graded uniqueness that is less symmetrical but instead privileges the original owner, allowing for a more exact count of other extant references. This model, based on **counting permissions** [Bornat et al. 2005], involves having many references with only read permissions, and a designated owner that keeps a count of how many such references exist, so that uniqueness can still be recovered.

Non-lexical lifetimes. The model of ownership developed in this work was explicitly not intended to be a complete model of every aspect of Rust's ownership system, instead aiming to extract the essential features and develop a framework through which they can coexist with more traditional linear and graded types. It would still be a valuable avenue to pursue these details further and capture some of Rust's more advanced features, however–in particular, incorporating notions such as **non-lexical lifetimes** would increase the power of the system described here. This would also allow for a potential encoding study between Granule's core calculus and one of the more faithful representations of Rust's type system such as λ_{Rust} [Jung et al. 2017].

Categorical model. Finally, alongside the operational heap semantics model presented in this paper it would be interesting to explore a **categorical model** based on adjunctions. Benton's linear/non-linear (LNL) logic [Benton 1995] is well-known, and progress has been made on using similar tools to understand more advanced substructural systems involving graded types [Eades III and Orchard 2020], much like the core calculus of Granule covered in this paper. Despite the close theoretical relationship between linearity and uniqueness, the categorical background of uniqueness types has been only briefly explored [Harrington 2006], and for more complex systems of ownership and borrowing even less so. This would be a fruitful pathway for further research.

8.2 Conclusion

Graded type systems and ownership with borrowing are both ways of carefully and precisely managing the usage of data built upon the theoretical foundations of linear logic, but these two approaches have developed through very different pathways on the road to being incorporated in modern-day practical programming languages. In this work, we have developed a core calculus that captures many of the key concepts for ownership tracking in a graded setting, connecting these fine-grained substructural notions with prior work on simpler systems such as linear and uniqueness types which sit closer to the theory. This has allowed us to not only better understand the relationship between these disparate approaches to resourceful reasoning but also to examine how they interact alongside one another.

By developing a framework through which both ownership properties and precise grades for reasoning about data can be tracked within the setting of the Granule language, we demonstrate that careful management of both resource and memory usage are not only compatible but complementary in a functional context. This paper represents one piece of two larger puzzles–one aiming to develop an in-depth theoretical understanding of Rust's comprehensive approach to memory management, and one aiming to expand the range of properties about programs that can be represented explicitly through graded types–and we look forward to seeing more ideas being shared (or, indeed, borrowed) across the boundary between these two closely related worlds in the future.

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP (2020), 90:1–90:28. https://doi.org/10.1145/3408972

Paulo Sérgio Almeida. 1997. Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer, 32–59.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not be Endofunctors. In *Foundations of Software Science and Computational Structures: 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 13*. Springer, 297–311.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 56–65.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612.

P Nick Benton. 1995. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *Computer Science Logic: 8th Workshop, CSL'94 Kazimierz, Poland, September 25–30, 1994 Selected Papers 8*. Springer, 121–135.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022. Coeffects for Sharing and Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 156 (oct 2022), 29 pages. https://doi.org/10.1145/3563319

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, 2003 Proceedings*. Springer, 55–72.

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. , 9:1–9:26 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *ESOP*, Vol. 8410. Springer, 351–370.

Pritam Choudhury, Harley Eades III, Richard A Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32.

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *ECOOP 2003–Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings 17*. Springer, 176–200.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Edsko de Vries. 2013. Modelling Unique and Affine Typing Using Polymorphism. In *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*. Springer-Verlag, Berlin, Heidelberg, 181–192. https://doi.org/10.1007/978-3-642-40355-2{_}13

Edsko de Vries, Adrian Francalanza, and Matthew Hennessy. 2009. Uniqueness Typing for Resource Management in Message-Passing Concurrency. In *Proceedings First International Workshop on Linearity, LINEARITY 2009, Coimbra, Portugal, 12th September 2009 (EPTCS, Vol. 22)*, Mário Florido and Ian Mackie (Eds.). 26–37. https://doi.org/10.4204/EPTCS.22.3

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–218.

Harley Eades III and Dominic Orchard. 2020. Grading Adjoint Logic. *arXiv preprint arXiv:2006.08854* (2020).

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.

Jean-Yves Girard, Andre Scedrov, and Philip J Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theoretical Computer Science* 97, 1 (1992), 1–66.

Dana Harrington. 2006. Uniqueness Logic. *Theoretical Computer Science* 354, 1 (2006), 24–41.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1029873.1029883

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. *SIGPLAN Not.* 26, 11 (nov 1991), 271–285. https://doi.org/10.1145/118014.117975

Jack Hughes, Daniel Marshall, James Wood, and Dominic Orchard. 2021a. Linear Exponentials as Graded Modal Types. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*.

Jack Hughes, Michael Vollmer, and Dominic Orchard. 2021b. Deriving Distributive Laws for Graded Linear Types. *Electronic Proceedings in Theoretical Computer Science* 353 (dec 2021), 109–131. https://doi.org/10.4204/eptcs.353.6

Pierre Jouvelot and David Gifford. 1991. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 303–310. https://doi.org/10.1145/99583.99623

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. In *ICFP'23*. ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/fp2-fully-in-place-functional-programming-2/ preprint.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Dhruv C Makwana and Neel Krishnaswami. 2019. NumLin: Linear Types for Linear Algebra. (2019).

Daniel Marshall and Dominic Orchard. 2022a. Graded Modal Types for Integrity and Confidentiality. In *17th Workshop on Programming Languages and Analysis for Security (PLAS 2022)*.

Daniel Marshall and Dominic Orchard. 2022b. Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types. *arXiv preprint arXiv:2203.12875* (2022).

Daniel Marshall and Dominic A Orchard. 2022c. How to Take the Inverse of a Type. In *Proceedings of 36th European Conference on Object-Oriented Programming (ECOOP 2022)*.

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*. Springer International Publishing Cham, 346–375.

Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. *Unpublished manuscript* (2001), 74–88.

Conor McBride. 2016. I Got Plenty o' Nuttin'. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 207–233.

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. *Programming Languages and Systems* 12648 (2021), 462.

Alan Mycroft and Janina Voigt. 2013. Notions of Aliasing and Ownership. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 59–83. https://doi.org/10.1007/978-3-642-36946-9_4

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: Uniqueness Types and Certified Compilation. *J. Funct. Program.* (2021).

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30.

David J Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (2021), 1–73.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly Bent to Free Us. *Proc. ACM Program. Lang.* 4, ICFP, Article 103 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408985

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 158–174.

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. 1994. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science*, Hans Jürgen Schneider and Hartmut Ehrig (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 358–379. https://doi.org/10.1007/3-540-57787-4_23

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* 29, 1 (1996), 17–64. https://doi.org/10.1016/S0743-1066(96)00068-4 High-Performance Implementations of Logic Programming Systems.

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (aug 2022), 28 pages. https://doi.org/10.1145/3547626

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. *SIGPLAN Not.* 46, 1 (Jan. 2011), 447–458. https://doi.org/10.1145/1925844.1926436

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*, Vol. 3. Citeseer, 5.

Philip Wadler. 1993. A Syntax for Linear Logic. In *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. 513–529. https://doi.org/10.1007/3-540-58027-1_24

David Walker, Karl Crary, and Greg Morrisett. 2000. Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (jul 2000), 701–771. https://doi.org/10.1145/363911.363923

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *arXiv preprint arXiv:1903.00982* (2019).

James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers Through Stateful Views. In *Practical Aspects of Declarative Languages*, Manuel V. Hermenegildo and Daniel Cabeza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–97.

A COLLECTED RULES

A.1 Typing

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 : C}\ \otimes_E$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 : B}\ 1_E$$

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{elim} \qquad \frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\ t : \Box_r A}\ \textsc{share} \qquad \frac{1 \sqsubseteq r \qquad \mathsf{noIDs}(\Gamma_1) \qquad \Gamma_1, \overline{id} \vdash t_1 : \Box_r A \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t_2 : \Box_r B}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone}$$

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{with}\&$$

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \textsc{split} \qquad \frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

$$\frac{\Gamma \vdash t : \&_p (A \otimes B)}{\Gamma \vdash \mathbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \textsc{push} \qquad \frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\ t : \&_p (A \otimes B)}\ \textsc{pull}$$

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t \rangle : \exists id.A[id/id']}\ \textsc{pack} \qquad \frac{\Gamma_1 \vdash t_1 : \exists id.A \qquad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ \langle id, x \rangle = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unpack}$$

Primitives.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{new}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{read}$$

$$\frac{p \equiv 1 \ \vee \ p \equiv *}{0 \cdot \Gamma \vdash \mathbf{writeArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{write}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{deleteArray} : \forall\ id.*(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}}\ \textsc{del}$$

Runtime typing.

$$\frac{\gamma \vdash t : A}{0 \cdot \Gamma, \gamma \vdash *t : *A}\ \textsc{nec} \qquad \frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash a : \mathsf{Array}_{id}\ A}\ \textsc{arr}$$

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : *(\mathsf{Array}_{id}\ A)}\ *\textsc{arr} \qquad \frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ A)}\ \&\textsc{arr}$$

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : \&_p A}\ \textsc{borrow} \qquad \frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \mathbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

Definition A.1 (Graded contexts). $[\Gamma]$ classifies those contexts which contain only graded variables:

$$\frac{}{[\emptyset]} \qquad \frac{[\Gamma]}{[\Gamma, x : [A]_r]}$$

Definition A.2 (Resource allocating terms). Predicate definition:

$$\frac{}{\mathsf{resourceAllocator}(\mathbf{newArray})}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ () = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ () = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((t_1, t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((t_1, t_2))}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t)}{\mathsf{resourceAllocator}([t])} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{share}\ t_1)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\mathbf{withBorrow}\ t_1\ t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((\mathbf{withBorrow}\ t_1\ t_2))}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{split}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{join}\ t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{join}\ t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{push}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{pull}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{pack}\ \langle id', t_1\rangle)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{unpack}\ \langle id', x\rangle = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{unpack}\ \langle id', x\rangle = t_1\ \mathbf{in}\ t_2)}$$

A.2 Reduction rules for heap semantics

$$\frac{\exists r'.\ r' + s \sqsubseteq r}{H, x{\mapsto_r} v : A \vdash x \rightsquigarrow_s H, x{\mapsto_r} v : A \vdash v}\ \rightsquigarrow_{\text{VAR}} \qquad \frac{\Gamma \vdash v : A}{H \vdash (\lambda x.t)\ v \rightsquigarrow_s H, x{\mapsto_s} v : A \vdash t}\ \rightsquigarrow_{\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\text{APPL}} \qquad \frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\ t_2 \rightsquigarrow_s H' \vdash v\ t_2'}\ \rightsquigarrow_{\text{APPR}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)}\ \rightsquigarrow_{\otimes\text{L}} \qquad \frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x,y) = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')}\ \rightsquigarrow_{\otimes\text{R}} \qquad \frac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{H \vdash \mathbf{let}\,(x,y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x{\mapsto_s} v_1 : A, y{\mapsto_s} v_2 : B \vdash t}\ \rightsquigarrow_{\otimes\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LETUNIT}} \qquad \frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \rightsquigarrow_s H \vdash t}\ \rightsquigarrow_{\beta\text{UNIT}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

$$\frac{\begin{array}{l}H \vdash t \rightsquigarrow_{s*r} H' \vdash t' \\ \Gamma \vdash [t] : \Box_r A\end{array}}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']}\ \rightsquigarrow_{\Box} \qquad \frac{\Gamma \vdash [v] : \Box_r A}{H \vdash \mathbf{let}\,[x] = [v]\,\mathbf{in}\,t \rightsquigarrow_s H, x{\mapsto_{(s*r)}} v : A \vdash t}\ \rightsquigarrow_{\Box\beta}$$

$$\frac{a\#H \qquad id\#H}{H \vdash \mathbf{newArray}\ n \rightsquigarrow_s H, a{\mapsto_1} id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\,\langle id, *a\rangle}\ \rightsquigarrow_{\text{NEWARRAY}}$$

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (*a)\ i \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, *a)}\ \rightsquigarrow_{\text{READARRAY}}$$

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (\&(*a))\ i \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, \&(*a))}\ \rightsquigarrow_{\text{READ\&ARR}}$$

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (*a)\ i\ v \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash *a}\ \rightsquigarrow_{\text{WRITEARRAY}}$$

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\ (*a) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\text{DELETEARRAY}}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\ t \rightsquigarrow_s H' \vdash \mathbf{share}\ t'}\ \rightsquigarrow_{\text{SHARE}} \qquad \frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\text{SHARE}\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ x\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\text{CLONE}}$$

$$\frac{\Gamma \vdash [v] : \Box_r A \qquad \mathrm{dom}(H') \equiv \mathrm{arrRefs}(v) \qquad (H'', \theta, \overline{id}) = \mathrm{copy}(H')}{H, H' \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \ \leadsto_s\ H, H', H'', x \mapsto_s \mathbf{pack}\ \langle \overline{id}, *(\theta(v)) \rangle : *A \vdash t_2}\ \leadsto_{\text{CLONE}\beta}$$

$$\frac{H \vdash t_1 \ \leadsto_s\ H' \vdash t_1'}{H \vdash \mathbf{withBorrow}\ t_1\ t_2 \ \leadsto_s\ H' \vdash \mathbf{withBorrow}\ t_1'\ t_2}\ \leadsto_{\text{WITH\&L}}$$

$$\frac{H \vdash t_2 \ \leadsto_s\ H' \vdash t_2'}{H \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2 \ \leadsto_s\ H' \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \leadsto_{\text{WITH\&R}}$$

$$\frac{}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \ \leadsto_s\ H \vdash \mathbf{unborrow}\ ([\&(*v)/x]t)}\ \leadsto_{\text{WITH\&}}$$

$$\frac{H \vdash t \ \leadsto_s\ H \vdash t'}{H \vdash \mathbf{unborrow}\ t \ \leadsto_s\ H \vdash \mathbf{unborrow}\ t'}\ \leadsto_{\text{UNBORROW}} \qquad \frac{}{H \vdash \mathbf{unborrow}\ (\&(*v)) \ \leadsto_s\ H \vdash *v}\ \leadsto_{\text{UN\&}}$$

$$\frac{H \vdash t \ \leadsto_s\ H' \vdash t'}{H \vdash \mathbf{split}\ t \ \leadsto_s\ H' \vdash \mathbf{split}\ t'}\ \leadsto_{\text{SPLIT}}$$

$$\frac{\#a_1 \qquad \#a_2}{H, id \mapsto \mathbf{arr}, a \mapsto_p id \vdash \mathbf{split}\ (\&(*a)) \ \leadsto_s\ H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id \vdash (\&(*a_1), \&(*a_2))}\ \leadsto_{\text{SPLITARR}}$$

$$\frac{\begin{array}{c} H \vdash \mathbf{split}\ (\&(*v)) \ \leadsto_s\ H' \vdash (\&(*v_1), \&(*v_2)) \\ H' \vdash \mathbf{split}\ (\&(*w)) \ \leadsto_s\ H'' \vdash (\&(*w_1), \&(*w_2)) \end{array}}{H \vdash \mathbf{split}\ (\&(*(v, w))) \ \leadsto_s\ H'' \vdash (\&(*(v_1, w_1)), \&(*(v_2, w_2)))}\ \leadsto_{\text{SPLIT}\otimes}$$

$$\frac{H \vdash t_1 \ \leadsto_s\ H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \ \leadsto_s\ H' \vdash \mathbf{join}\ t_1'\ t_2}\ \leadsto_{\text{JOINL}} \qquad \frac{H \vdash t_2 \ \leadsto_s\ H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \ \leadsto_s\ H' \vdash \mathbf{join}\ v\ t_2'}\ \leadsto_{\text{JOINR}}$$

$$\frac{\#a}{H, id \mapsto \mathbf{arr}, a_1 \mapsto_p id, a_2 \mapsto_q id \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) \ \leadsto_s\ H, id \mapsto \mathbf{arr}, a \mapsto_{(p+q)} id \vdash \&(*a)}\ \leadsto_{\text{JOINARR}}$$

$$\frac{\begin{array}{c} H \vdash \mathbf{join}\ (\&(*v_1))\ (\&(*v_2)) \ \leadsto_s\ H' \vdash (\&(*v)) \\ H' \vdash \mathbf{join}\ (\&(*w_1))\ (\&(*w_2)) \ \leadsto_s\ H'' \vdash (\&(*w)) \end{array}}{H \vdash \mathbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) \ \leadsto_s\ H'' \vdash \&(*(v, w))}\ \leadsto_{\text{JOIN}\otimes}$$

$$\frac{H \vdash t \ \leadsto_s\ H' \vdash t'}{H \vdash \mathbf{push}\ t \ \leadsto_s\ H' \vdash \mathbf{push}\ t'}\ \leadsto_{\text{PUSH}} \qquad \frac{H \vdash t \ \leadsto_s\ H' \vdash t'}{H \vdash \mathbf{pull}\ t \ \leadsto_s\ H' \vdash \mathbf{pull}\ t'}\ \leadsto_{\text{PULL}}$$

$$\frac{}{H \vdash \mathbf{push}\ *\ (v_1, v_2) \ \leadsto_s\ H \vdash (*v_1, *v_2)}\ \leadsto_{\text{PUSH}*}$$

$$\frac{}{H \vdash \mathbf{pull}\ (*v_1, *v_2) \ \leadsto_s\ H \vdash *(v_1, v_2)}\ \leadsto_{\text{PULL}*}$$

$$\frac{}{H \vdash \mathbf{push}\,\&(*(v_1, v_2)) \rightsquigarrow_s H \vdash (\&(*v_1), \&(*v_2))}\ \rightsquigarrow_{\text{PUSH\&}}$$

$$\frac{}{H \vdash \mathbf{pull}\,(\&(*v_1), \&(*v_2)) \rightsquigarrow_s H \vdash \&(*(v_1, v_2))}\ \rightsquigarrow_{\text{PULL\&}}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash *t \rightsquigarrow_s H' \vdash *t'}\ \rightsquigarrow_{*} \qquad \frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \&t \rightsquigarrow_s H' \vdash \&t'}\ \rightsquigarrow_{\&}$$

$$\frac{}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = \mathbf{pack}\,\langle id', v\rangle\ \mathbf{in}\ t \rightsquigarrow_s H[id'/id], x{\mapsto_r} v \vdash t}\ \rightsquigarrow_{\exists\beta}$$

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \mathbf{pack}\,\langle id, t\rangle \rightsquigarrow_s H \vdash \mathbf{pack}\,\langle id, t'\rangle}\ \rightsquigarrow_{\text{PACK}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H \vdash t_1'}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\text{UNPACK}}$$

Multi-reduction rules.

$$\frac{}{H \vdash t \Rightarrow_s H \vdash t}\ \text{REFL} \qquad \frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_2 \qquad H' \vdash t_2 \Rightarrow_s H'' \vdash t_3}{H \vdash t_1 \Rightarrow_s H'' \vdash t_3}\ \text{EXT}$$

B SUBSTITUTION PROOFS

Lemma B.1 (Linear substitution is admissible, extending [Orchard et al. 2019]). *If* $\Gamma_1 \vdash t_1 : A$ *and* $\Gamma_2, x : A \vdash t_2 : B$ *then* $\Gamma_2 + \Gamma_1 \vdash [t_1/x]t_2 : B$.

Proof. By induction on the typing derivation of t_2.

- (pr)

$$\frac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \text{PR}$$

  where $t_2 = [t]$. Trivial since the form of the typing does not match here: no linear variable possible.
- (share)

$$\frac{\Gamma_2, x : A \vdash t : *A}{\Gamma_2, x : A \vdash \mathbf{share}\ t : \Box_r A}\ \text{SHARE}$$

  where $B = \Box_r A$.
  By induction on the premise then $\Gamma_1 + \Gamma_2 \vdash [t_1/x]t : *A$, from which we build the conclusion:

$$\frac{\Gamma_2 \vdash [t_1/x]t : *A}{\Gamma_2 \vdash \mathbf{share}\ ([t_1/x]t) : \Box_r A}\ \text{SHARE}$$

- (bind) Two possibilities:

(1) Linear variable x in the left premise:

$$\frac{\Gamma_1', x : A \vdash t_1' : \Box_r A' \qquad \Gamma_2', y : *(\#A') \vdash t_2' : \Box_r B \qquad r \sqsubseteq 1}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ y\ \mathbf{in}\ t_2' : \Box_r B}\ \text{CLONE}'$$

By induction on the first premise: $\Gamma_1' + \Gamma_1 \vdash [t/x]t_1' : \Box_r A'$
Then we reconstruct the typing as:

$$\frac{\Gamma_1 + \Gamma_1 \vdash [t/x]t_1' : \Box_r A' \qquad \Gamma_2', y : *(\#A') \vdash t_2' : \Box_r B' \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{clone}\ [t/x]t_1'\ \mathbf{as}\ y\ \mathbf{in}\ t_2' : \Box_r B'}\ \textsc{clone}',$$

satisfying the goal (by commutativity of +).

(2) Linear variable x in the right premise:

$$\frac{\Gamma_1' \vdash t_1' : \Box_r A' \qquad \Gamma_2', x : A, y : *(\#A') \vdash t_2' : \Box_r B' \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ y\ \mathbf{in}\ t_2' : \Box_r B'}\ \textsc{clone}',$$

By induction on the second premise: $(\Gamma_2' + \Gamma_1), y : *A' \vdash [t/x]t_2' : \Box_r B'$
Then we reconstruct the typing as:

$$\frac{\Gamma_1' \vdash t_1' : \Box_r A' \qquad (\Gamma_2' + \Gamma_1), y : *(\#A') \vdash [t/x]t_2' : \Box_r B'}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ y\ \mathbf{in}\ [t/x]t_2' : \Box_r B'}\textsc{Tybind}$$

satisfying the goal.

- (withBorrow) Two possibilities:

(1) Linear variable in the first premise:

$$\frac{\Gamma_1', x : A \vdash t : *A' \qquad \Gamma_2' \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{withBorrow}\ f\ t : *B'}\ \textsc{with}\&$$

By induction on the first premise then $\Gamma_1' + \Gamma_1 \vdash [t_1/x]t : *A'$.
From this we construct the goal:

$$\frac{\Gamma_1' + \Gamma_1 \vdash [t_1/x]t : *A' \qquad \Gamma_2' \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{withBorrow}\ f\ [t_1/x]t : *B'}\ \textsc{with}\&$$

satisfying the goal by commutativity of +.

(2) Linear variable in the second premise:

$$\frac{\Gamma_1' \vdash t : *A' \qquad \Gamma_2', x : A \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{withBorrow}\ f\ t : *B'}\ \textsc{with}\&$$

By induction on the second premise then $\Gamma_2' + \Gamma_1 \vdash [t_1/x]f : \&_1 A' \multimap \&_1 B'$.
From this we construct the goal:

$$\frac{\Gamma_1' \vdash t : *A' \qquad \Gamma_2' + \Gamma_1 \vdash [t_1/x]f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{withBorrow}\ [t_1/x]f\ t : *B'}\ \textsc{with}\&$$

satisfying the goal.

- (split)

$$\frac{\Gamma, x : A \vdash t : \&_{p+q} A}{\Gamma, x : A \vdash \mathbf{split}\ t : \&_p A \otimes \&_q A}\ \textsc{with}\&$$

Then by induction on the premise we have: $\Gamma_1 + \Gamma \vdash [t_1/x]t : B$ from which we construct the goal:

$$\frac{\Gamma_1 + \Gamma \vdash [t_1/x]t : \&_{p+q} A}{\Gamma_1 + \Gamma \vdash \mathbf{split}\ ([t_1/x]t) : \&_p A \otimes \&_q A}\ \textsc{with}\&$$

- (join)

$$\frac{\Gamma_1 \vdash t_1' : \&_p A \qquad \Gamma_2 \vdash t_2' : \&_q A \qquad p+q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \text{WITH\&}$$

Then there are two possibilities depending on the location of the linear typing variable:
(1) (on the left):

$$\frac{\Gamma_1', x : A \vdash t_1' : \&_p A \qquad \Gamma_2' \vdash t_2' : \&_q A \qquad p+q \leq 1}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \text{WITH\&}$$

Then by induction on the premise we have: $\Gamma_1' + \Gamma_1 \vdash [t_1/x]t_1' : \&_p A$ from which we construct the goal:

$$\frac{\Gamma_1', x : A \vdash [t/x]t_1' : \&_p A \qquad \Gamma_2' \vdash t_2' : \&_q A \qquad p+q \leq 1}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{join}\ ([t_1/x]t_1')\ t_2' : \&_{p+q} A}\ \text{WITH\&}$$

(2) (on the right):

$$\frac{\Gamma_1' \vdash t_1' : \&_p A \qquad \Gamma_2', x : A \vdash t_2' : \&_q A \qquad p+q \leq 1}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \text{WITH\&}$$

Then by induction on the premise we have: $\Gamma_2' + \Gamma_1 \vdash [t_1/x]t_2' : \&_q A$ from which we construct the goal:

$$\frac{\Gamma_1' \vdash t_1' : \&_p A \qquad \Gamma_2' + \Gamma_1 \vdash [t_1/x]t_2' : \&_q A \qquad p+q \leq 1}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{join}\ t_1'\ ([t_1/x]t_2') : \&_{p+q} A}\ \text{WITH\&}$$

- (push)

$$\frac{\Gamma, x : A \vdash t : \&_p (A \otimes B)}{\Gamma, x : A \vdash \mathbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \text{PUSH}$$

Then by induction on the premise we have: $\Gamma + \Gamma_1 \vdash [t_1/x]t : \&_p (A \otimes B)$ from which we construct the goal:

$$\frac{\Gamma + \Gamma_1 \vdash [t_1/x]t : \&_p (A \otimes B)}{\Gamma + \Gamma_1 \vdash \mathbf{push}\ [t_1/x]t : \&_p A \otimes \&_q A}\ \text{PUSH}$$

- (pull)

$$\frac{\Gamma, x : A \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma, x : A \vdash \mathbf{pull}\ t : \&_p (A \otimes B)}\ \text{PULL}$$

Then by induction on the premise we have: $\Gamma + \Gamma_1 \vdash [t_1/x]t : (\&_p A) \otimes (\&_p B)$ from which we construct the goal:

$$\frac{\Gamma + \Gamma_1 \vdash [t_1/x]t : (\&_p A) \otimes (\&_p B)}{\Gamma + \Gamma_1 \vdash \mathbf{pull}\ [t_1/x]t : \&_p (A \otimes B)}\ \text{PULL}$$

- (newArray), (readArray), (writeArray), (deleteArray) all trivial as they are atomic with substitution having no effect.

□

Lemma B.2 (Graded substitution is admissible, extending [Orchard et al. 2019]). *If* $[\Gamma_1] \vdash t_1 : A$ *and* $\Gamma_2, x : [A]_r \vdash t_2 : B$ *and* $\neg\mathsf{resourceAllocator}(t_1)$ *then* $\Gamma_2 + r \cdot \Gamma_1 \vdash [t_1/x]t_2 : B$.

Proof. By induction on the typing derivation of t_2.

- (pr)

$$\frac{\Gamma_2', x : [A]_{r_2} \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r_1 \cdot (\Gamma_2', x : [A]_{r_2}) \vdash [t] : \Box_{r_1} A}\ \textsc{pr}$$

  where $r = r_1 * r_2$ and $t_2 = [t]$.
  By induction on the premise then we have $\Gamma_2', r_2 \cdot \Gamma_1 \vdash [t_1/x]t : A$.
  Then we construct the goal:

$$\frac{\Gamma_2', r_2 \cdot \Gamma_1 \vdash [t_1/x]t : A \qquad \neg\mathsf{resourceAllocator}([t_1/x]t)}{r_1 \cdot (\Gamma_2', r_2 \cdot \Gamma_1) \vdash [[t_1/x]t] : \Box_{r_1} A}\ \textsc{pr}$$

  where $\neg\mathsf{resourceAllocator}([t_1/x]t)$ follows from $\neg\mathsf{resourceAllocator}(t)$ and $\neg\mathsf{resourceAllocator}(t_1)$ and which equals $r_1 \cdot \Gamma_2' + r_1 \cdot r_2 \cdot \Gamma_1 \vdash [[t_1/x]t] : \Box_{r_1} A$ satisfying the goal here.
- (share)

$$\frac{\Gamma_2, x : [A]_r \vdash t : {*}A}{\Gamma_2, x : [A]_r \vdash \mathbf{share}\ t : \Box_s A}\ \textsc{share}$$

  where $B = \Box_s A$.
  By induction on the premise then $\Gamma_2 + r \cdot \Gamma_1 \vdash [t_1/x]t : {*}A$, from which we build the conclusion:

$$\frac{\Gamma_2 + r \cdot \Gamma_1 \vdash [t_1/x]t : {*}A}{\Gamma_2 + r \cdot \Gamma_1 \vdash \mathbf{share}\ ([t_1/x]t) : \Box_s A}\ \textsc{share}$$

- (bind)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t_1' : \Box_s A' \qquad \Gamma_2', y : {*}(\#A'), x : [A]_{r_2} \vdash t_2' : \Box_s B \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_2', x : [A]_{r_1 + r_2} \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ y\ \mathbf{in}\ t_2' : \Box_s B}\ \textsc{clone}'$$

  with $r = r_1 + r_2$ without loss of generality (since any context not including x can instead have weakening applied to have either $r_1 = 0$ and/or $r_2 = 0$).
  By induction on the premises, we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t/x]t_1' : \Box_s A'$ (2) $(\Gamma_2' + r_2 \cdot \Gamma_1), y : {*}A' \vdash [t/x]t_2' : \Box_s B'$
  Then we reconstruct the typing as:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t/x]t_1' : \Box_s A' \qquad (\Gamma_2' + r_2 \cdot \Gamma_1), y : {*}(\#A') \vdash [t/x]t_2' : \Box_s B'}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{clone}\ [t/x]t_1'\ \mathbf{as}\ y\ \mathbf{in}\ [t/x]t_2' : \Box_s B'}\ \textsc{Tybind}$$

  satisfying the goal.
- (withBorrow)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t : {*}A' \qquad \Gamma_2', x : [A]_{r_2} \vdash f : \&_1 A' \multimap \&_1 B'}{(\Gamma_1' + \Gamma_2'), x : [A]_{r_1 + r_2} \vdash \mathbf{withBorrow}\ f\ t : {*}B'}\ \textsc{with\&}$$

  By induction on the premises, we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t_1/x]t : {*}A'$. (2) $\Gamma_2' + r_2 \cdot \Gamma_1 \vdash [t_1/x]f : \&_1 A' \multimap \&_1 B'$.
  From this we construct the goal:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t_1/x]t : {*}A' \qquad \Gamma_2' + r_2 \cdot \Gamma_1 \vdash [t_1/x]f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{withBorrow}\ [t_1/x]f\ t : {*}B'}\ \textsc{with\&}$$

  satisfying the goal.

- (split)

$$\frac{\Gamma, x : [A]_r \vdash t : \&_{p+q} A}{\Gamma, x : [A]_r \vdash \mathbf{split}\ t : \&_p A \otimes \&_q A}\ \text{WITH\&}$$

Then by induction on the premise we have: $\Gamma_1 + r \cdot \Gamma \vdash [t_1/x]t : B$ from which we construct the goal:

$$\frac{\Gamma_1 + r \cdot \Gamma \vdash [t_1/x]t : \&_{p+q} A}{\Gamma_1 + r \cdot \Gamma \vdash \mathbf{split}\ ([t_1/x]t) : \&_p A \otimes \&_q A}\ \text{WITH\&}$$

- (join)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t_1' : \&_p A \qquad \Gamma_2', x : [A]_{r_2} \vdash t_2' : \&_q A \qquad p + q \leq 1}{(\Gamma_1' + \Gamma_2'), x : [A]_{(r_1 + r_2)} \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \text{WITH\&}$$

Then by induction on the premisese we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t_1/x]t_1' : \&_p A$ (2) $\Gamma_2' + r_2 \cdot \Gamma_1 \vdash [t_1/x]t_2' : \&_q A$ from which we construct the goal:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash [t_1/x]t_1' : \&_p A \qquad \Gamma_2' + r_2 \cdot \Gamma_1 \vdash [t_1/x]t_2' : \&_q A \qquad p + q \leq 1}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{join}\ t_1'\ ([t_1/x]t_2') : \&_{p+q} A}\ \text{WITH\&}$$

satisfying the goal.

- (push)

$$\frac{\Gamma, x : [A]_r \vdash t : \&_p (A \otimes B)}{\Gamma, x : [A]_r \vdash \mathbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \text{PUSH}$$

Then by induction on the premise we have: $\Gamma + r \cdot \Gamma_1 \vdash [t_1/x]t : \&_p (A \otimes B)$ from which we construct the goal:

$$\frac{\Gamma + r \cdot \Gamma_1 \vdash [t_1/x]t : \&_p (A \otimes B)}{\Gamma + r \cdot \Gamma_1 \vdash \mathbf{push}\ [t_1/x]t : \&_p A \otimes \&_q A}\ \text{PUSH}$$

- (pull)

$$\frac{\Gamma, x : [A]_r \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma, x : [A]_r \vdash \mathbf{pull}\ t : \&_p (A \otimes B)}\ \text{PULL}$$

Then by induction on the premise we have: $\Gamma + r \cdot \Gamma_1 \vdash [t_1/x]t : (\&_p A) \otimes (\&_p B)$ from which we construct the goal:

$$\frac{\Gamma + r \cdot \Gamma_1 \vdash [t_1/x]t : (\&_p A) \otimes (\&_p B)}{\Gamma + r \cdot \Gamma_1 \vdash \mathbf{pull}\ [t_1/x]t : \&_p (A \otimes B)}\ \text{PULL}$$

- (newArray), (readArray), (writeArray), (deleteArray) all trivial as they are atomic with substitution having no effect. a

□

C TYPE SAFETY

C.1 Progress proof

Lemma C.1. *Value lemma*
Given $\Gamma \vdash v : A$ *then, depending on the type, the shape of* v *can be inferred:*

- $A = A' \to B$ *then* $v = \lambda x.v'$ *or a partially applied primitive term* p.
- $A = \Box_r A'$ *then* $v = [v']$.
- $A = A' \otimes B$ *then* $v = (v_1, v_2)$.
- $A = 1$ *then* $v = ()$.
- $A = *A'$ *then* $v = *v'$.
- $A = \&_p A'$ *then* $v = \&(*v')$.
- $A = \mathbb{N}$ *then* $v = n$.
- $A = \mathbb{F}$ *then* $v = f$.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ *then* $v = a$.
- $A = \exists id.A'$ *then* $v = \textbf{\textit{pack}}\ \langle id', v' \rangle$.

Proof. Recall that the value terms sub-grammar is:

$$v ::= (v_1, v_2) \mid () \mid *v \mid \&v \mid [v] \mid \lambda x.t \mid i \mid a \mid p \mid \textbf{pack}\ \langle id', v' \rangle \qquad \text{(value terms sub-grammar)}$$

where p are partially-applied primitives, e.g.,

$$\begin{aligned} p ::= {} & \textbf{newArray} \mid \textbf{readArray} \mid \textbf{readArray}\ (*a) \\ & \mid \textbf{writeArray} \mid \textbf{writeArray}\ (*a) \mid \textbf{writeArray}\ (*a)\ i \mid \textbf{deleteArray} \end{aligned}$$

We then proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exists additional derivations based on dereliction and approximation, e.g., for the case where $A = A' \to B$:

$$\frac{\Gamma, x : A'' \vdash t : A' \to B}{\Gamma, x : [A'']_1 \vdash t : A' \to B}\ \text{DER}$$

$$\frac{\Gamma, y : [A'']_r, \Gamma' \vdash t : A' \to B \quad r \sqsubseteq s}{\Gamma, y : [A'']_s, \Gamma' \vdash t : A' \to B}\ \text{APPROX}$$

In all of these cases we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion.

We elide handling this separately each time in the cases that follow as the reasoning through dereliction is the same each time.

- $A = A' \to B$ then there are two classes of possible typing:
  - Abstract term:
  $$\frac{\Gamma, x : A' \vdash v' : B}{\Gamma \vdash \lambda x.v' : A' \to B}\ \text{ABS}$$
  thus $v = \lambda x.v'$ as in the lemma statement.
  - Primitive term p formed by an application of zero or more values to a primitive operation, of which there are then seven possibilities:

  (1)
  $$\frac{}{0 \cdot \Gamma \vdash \textbf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{NEW}$$
  thus $v = \textbf{newArray}$

(2)

$$\frac{}{0 \cdot \Gamma \vdash \textbf{readArray} : \&_p(\mathsf{Array}_{id}\,\mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{read}$$

thus $v = \textbf{readArray}$

(3)

$$\frac{\dfrac{\dfrac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\,A \vdash *a : *(\mathsf{Array}_{id}\,A)}\ *\textsc{arr}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \&a : \&_1\mathsf{Array}_{id}\,\mathbb{F}}\ \textsc{borrow}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \textbf{readArray}\,(\&a) : \mathbb{N} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{app}$$

thus $v = \textbf{readArray}\,(\&a)$

(4)

$$\frac{p \equiv 1 \ \vee \ p \equiv *}{0 \cdot \Gamma \vdash \textbf{writeArray} : \&_p(\mathsf{Array}_{id}\,\mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{write}$$

thus $v = \textbf{writeArray}$

(5)

$$\frac{\dfrac{\dfrac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\,A \vdash *a : *(\mathsf{Array}_{id}\,A)}\ *\textsc{arr}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \&a : \&_1\mathsf{Array}_{id}\,\mathbb{F}}\ \textsc{borrow}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \textbf{writeArray}\,\&a : \mathbb{N} \multimap \mathbb{F} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{app}$$

thus $v = \textbf{writeArray}\,\&a$

(6)

$$\frac{\dfrac{\dfrac{\dfrac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\,A \vdash *a : *(\mathsf{Array}_{id}\,A)}\ *\textsc{arr}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \&a : \&_1\mathsf{Array}_{id}\,\mathbb{F}}\ \textsc{borrow}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \textbf{writeArray}\,(\&a) : \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{app} \qquad \emptyset \vdash i : \mathbb{N}}{[\Gamma], a : \mathsf{Array}_{id}\,A \vdash \textbf{writeArray}\,(\&a)\,i : \mathbb{F} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\,\mathbb{F})}\ \textsc{app}$$

thus $v = \textbf{writeArray}\,(\&a)\,i$

(7)

$$\frac{}{0 \cdot \Gamma \vdash \textbf{deleteArray} : \forall\, id.*(\mathsf{Array}_{id}\,\mathbb{F}) \multimap \mathsf{unit}}\ \textsc{del}$$

thus $v = \textbf{deleteArray}$

- $A = \Box_r A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma \vdash v' : A'}{r \cdot \Gamma \vdash [v'] : \Box_r A'}\ \textsc{pr}$$

thus $v = [v']$ as in the lemma statement.

- $A = A' \otimes B$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma_1 \vdash v_1 : A' \quad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A' \otimes B}\ \otimes_I$$

thus $v = (v_1, v_2)$ as in the lemma statement.

- $A = 1$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

thus $v = ()$ as in the lemma statement.

- $A = *A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\emptyset \vdash t : A'}{[\Gamma] \vdash *t : *A'}\ \text{NEC}$$

  thus $v = *t$ as in the lemma statement.
- $A = \&_p A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\dfrac{\emptyset \vdash t : A'}{[\Gamma] \vdash *t : *A'}\ \text{NEC}}{[\Gamma] \vdash \&(*t) : \&_1 A'}\ \text{BORROW}$$

  thus $v = \&(*t)$ as in the lemma statement.
- $A = \mathbb{N}$ then $v = n$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).
- $A = \mathbb{F}$ then $v = f$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ then $v = a$ then the only possible typing that is a value is given by:

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash a : \mathsf{Array}_{id}\ A}\ \text{ARR}$$

  (and since the only type of arrays is $\mathbb{F}$ currently).
- $A = \exists id.A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma \vdash v' : A \quad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \text{PACK}$$

  thus $v = \mathbf{pack}\ \langle id', v'\rangle$ as in the lemma statement.

□

Lemma C.2 (Unique value lemma). *Given $\Gamma \vdash *v : *A$ then, depending on the type A, the shape of v can be inferred:*

- $A = A' \otimes B'$ *then* $v = (v_1, v_2)$.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ *then* $v = a$.

and there are no other possible typings for $*v$. Furtheremore, $\exists \Gamma', \gamma$ such that $0 \cdot \Gamma', \gamma \vdash *v : *A$, i.e., it can be type in a runtime context only.

Proof. There are only two possible typings for $*v$.

- $A = A' \otimes B'$ where there is only one possible non-dereliction typing of a value at the type $*(A' \otimes B')$:

$$\frac{\dfrac{\dfrac{}{0 \cdot \Gamma_1, \gamma_1 \vdash *v_1 : *A'}\ \text{INDUCTION.} \quad \dfrac{}{0 \cdot \Gamma_2, \gamma_2 \vdash *v_2 : *B'}\ \text{INDUCTION.}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma_1 + \gamma_2 \vdash (*v_1, *v_2) : *A' \otimes *B'}\ \otimes_I}{0 \cdot (\Gamma_1 + \Gamma_2) + \gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B')}\ \text{PULL}$$

  thus $v = (v_1, v_2)$ as in the lemma statement and $\Gamma' = \Gamma_1 + \Gamma_2$ and $\gamma = \gamma_1 + \gamma_2$.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ where there is only one possible non-dereliction typing of a value at the type $*(\mathsf{Array}_{id}\ \mathbb{F})$:

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : *(\mathsf{Array}_{id}\ A)}\ *\text{ARR}$$

thus $v = a$ as in the lemma statement (and since the only type of arrays is $\mathbb{F}$ currently) and $\Gamma' = \Gamma$ and $\gamma = a : \mathsf{Array}_{id}\ A$.

□

Theorem C.3 (Progress). *Given $\Gamma \vdash t : A$, then t is either a value, or if $H \bowtie \Gamma_0 + \Gamma$ there exists a heap H', term t', grade s, such that $H \vdash t \rightsquigarrow_s H' \vdash t'$.*

Proof. By induction on typing.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \text{VAR}$$

Here, $H \bowtie [\Gamma], x : A$, which by inversion of heap compatibility implies that $H = H', x{\mapsto_r} v : A$. Hence, we can reduce by the following rule:

$$\frac{\exists r'.\ r' + s \sqsubseteq r}{H, x{\mapsto_r} v : A \vdash x \rightsquigarrow_s H, x{\mapsto_r} v : A \vdash v}\ \rightsquigarrow_{\text{VAR}}$$

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{ABS}$$

A value.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \text{APP}$$

By induction on the first premise, there are two possibilities.

(1) t_1 is a value and therefore by the value lemma there are a number of choices:
   - $t_1 = \lambda x.t_1'$. Therefore we induct on the second argument:
     * If $t_2 = v$ for some value v, then we can reduce by the following rule:

$$\frac{\Gamma \vdash v : A}{H \vdash (\lambda x.t)\ v \rightsquigarrow_s H, x{\mapsto_s} v : A \vdash t}\ \rightsquigarrow_{\beta}$$

     * If t_2 is not a value, then by induction on the second premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\ t_2 \rightsquigarrow_s H' \vdash v\ t_2'}\ \rightsquigarrow_{\text{APPR}}$$

   - $t_1 =$ **newArray** therefore $A = \mathbb{N}$
     Therefore we induct on the second argument:
     * t_2 is a value and therefore by the value lemma (Lemma C.2) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \textbf{newArray}\ n : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{TyDerivednewArray}$$

     with $H \bowtie (\Gamma_0 + \Gamma)$.

Thus there is a reduction as follows:

$$\frac{a\#H \qquad id\#H}{H \vdash \mathbf{newArray}\ n \rightsquigarrow_s H, a \mapsto_1 id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\ \langle id, *a\rangle}\ \rightsquigarrow_{\text{NEWARRAY}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{newArray}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{newArray}\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readArray}$ therefore $A = \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $\&_p(\mathsf{Array}_{id}\ \mathbb{F})$ (Lemma C.2) $t_2 = (\&a)$ and thus $t_1\ t_2 = \mathbf{readArray}\ (\&a)$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readArray}\ (\&(*a))$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma C.2) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{ARR}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (\&(*a)) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \&\text{ARR} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\ (\&(*a))\ i : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{TyDerivedreadArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (\&(*a))\ i \rightsquigarrow_s H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash (v, \&(*a))}\ \rightsquigarrow_{\text{READ\&ARRA}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\ (\&(*a))\ t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\ (\&(*a))\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readArray}\ (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma C.2) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{ARR}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : *(\mathsf{Array}_{id}\ \mathbb{F})}\ *\text{ARR} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\ (*a)\ i : \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{TyDerivedreadArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (*a)\ i \rightsquigarrow_s H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash (v, *a)}\ \rightsquigarrow_{\text{READARRAY}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\ (*a)\ t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\ (*a)\ t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{writeArray}$ therefore $A = \&_1(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_1(\mathsf{Array}_{id}\ \mathbb{F})$
* t_2 is a value therefore by the value lemma (Lemma C.2) $t_2 = (\&a)$ and thus $t_1\ t_2 = \mathbf{writeArray}\ (\&a)$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{writeArray}\ (\&(*a))$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap \&_1(\mathsf{Array}_{id}\ \mathbb{F})$
* t_2 is a value therefore by the value lemma (Lemma C.2) $t_2 = n$ and thus $t_1\ t_2 = \mathbf{writeArray}\ (\&(*a))\ n$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{writeArray}\ (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}_{id}\ \mathbb{F})$
* t_2 is a value therefore by the value lemma (Lemma C.2) $t_2 = n$ and thus $t_1\ t_2 = \mathbf{writeArray}\ (*a)\ n$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (*a)\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (*a)\ t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{writeArray}\ (\&(*a))\ i$ therefore $A = \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})$
* t_2 is a value and therefore the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma C.2) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{ARR}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_1(\mathsf{Array}_{id}\ \mathbb{F})}\ \&\text{ARR} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ f : \&_1(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{TyDerivedwriteArray}$$

with $H' \bowtie ([\Gamma_0] + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\frac{}{H, a \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ v \rightsquigarrow_s H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash \&(*a)} \rightsquigarrow_{\text{write\&Array}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{writeArray}\ (*a)\ i$ therefore $A = \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma C.2) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{arr}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : *(\mathsf{Array}_{id}\ \mathbb{F})}\ {*}\textsc{arr} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\ (*a)\ i\ f : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{TyDerivedwriteArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a{\mapsto_p} id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\dfrac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (*a)\ i\ v \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash *a}\ \rightsquigarrow_{\textsc{writeArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

– $t_1 = \mathbf{deleteArray}$ therefore $A = *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}$

* t_2 is a value and therefore by the value lemma (Lemma C.2) $t_2 = *a$ thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{arr}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ \mathbb{F})}\ {*}\textsc{arr}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{deleteArray}\ *\ a : \mathsf{unit}}\ \textsc{TyDerivedDeleteArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a{\mapsto_p} id, id \mapsto \mathbf{arr}$.
There there is a reduction as follows:

$$\dfrac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\ (*a) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\textsc{deleteArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{deleteArray}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{deleteArray}\ t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

(2) Otherwise, by induction on the premise we then have that there exists heap H', term t_1', and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\dfrac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\textsc{appL}}$$

• (pr)

$$\dfrac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

By induction on the premise, there are two possibilities.
(1) If t is a value, then $[t]$ is also a value.

(2) Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_{s*r} H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t' \qquad \Gamma \vdash [t] : \Box_r A}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']} \rightsquigarrow_{\Box}$$

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \text{ELIM}$$

By induction on the first premise, there are two possibilities.

(1) t_1 is a value and therefore by the value lemma $t_1 = [v]$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v] : \Box_r A}\ \text{PR} \qquad \Gamma_2, x : \Box_r A \vdash t_2 : B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = [v]\,\mathbf{in}\,t_2 : B}\ \text{ELIM}$$

Therefore we can reduce by the following rule:

$$\frac{[\Gamma_1] \vdash v : A}{H \vdash \mathbf{let}\,[x] = [v]\,\mathbf{in}\,t \rightsquigarrow_s H, x \mapsto_{s*r} v : A \vdash t} \rightsquigarrow_{\Box\beta}$$

(2) Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\text{LET}\Box}$$

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{DER}$$

Goal achieved immediately by induction on the premise.

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \text{APPROX}$$

Goal achieved immediately by induction on the premise.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B} \otimes_I$$

By induction on the premises there are three possible cases.

(1) If both t_1 and t_2 are values then (t_1, t_2) is also a value.

(2) If only t_1 is a value then by induction on the second premise we then have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')} \rightsquigarrow_{\otimes \text{R}}$$

(3) If neither t_1 nor t_2 are values then by induction on the first premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)} \rightsquigarrow_{\otimes \mathrm{L}}$$

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

By induction on the first premise, there are two possibilities.

(1) t_1 is a value and therefore by the value lemma $t_1 = (v_1, v_2)$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A \otimes B}\ \otimes_I \qquad \Gamma_3, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

Therefore, we can reduce by the following rule:

$$\frac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x \overset{s}{\mapsto} v_1 : A, y \overset{s}{\mapsto} v_2 : B \vdash t} \rightsquigarrow_{\otimes\beta}$$

(2) Otherwise, by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\mathrm{LET}\otimes}$$

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

A value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

By induction on the first premise, there are two possibilities.

(1) t is a value and therefore by the value lemma $t = ()$. Therefore we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \rightsquigarrow_s H \vdash t} \rightsquigarrow_{\beta\mathrm{UNIT}}$$

(2) Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\mathrm{LETUNIT}}$$

- (returnGen)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\,t : \Box_r A}\ \mathrm{SHARE}$$

By induction on the premise, there are two possibilities.

(1) t is a value and therefore by the value lemma $t = *v$. Therefore we can reduce by the following rule:

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]} \rightsquigarrow_{\text{SHARE}\beta}$$

(2) Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\ t \rightsquigarrow_s H' \vdash \mathbf{share}\ t'} \rightsquigarrow_{\text{SHARE}}$$

- (bindGen)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \mathbf{clone'}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \text{CLONE'}$$

By induction on the first premise, there are two possibilities.

(1) t_1 is a value and therefore by the value lemma $t_1 = [v]$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v] : \Box_r A}\ \text{PR} \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \text{CLONE'}$$

Therefore we can reduce using the following rule:

$$\frac{\Gamma \vdash [v] : \Box_r A \qquad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H')}{H, H' \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H, H', H'', x \mapsto_s \mathbf{pack}\ \langle \overline{id}, *(\theta(v)) \rangle : *A \vdash t_2} \rightsquigarrow_{\text{CLONE}\beta}$$

(2) Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ x\ \mathbf{in}\ t_2} \rightsquigarrow_{\text{CLONE}}$$

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ t_1\ t_2 : *B}\ \text{WITH\&}$$

By induction on the first premise, there are two possibilities.

(1) t_1 is a value and therefore by the value lemma $t_1 = (\lambda x.t)$. Then, again, we have two possibilities:

- t_2 is also a value, and therefore by the value lemma $t_2 = (*v)$ for some v. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H \vdash \mathbf{unborrow}\ ([\&(*v)/x]t)} \rightsquigarrow_{\text{WITH\&}}$$

- t_2 is not a value. Then, by induction on the premise we have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\text{WITH\&R}}$$

(2) t_1 is not a value. Then, by induction on the premise we have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\text{WITH\&L}}$$

- (split)

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \textbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \text{SPLIT}$$

By induction on the premise, there are two cases.

(1) If t is a value, then by the value lemma and the unique value lemma there are again two possibilities for the form of t.

– t could have the form $\&(*a)$. This refines the typing as follows:

$$\frac{[\Gamma], a : \text{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\text{Array}_{id}\ \mathbb{F})\ \text{\&ARR}}{[\Gamma], a : \text{Array}_{id}\ \mathbb{F} \vdash \textbf{split}\ (\&(*a)) : \&_{\frac{p}{2}}(\text{Array}_{id}\ \mathbb{F}) \otimes \&_{\frac{p}{2}}(\text{Array}_{id}\ \mathbb{F})}\ \text{SPLIT}$$

Then we can reduce by the following rule:

$$\frac{\#a_1 \qquad \#a_2}{H, id \mapsto \textbf{arr}, a \mapsto_p id \vdash \textbf{split}\ (\&(*a)) \rightsquigarrow_s H, id \mapsto \textbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id \vdash (\&(*a_1), \&(*a_2))}\ \rightsquigarrow_{\text{SPLITARR}}$$

– t could have the form $\&(*(v, w))$. There are two possible typings in this case:

$$\dfrac{\dfrac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash v_2 : B}{\gamma_1 + \gamma_2 \vdash (v_1, v_2) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \text{NEC}}{\gamma_1 + \gamma_2 \vdash \&(*(v_1, v_2)) : \&_p(A' \otimes B)}\ \text{BORROW}}{\gamma_1 + \gamma_2 \vdash \textbf{split}\ (\&(*(v_1, v_2))) : (\&_{\frac{1}{2}}(A' \otimes B) \otimes \&_{\frac{1}{2}}(A' \otimes B))}\ \text{SPLIT}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A'}{\gamma_1 \vdash *v_1 : *A'}\ \text{NEC} \qquad \dfrac{\gamma_2 \vdash v_2 : B}{\gamma_2 \vdash *v_2 : *B}\ \text{NEC}}{\gamma_1 + \gamma_2 \vdash (*v_1, *v_2) : *A' \otimes *B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \text{PULL}}{\gamma_1 + \gamma_2 \vdash \&(*(v_1, v_2)) : \&_p(A' \otimes B)}\ \text{BORROW}}{\gamma_1 + \gamma_2 \vdash \textbf{split}\ (\&(*(v_1, v_2))) : (\&_{\frac{p}{2}}(A' \otimes B) \otimes \&_{\frac{p}{2}}(A' \otimes B))}\ \text{SPLIT}$$

In either instance, we can reduce by the following rule:

$$\frac{\begin{array}{c} H \vdash \mathbf{split}\ (\&(*v)) \rightsquigarrow_s H' \vdash (\&(*v_1), \&(*v_2)) \\ H' \vdash \mathbf{split}\ (\&(*w)) \rightsquigarrow_s H'' \vdash (\&(*w_1), \&(*w_2)) \end{array}}{H \vdash \mathbf{split}\ (\&(*(v,w))) \rightsquigarrow_s H'' \vdash (\&(*(v_1,w_1)), \&(*(v_2,w_2)))} \rightsquigarrow_{\text{SPLIT}\otimes}$$

(2) If t is not a value, then by induction on the premise we have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{split}\ t \rightsquigarrow_s H' \vdash \mathbf{split}\ t'} \rightsquigarrow_{\text{SPLIT}}$$

- (join)

$$\frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p+q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \text{JOIN}$$

By induction on the first premise, there are two cases.

(1) If t_1 is a value, then first we should consider whether t_2 is also a value.

– If t_2 is also a value, then by the value lemma and the unique value lemma there are two possibilities for the form of t_1.

(a) t_1 could have the form $\&(*a_1)$. This refines the typing as follows:

$$\frac{[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a_1) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})\ \&\text{ARR} \qquad [\Gamma], a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a_2) : \&_q(\mathsf{Array}_{id}\ \mathbb{F})\ \&\text{ARR}}{[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F}, a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) : \&_{p+q}(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{JOIN}$$

Note that the typing in this case restricts t_2 to also have the form $\&(*a_2)$. Then we can reduce by the following rule:

$$\frac{\#a}{H, id \mapsto \mathbf{arr}, a_1 \mapsto_p id, a_2 \mapsto_q id \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) \rightsquigarrow_s H, id \mapsto \mathbf{arr}, a \mapsto_{(p+q)} id \vdash \&(*a)} \rightsquigarrow_{\text{JOINARR}}$$

(b) t_1 could have the form $\&(*(v_1, w_1))$. There are two possible typings in this case:

$$\frac{\dfrac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash w_1 : B}{\gamma_1 + \gamma_2 \vdash (v_1, w_1) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, w_1) : *(A' \otimes B)}\ \text{NEC}}{\gamma_1 + \gamma_2 \vdash \&(*(v_1, w_1)) : \&_p(A' \otimes B)}\ \text{BORROW} \qquad \frac{\dfrac{\dfrac{\gamma_3 \vdash v_2 : A' \quad \gamma_4 \vdash w_2 : B}{\gamma_3 + \gamma_4 \vdash (v_2, w_2) : A' \otimes B}\otimes_I}{\gamma_3 + \gamma_4 \vdash *(v_2, w_2) : *(A' \otimes B)}\ \text{NEC}}{\gamma_3 + \gamma_4 \vdash \&(*(v_2, w_2)) : \&_p(A' \otimes B)}\ \text{BORROW}}{\gamma_1 + \gamma_2 + \gamma_3 + \gamma_4 \vdash \mathbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) : \&_{p+q}(A' \otimes B)}\ \text{JOIN}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A'}{\gamma_1 \vdash *v_1 : *A'}\ \text{NEC} \qquad \dfrac{\gamma_1 \vdash w_1 : B}{\gamma_1 \vdash *w_1 : *B}\ \text{NEC}}{\gamma_1 + \gamma_2 \vdash (*v_1, *w_1) : *A' \otimes *B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, w_1) : *(A' \otimes B)}\ \text{PULL}}{\gamma_1 + \gamma_2 \vdash \&(*(v_1, w_1)) : \&_p(A' \otimes B)}\ \text{BORROW} \qquad \frac{\dfrac{\dfrac{\dfrac{\gamma_3 \vdash v_2 : A'}{\gamma_3 \vdash *v_2 : *A'}\ \text{NEC} \qquad \dfrac{\gamma_4 \vdash w_2 : B}{\gamma_4 \vdash *w_2 : *B}\ \text{NEC}}{\gamma_3 + \gamma_4 \vdash (*v_2, *w_2) : *A' \otimes *B}\otimes_I}{\gamma_3 + \gamma_4 \vdash *(v_2, w_2) : *(A' \otimes B)}\ \text{PU}}{\gamma_3 + \gamma_4 \vdash \&(*(v_2, w_2)) : \&_q(A' \otimes B)}}{\gamma_1 + \gamma_2 + \gamma_3 + \gamma_4 \vdash \mathbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) : \&_{p+q}(A' \otimes B)}$$

Note that the typing in both situations restricts t_2 to also have the form $\&(*(v_2, w_2))$. In either instance we can reduce by the following rule:

$$\frac{\begin{array}{l} H \vdash \mathbf{join}\ (\&(*v_1))\ (\&(*v_2)) \rightsquigarrow_s H' \vdash (\&(*v)) \\ H' \vdash \mathbf{join}\ (\&(*w_1))\ (\&(*w_2)) \rightsquigarrow_s H'' \vdash (\&(*w)) \end{array}}{H \vdash \mathbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) \rightsquigarrow_s H'' \vdash \&(*(v, w))} \rightsquigarrow_{\text{JOIN}\otimes}$$

– If t_2 is not a value, then by induction on the second premise we have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'} \rightsquigarrow_{\text{JOINR}}$$

(2) If t_1 is not a value, then by induction on the second premise we have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'} \rightsquigarrow_{\text{JOINR}}$$

- (nec)

$$\frac{\gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{0 \cdot \Gamma, \gamma \vdash *t : *A}\ \text{NEC}$$

By induction on the premise, there are two possibilities.

(1) If t is a value, then $*t$ is also a value.

(2) Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash *t \rightsquigarrow_s H' \vdash *t'} \rightsquigarrow_*$$

- (push)

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \text{PUSH}$$

By induction on the premise, there are two possibilities.

(1) If t is a value, then by the value lemma and the unique value lemma there are two possibilities for the form of t.

– t could have the form $*(v_1, v_2)$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{push}\ *\ (v_1, v_2) \rightsquigarrow_s H \vdash (*v_1, *v_2)} \rightsquigarrow_{\text{PUSH}*}$$

– t could have the form $\&(*(v_1, v_2))$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{push}\ \&(*(v_1, v_2)) \rightsquigarrow_s H \vdash (\&(*v_1), \&(*v_2))} \rightsquigarrow_{\text{PUSH}\&}$$

(2) If t is not a value, by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{push}\, t \rightsquigarrow_s H' \vdash \mathbf{push}\, t'} \rightsquigarrow_{\text{PUSH}}$$

- (pull)

$$\frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p (A \otimes B)} \text{ PULL}$$

By induction on the premise, there are two possibilities.

(1) If t is a value, then by the value lemma and the unique value lemma there are two possibilities for the form of t.
   - t could have the form $(*v_1, *v_2)$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{pull}\, (*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)} \rightsquigarrow_{\text{PULL}*}$$

   - t could have the form $(\&(*v_1), \&(*v_2))$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{pull}\, (\&(*v_1), \&(*v_2)) \rightsquigarrow_s H \vdash \&(*(v_1, v_2))} \rightsquigarrow_{\text{PULL}\&}$$

(2) If t is not a value, by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{pull}\, t \rightsquigarrow_s H' \vdash \mathbf{pull}\, t'} \rightsquigarrow_{\text{PULL}}$$

- (array)

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash a : \mathsf{Array}_{id}\, A} \text{ ARR}$$

A value.

- (uniqueArray)

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash *a : *(\mathsf{Array}_{id}\, A)} *\text{ARR}$$

A value.

- (borrowedArray)

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\, A)} \&\text{ARR}$$

A value.

- (borrow)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : \&_p A} \text{ BORROW}$$

By induction on the premise, there are two possibilities.

(1) If t is a value, then $\&t$ is also a value.

(2) Otherwise by induction on the premise we then have that there exists heap H', term t'_1 and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \&t \rightsquigarrow_s H' \vdash \&t'} \rightsquigarrow_{\&}$$

- (unborrow)

$$\frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \textbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

By induction on the premise, there are two possibilities.

(1) t is a value, and therefore by the value lemma $t = \&(*v)$ for some value v. Then we can reduce by the following rule:

$$\frac{}{H \vdash \textbf{unborrow}\ (\&(*v)) \leadsto_s H \vdash *v}\ \leadsto_{\textsc{un}\&}$$

(2) t is not a value. By induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \leadsto_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \leadsto_s H \vdash t'}{H \vdash \textbf{unborrow}\ t \leadsto_s H \vdash \textbf{unborrow}\ t'}\ \leadsto_{\textsc{unborrow}}$$

- (pack)

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \textbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \textsc{pack}$$

By induction on the premise, there are two possibilities.

(1) If t is a value, then $\textbf{pack}\ \langle id', t\rangle$ is also a value.

(2) Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \leadsto_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \leadsto_s H \vdash t'}{H \vdash \textbf{pack}\ \langle id, t\rangle \leadsto_s H \vdash \textbf{pack}\ \langle id, t'\rangle}\ \leadsto_{\textsc{pack}}$$

- (unpack)

$$\frac{\begin{array}{l}\Gamma_1 \vdash t_1 : \exists id.A \\ \quad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)\end{array}}{\Gamma_1 + \Gamma_2 \vdash \textbf{unpack}\ \langle id, x\rangle = t_1\ \textbf{in}\ t_2 : B}\ \textsc{unpack}$$

By induction on the premise, there are two possibilities.

(1) If t_1 is a value, then by the value lemma it has the form $\textbf{pack}\ \langle id', v\rangle$ for some value v. Then we can reduce by the following rule:

$$\frac{}{H \vdash \textbf{unpack}\ \langle id, x\rangle = \textbf{pack}\ \langle id', v\rangle\ \textbf{in}\ t \leadsto_s H[id'/id], x \mapsto_r v \vdash t}\ \leadsto_{\exists\beta}$$

(2) Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \leadsto_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \leadsto_s H \vdash t_1'}{H \vdash \textbf{unpack}\ \langle id, x\rangle = t_1\ \textbf{in}\ t_2 \leadsto_s H \vdash \textbf{unpack}\ \langle id, x\rangle = t_1'\ \textbf{in}\ t_2}\ \leadsto_{\textsc{unpack}}$$

□

C.2 Type preservation proof

Lemma C.4 (Admissibility of weakening).

$$\Gamma \vdash t : A \implies (0 \cdot \Gamma'), \Gamma \vdash t : A$$

Proof. By induction on the structure of typing and since all 'leaf' nodes of a typing derivation permit weakening (e.g., var rule, primitive rules). □

Lemma C.5 (Renaming array refs). *Given an array reference renaming θ (generated from a clone) then* $\Gamma \vdash t : A \implies \theta(\Gamma) \vdash \theta(t) : A$.

Proof. By trivial induction, with the only action happening in the use of the arr runtime typing rule for array references, in which case this acts just like alpha renaming via θ. □

Theorem C.6 (Type Preservation). *For a well-typed term $\Gamma \vdash t : A$ and all s, Γ_0, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ and a reduction $H \vdash t \rightsquigarrow_s H' \vdash t'$ we have:*

$$\exists \Gamma', H'.\ \ \Gamma' \vdash t' : A \ \ \wedge \ \ H' \bowtie (\Gamma_0 + s \cdot \Gamma')$$

Proof. By induction on the structure of typing and reductions.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

and one possible reduction:

$$\frac{\exists r'.\ r' + s \sqsubseteq r}{H, x \mapsto_r v : A \vdash x \rightsquigarrow_s H, x \mapsto_r v : A \vdash v}\ \rightsquigarrow_{\textsc{var}}$$

with incoming heap compatibility:

$$(H, x \mapsto_r v : A) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma, x : A))$$

with derivation:

$$\frac{H \bowtie (\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma') \qquad x \notin \mathrm{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r''.\ s + r'' \equiv r}{(H, x \mapsto_r v : A) \bowtie (\Gamma_0 + s * 0 \cdot \Gamma, x : [A]_s)}\ \textsc{extLin}$$

Goal 1:

$$\exists \Gamma''.\ \Gamma'' \vdash v : A$$

Which is provided by the third premise of the head compatibility derivation here, but with weakening (Lemma C.4) such that we have:

$$\frac{\Gamma' \vdash v : A}{0 \cdot \Gamma, \Gamma', x : [A]_0 \vdash v : A}\ \textsc{Lemma C.4}$$

Goal 2

$$(H, x \mapsto_r v : A) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma, \Gamma', x : [A]_0))$$

given by the following derivation from the premise of the incoming heap compatibility:

$$\frac{\dfrac{H \bowtie (\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma')}{H \bowtie ((\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma') + 0 \cdot \Gamma')}\,0\text{ unitality} \qquad x \notin \mathrm{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'''.\ 0 + r''' \equiv r}{H, x \mapsto_r v : A \bowtie ((\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma'), x : [A]_0)}\ \textsc{ext}$$

where the premise $\exists r''''.\ 0 + r''' \equiv r$ is fulfilled by $r''' = r$ by unitality and since $0 * s = 0$.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

Has no reduction so the case is trivial here.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.
There are four possible general reductions (and then further reductions for the operation of primitives, separeted our below).

General reductions (app).

(1) (beta)

$$\frac{\Gamma \vdash v : A}{H \vdash (\lambda x.t)\ v \rightsquigarrow_s H, x{\mapsto_s}v : A \vdash t}\ \rightsquigarrow_\beta$$

i.e. $t_1 = (\lambda x.t)\ v$ with the refined typing (where $\Gamma = \Gamma_2$):

$$\frac{\dfrac{\Gamma_1, x : A \vdash t : B}{\Gamma_1 \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \Gamma_2 \vdash v : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t)\ v : B}\ \textsc{app}$$

Therefore the resulting typing judgment is (goal) $\Gamma_1, x : A \vdash t : B$ given by the first premise here.
The goal heap compatibility is: (goal 2) $(H, x{\mapsto_1}v : A) \bowtie (\Gamma_0 + s \cdot (\Gamma_1, x : A))$
We construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{H \bowtie (\Gamma_0 + s \cdot \Gamma_1 + s \cdot \Gamma_2) \quad x \notin \mathrm{dom}(H) \quad \Gamma_2 \vdash v : A \quad \exists r'.\ s + r' \equiv 1}{H, x{\mapsto_1}v : A \bowtie (\Gamma_0 + s \cdot \Gamma_1), x : [A]_s}\ \textsc{ext}$$

where $r' = 0$ here.

(2) (appL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\text{APPL}}$$

with incoming heap compatibility $H \bowtie (\Gamma_0' + \Gamma_1 + \Gamma_2)$.
Applying induction on the premise reduction with heap compatibility given by the incoming heap compatibility but with $\Gamma_0 = \Gamma_0' + \Gamma_2$ then yields:

(a) $\exists \Gamma_1'.\ \Gamma_1' \vdash t_1' : A \multimap B$
(b) $H' \bowtie (\Gamma_0' + \Gamma_2 + \Gamma_1')$
(Goal 1) is then given by the reconstructed application type $\Gamma_1' + \Gamma_2 \vdash t_1'\ t_2 : B$
(Goal 2) is $H' \bowtie (\Gamma_0' + \Gamma_1' + \Gamma_2)$ which is given by the second conjunct above by commutativity of $+$.

(3) (appR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\ t_2 \rightsquigarrow_s H' \vdash v\ t_2'}\ \rightsquigarrow_{\text{APPR}}$$

Same as (appL) but by induction on the premise with t_2.

(4) (prim)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash pr\ t \rightsquigarrow_s H' \vdash pr\ t'} \rightsquigarrow_{\text{PRIM}}$$

Same as (appL) but by induction on the premise with t.

Primitives (app).

(1) (new)

$$\frac{a\#H \qquad id\#H}{H \vdash \mathbf{newArray}\ n \rightsquigarrow_s H, a{\mapsto_1} id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\ \langle id, *a\rangle} \rightsquigarrow_{\text{NEWARRAY}}$$

Thus typing refines to:

$$\frac{0 \cdot \Gamma_1 \vdash \mathbf{newArray} : \mathbb{N} \multimap \exists id.*(\mathsf{Array}_{id}\ \mathbb{F}) \quad 0 \cdot \Gamma_2 \vdash n : \mathbb{N}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 \vdash \mathbf{newArray}\ n : \exists id.*(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2)$ which is equal to $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2$ by distributivity of $*$ over $+$.

Goal 1 (typing) is thus given by:

$$\frac{\dfrac{}{0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ \mathbb{F})}\ {}^{*}\text{ARR}}{0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A \vdash \mathbf{pack}\ \langle id, *a\rangle : \exists id.\mathsf{Array}_{id}\ \mathbb{F}}\ \text{PACK}$$

i.e., we set $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A$ (which notably has runtime typing of a).

Goal 2 (compatibility) is $(H, a{\mapsto_1} id, id \mapsto \mathsf{init}) \bowtie (\Gamma_0 + (0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A))$

We construct this goal as follows (leveraging distributivity of $\cdot$ over $+$):

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2}{H, a{\mapsto_1} id, id \mapsto \mathsf{init} \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A)}\ \text{EXTREF}$$

(2) (read)

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (*a)\ i \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, *a)} \rightsquigarrow_{\text{READARRAY}}$$

Thus typing refines to

$$\frac{\begin{array}{c}0 \cdot \Gamma_1 \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ A) \qquad 0 \cdot \Gamma_3 \vdash i : \mathbb{N}\end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\ (*a)\ i : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}\times 2$$

i.e. where $p = *$ and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3}{(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTREF}$$

Goal 1 (typing) is thus given by:

$$\frac{\emptyset \vdash v : \mathbb{F} \qquad 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}{0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (v, *a) : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \otimes_I$$

i.e. $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}$.

Goal 2 (heap compatibility) is

$$(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F})$$

which is provided exactly by the incoming heap compatibility.

(3) (read')

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (\&(*a))\ i \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash (v, \&(*a))}\ \rightsquigarrow_{\text{READ\&ARRA}}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ A) \qquad 0 \cdot \Gamma_3 \vdash i : \mathbb{N} \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\ (\&(*a))\ i : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}\times 2$$

and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3}{(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTREF}$$

Goal 1 (typing) is thus given by:

$$\frac{\emptyset \vdash v : \mathbb{F} \qquad 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}{0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (v, \&(*a)) : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \otimes_I$$

i.e. $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}$.

Goal 2 (heap compatibility) is

$$(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F})$$

which is provided exactly by the incoming heap compatibility.

(4) (write)

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ v \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash \&(*a)}\ \rightsquigarrow_{\text{WRITE\&ARRAY}}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{writeArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ A) \qquad 0 \cdot \Gamma_3 \vdash i : \mathbb{N} \qquad 0 \cdot \Gamma_4 \vdash v : \mathbb{F} \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ v : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}\times 3$$

and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4}{(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTREF}$$

Goal 1 (typing) is thus given by:

$$\frac{0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ A)}{0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (\&(*a)) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{LEMMA C.4}$$

i.e. $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}$.

Goal 2 (heap compatibility) is

$$(H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F})$$

which is provided exactly by the incoming heap compatibility.

(5) (write')

$$\frac{}{H, a{\mapsto_p} id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (*a)\ i\ v \rightsquigarrow_s H, a{\mapsto_p} id, id \mapsto \mathbf{arr}[i] = v \vdash *a}\ \rightsquigarrow_{\text{WRITEARRAY}}$$

Similar to the above and the (read) case, mutatis mutandis.

(6) (delete)

$$\frac{}{H, a\mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\ (*a) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\text{DELETEARRAY}}$$

Thus typing refines to

$$\frac{\begin{array}{c}0 \cdot \Gamma_1 \vdash \mathbf{deleteArray} : \forall\, id.*(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit} \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ A)\end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{deleteArray}\ (*a) : \mathsf{unit}}\ \text{APP}$$

and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2}{(H, a\mapsto_p id, id \mapsto \mathbf{arr}) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTREF}$$

Goal 1 (typing) is thus given by:

$$\frac{}{0 \cdot (\Gamma_1 + \Gamma_2) \vdash () : \mathsf{unit}}\ 1_I$$

with $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2)$
Goal 2 (heap compatibility) is $H \bowtie (\Gamma_0 + (\Gamma_1 + \Gamma_2))$ given by the premise of incoming heap compatibility.

- (boxElim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

And two possible reductions:

(1) (betaBoxElim)

$$\frac{\Gamma \vdash [v] : \Box_r A}{H \vdash \mathbf{let}\ [x] = [v]\ \mathbf{in}\ t \rightsquigarrow_s H, x\mapsto_{(s*r)} v : A \vdash t}\ \rightsquigarrow_{\Box\beta}$$

refining the typing to:

$$\frac{\dfrac{[\Gamma_1'] \vdash v : A}{r \cdot \Gamma_1' \vdash [v] : \Box_r A}\ \text{PR} \qquad \Gamma_2, x : [A]_r \vdash t : B}{r \cdot \Gamma_1' + \Gamma_2 \vdash \mathbf{let}\ [x] = [v]\ \mathbf{in}\ t : B}\ \text{ELIM}$$

with incoming heap compatibility:

$$H \bowtie \Gamma_0 + r \cdot \Gamma_1' + \Gamma_2$$

Goal 1: typing Provided by the premise here as:

$$\Gamma_2, x : [A]_r \vdash t : A$$

i.e., we set the goal $\Gamma' = \Gamma_2, x : [A]_r$.
Goal 2: heap compatibility is $(H, x\mapsto_r v : A) \bowtie (\Gamma_0 + \Gamma_2, x : [A]_r)$ which refines to: $(H, x\mapsto_r v : A) \bowtie (\Gamma_0 + \Gamma_2), x : [A]_r$ by the disjointness of x
We construct this goal via the premise heap compatibility:

$$\frac{H \bowtie \Gamma_0 + r \cdot \Gamma_1' + \Gamma_2}{(H, x\mapsto_r v : A) \bowtie (\Gamma_0 + \Gamma_2), x : [A]_r}\ \text{EXT}$$

(2) (congBoxElim)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

with incoming heap compatibility:

$$H \bowtie \Gamma_0' + \Gamma_1 + \Gamma_2$$

Applying induction on the premise reduction with heap compatibility given by the incoming heap compatibility but with $\Gamma_0 = \Gamma_0' + \Gamma_2$ then yields:

(a) $\exists \Gamma_1'.\ \Gamma_1' \vdash t_1' : \Box_r A$

(b) $H' \bowtie (\Gamma_0' + \Gamma_2 + \Gamma_1')$

(Goal 1) is then given by the reconstructed application type $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2 : B$

(Goal 2) is $H' \bowtie (\Gamma_0' + \Gamma_1' + \Gamma_2)$ which is given by the second conjunct above by commutativity of $+$.

- (boxIntro)

$$\frac{\Gamma \vdash t : A \qquad \neg \text{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \text{PR}$$

and one possible reduction

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t' \qquad \Gamma \vdash [t] : \Box_r A}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']}\ \rightsquigarrow_{\Box}$$

with incoming heap compatibility:

$$H \bowtie \Gamma_0' + s' * r \cdot \Gamma$$

Applying induction on the premise reduction with $\Gamma_0 = \Gamma_0'$ and $s = s' * r$ then yields:

(1) $\exists \Gamma_1'.\ \Gamma_1' \vdash t_1' : \Box_r A$

(2) $H' \bowtie (\Gamma_0' + (s' * r) \cdot \Gamma_1')$

(Goal 1) is then given by the reconstructed application type $r \cdot \Gamma_1' \vdash [t_1'] : \Box_r A$

(Goal 2) is $H' \bowtie (\Gamma_0' + s' * r \cdot \Gamma_1')$ which is given by the second conjunct above by commutativity of $+$ and associativity of $*$.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{DER}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma, x : [A]_1)$ which refines to $H \bowtie \Gamma_0 + s \cdot \Gamma, x : [A]_s$ and a reduction:

$$H \vdash t \rightsquigarrow_s H' \vdash t'$$

Induction requires $H \bowtie \Gamma_0 + s \cdot (\Gamma, x : A)$ which by the definition of scalar multiplication is just $H \bowtie \Gamma_0 + s \cdot \Gamma, x : [A]_s$, thus we can apply induction and get the result of $\Gamma' \vdash t' : B$ and $H' \bowtie \Gamma_0 + s \cdot \Gamma_1'$ satisfying the goal here.

- (tensorIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

with incoming heap compatbitiliy $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.

And two possible reductions:

(1) (congPairL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)} \rightsquigarrow_{\otimes \mathrm{L}}$$

By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : A$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying $\otimes_I$ again to get $\Gamma_1' + \Gamma_2 \vdash (t_1', t_2) : A \otimes B$ and with (ii) providing the reqiured heap compatibility (by commutativity of +).

(2) (congPairR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')} \rightsquigarrow_{\otimes \mathrm{R}}$$

Essentially the same as the preceding case (congPairL) but by induction on the second premise.

- (tensorElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.
And two possible reductions:

(1) (pairBeta)

$$\frac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x \overset{s}{\mapsto} v_1 : A, y \overset{s}{\mapsto} v_2 : B \vdash t} \rightsquigarrow_{\otimes\beta}$$

with typing $\Gamma_2, x : A, y : B \vdash t_2 : C$ as the result i.e., with $\Gamma' = \Gamma_2, x : A, y : B$ and outgoing heap compatibility

(2) (pairElim)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\mathrm{LET}\otimes}$$

By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : A \otimes B$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying $\otimes_E$ again to get $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2 : C$ and with (ii) providing the reqiured heap compatibility (by commutativity of +).

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

Trivial case since there is no heap semantics rule as this is already a normal form value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

And two possible reductions:

(1) (betaUnit)

$$\frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \rightsquigarrow_s H \vdash t} \rightsquigarrow_{\beta\mathrm{UNIT}}$$

which refines the typing to:

$$\frac{0 \cdot \Gamma_1 \vdash () : \mathsf{unit} \quad \Gamma_2 \vdash t_2 : B}{0 \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = ()\ \mathbf{in}\ t_2 : B}\ 1_E$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma_1 + \Gamma_2)$ which refines to $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + s \cdot \Gamma_2$. The resulting typing is thus given by:

$$\frac{\Gamma_2 \vdash t_2 : B}{0 \cdot \Gamma_1 + \Gamma_2 \vdash t_2 : B}\ \textsc{Lemma C.4}$$

i.e., $\Gamma' = 0 \cdot \Gamma_1 + \Gamma_2$ and outgoing heap compatibility is provided exactly by the incoming heap compatibility.

(2) (congUnitElim)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ () = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\textsc{letunit}}$$

By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : \mathsf{unit}$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying 1_E again to get $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\ () = t_1'\ \mathbf{in}\ t_2 : B$ and with (ii) providing the reqiured heap compatibility (by commutativity of +).

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\ t : \Box_r A}\ \textsc{share}$$

And two possible reductions:

(1) (share)

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\textsc{share}\beta}$$

with refined (runtime) typing, and by Lemma C.2,

$$\frac{\dfrac{0 \cdot \Gamma_1', \gamma \vdash v : A}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash *v : *A}\ \textsc{nec}}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash \mathbf{share}\ (*v) : \Box_r A}\ \textsc{share}$$

and thus incoming heap compatibility $H, H' \bowtie \Gamma_0 + s \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma)$ which refines to: $H, H' \bowtie \Gamma_0 + 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$.

The resulting type is given by:

$$\frac{\dfrac{0 \cdot \Gamma_1', \gamma \vdash v : A}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash v : A}\ \textsc{Lemma C.4}}{r \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma) \vdash [v] : \Box_r A}\ \textsc{pr}$$

where $r \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma) = 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$ and thus $\Gamma' = 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$ and the outgoing heap compatibility is provided by the incoming heap compatibility but where the heap has zeroed out H (which does not affect the heap compatibility; it can be rederived with different fractions).

(2) (congShare)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\ t \rightsquigarrow_s H' \vdash \mathbf{share}\ t'}\ \rightsquigarrow_{\textsc{share}}$$

By induction on the premise with $\Gamma_0' = \Gamma_0$ then we have: (i) $\Gamma' \vdash t' : *A$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma'$. From which we construct the resulting typing, via applying SHARE again to get $\Gamma' \vdash \mathbf{share}\ t' : \Box_r A$ and with (ii) providing the reqiured heap compatibility.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \mathbf{clone'}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone'}$$

And two possible reductions:

(1) (cloneCongr)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ x\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\textsc{clone}}$$

Follows by induction similar to other congruences.

(2) (cloneBeta)

$$\frac{\Gamma \vdash [v] : \Box_r A \qquad \mathrm{dom}(H') \equiv \mathrm{arrRefs}(v) \qquad (H'', \theta, \overline{id}) = \mathrm{copy}(H')}{H, H' \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H, H', H'', x{\mapsto_s}\mathbf{pack}\ \langle \overline{id}, *(\theta(v))\rangle : *A \vdash t_2}\ \rightsquigarrow_{\textsc{clone}\beta}$$

refining the typing to:

$$\frac{\dfrac{\Gamma_1, \overline{id_1} \vdash v : A}{r \cdot \Gamma_1, \overline{id_1} \vdash [v] : \Box_r A}\ \textsc{pr} \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}]) \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{(r \cdot \Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone'}$$

with incoming heap compatibility $H, H' \bowtie \Gamma_0 + s \cdot (r \cdot \Gamma_1 + \Gamma_2)$.

Goal typing is then given by:

$$\Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}]) \vdash t_2 : \Box_r B$$

thus with $\Gamma' = \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}])$.

The typing of the extended heap is given by the value:

$$\frac{\dfrac{\dfrac{\Gamma_1, \overline{id_1} \vdash v : A}{\theta(\Gamma_1, \overline{id_1}) \vdash \theta(v) : \theta(A)}\ \textsc{Lemma C.5}}{\theta(\Gamma_1, \overline{id_1}) \vdash *(\theta(v)) : *\theta(A)}\ \textsc{nec}}{\theta(\Gamma_1), \overline{id} \vdash \mathbf{pack}\ \langle \overline{id}, *(\theta(v))\rangle : \exists \overline{id'}.*\theta(A)[id'/id]}\ \textsc{pack} \tag{1}$$

since θ maps each $\overline{id_1}$ to $\overline{id}$ then $\exists \overline{id'}.*\theta(A)[id'/id] = \exists \overline{id'}.*A[id'/id_1]$.

Goal heap comptibility: $(H, H', H'', x{\mapsto_s} * (\theta(v)) : *A) \bowtie (\Gamma_0 + s \cdot (\Gamma_2, x : *(\#A)))$ given by:

$$\frac{\dfrac{\dfrac{H, H' \bowtie (\Gamma_0 + s \cdot \Gamma_2 + r * s \cdot \Gamma_1)}{H, H', H'' \bowtie (\Gamma_0 + s \cdot \Gamma_2 + r * s \cdot \theta(\Gamma_1))} \qquad 1 \sqsubseteq r \qquad x \notin \mathrm{dom}(H) \quad (1) \qquad \exists r'.\ r + r' \equiv s}{(H, H', H'', x{\mapsto_s}\mathbf{pack}\ \langle \overline{id}, *(\theta(v))\rangle : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}])) \bowtie (\Gamma_0 + s \cdot \Gamma_2, x : [\exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}])]_s)}\ \textsc{ext}$$

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{with}\&$$

And three possible reductions:

(1) (congWithBorrowL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\text{WITH\&L}}$$

Which follows by induction similar to other inductive cases.

(2) (congWithBorrowR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\text{WITH\&R}}$$

Which follows by induction similar to other inductive cases.

(3) (withBorrow)

$$\frac{}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H \vdash \mathbf{unborrow}\ ([\&(*v)/x]t)}\ \rightsquigarrow_{\text{WITH\&}}$$

which refines the typing to:

$$\frac{\dfrac{\Gamma_2, x : \&_1 A \vdash t : \&_1 B}{\Gamma_2 \vdash \lambda x.t : \&_1 A \multimap \&_1 B}\ \text{WITH\&} \qquad \dfrac{\Gamma_1 \vdash v : A}{\Gamma_1 \vdash *v : *A}\ \text{NEC}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) : *B}\ \text{WITH\&}$$

with incoming heap compatibility $H \bowtie (\Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2))$.
Using the derivation:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{\Gamma_1 \vdash *v : *A}\ \text{NEC}}{\Gamma_1 \vdash \&(*v) : \&_1 A}\ \text{BORROW}$$

and with Theorem B.1 (linear substitution) with the typing of t here then we get:

$$\Gamma_1 + \Gamma_2 \vdash [\&(*v)/x]t : \&_1 B$$

which we use to get the resulting typing:

$$\frac{\Gamma_1 + \Gamma_2 \vdash [\&(*v)/x]t : \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unborrow}\ [\&(*v)/x]t : *A}\ \text{UNBORROW}$$

thus: $\Gamma' = \Gamma_1 + \Gamma_2$ and the goal heap compatibility is exactly the incoming heap compatibility.

- (unborrow)

$$\frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \mathbf{unborrow}\ t : *A}\ \text{UNBORROW}$$

And two possible reductions:

(1) (congUnborrow)

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \mathbf{unborrow}\ t \rightsquigarrow_s H \vdash \mathbf{unborrow}\ t'}\ \rightsquigarrow_{\text{UNBORROW}}$$

By induction as in the other inductive cases.

(2) (unborrowBorrw)

$$\frac{}{H \vdash \mathbf{unborrow}\ (\&(*v)) \rightsquigarrow_s H \vdash *v}\ \rightsquigarrow_{\text{UN\&}}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash v : A}{\Gamma \vdash (*v) : *A}\,\text{NEC}}{\Gamma \vdash \&(*v) : \&_1 A}\,\text{BORROW}}{\Gamma \vdash \textbf{unborrow}\ (\&(*v)) : *A}\,\text{UNBORROW}$$

and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot \Gamma$
The resulting typing matches thes goal as:

$$\dfrac{\Gamma \vdash v : A}{\Gamma \vdash (*v) : *A}\,\text{NEC}$$

thus $\Gamma' = \Gamma$ and outgoing heap compatibility is provided by the incoming compatibility.

- (push)

$$\dfrac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \textbf{push}\ t : (\&_p A) \otimes (\&_p B)}\,\text{PUSH}$$

And three possible reductions:

(1) (congPush)

$$\dfrac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{push}\ t \rightsquigarrow_s H' \vdash \textbf{push}\ t'}\,\rightsquigarrow_{\text{PUSH}}$$

Inductive case as in other inductive rules.

(2) (pushUnique)

$$\dfrac{}{H \vdash \textbf{push}\ *(v_1, v_2) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\,\rightsquigarrow_{\text{PUSH}*}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\,\text{NEC}}{\Gamma_1 + \Gamma_2 \vdash \textbf{push}\ (*(v_1, v_2)) : (*A) \otimes (*B)}\,\text{PUSH}$$

i.e., in the original typing $p = *$ and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$
The goal typing is then provided by:

$$\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\,\text{NEC} \qquad \dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\,\text{NEC}}{\Gamma_1 + \Gamma_2 \vdash (*v_1, *v_2) : (*A \otimes *B)}\otimes_I$$

with the goal heap compatbility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

(3) (pushBorrow)

$$\dfrac{}{H \vdash \textbf{push}\ \&(*(v_1, v_2)) \rightsquigarrow_s H \vdash (\&(*v_1), \&(*v_2))}\,\rightsquigarrow_{\text{PUSH}\&}$$

thus refining the typing to

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\text{NEC}}{\Gamma_1 + \Gamma_2 \vdash \&(*(v_1, v_2)) : \&_p(A \otimes B)}\text{BORROW}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{push}\,\&(*(v_1, v_2)) : (\&_p A) \otimes (\&_p B)}\text{PUSH}$$

and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$
The goal typing is then provided by:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\text{NEC}}{\Gamma_1 \vdash \&(*v_1) : \&_p A}\text{BORROW} \qquad \dfrac{\dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\text{NEC}}{\Gamma_2 \vdash \&(*v_2) : \&_p B}\text{BORROW}}{\Gamma_1 + \Gamma_2 \vdash (\&(*v_1), \&(*v_2)) : (\&_p A \otimes \&_p B)}\otimes_I$$

with thus $\Gamma' = \Gamma_1 + \Gamma_2$ and with the goal heap compatbility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

- (pull)

$$\dfrac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p(A \otimes B)}\text{PULL}$$

And three possible reductions:

(1) (congPull)

$$\dfrac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{pull}\, t \rightsquigarrow_s H' \vdash \mathbf{pull}\, t'} \rightsquigarrow_{\text{PULL}}$$

Inductive case as in other inductive rules.

(2) (pullUnique)

$$\dfrac{}{H \vdash \mathbf{pull}\,(*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)} \rightsquigarrow_{\text{PULL}*}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\text{NEC} \qquad \dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\text{NEC}}{\Gamma_1 + \Gamma_2 \vdash (*v_1, *v_2) : (*A \otimes *B)}\otimes_I}{\Gamma_1 + \Gamma_2 \vdash \mathbf{pull}\,(*v_1, *v_2) : *(A \otimes B)}\text{PULL}$$

thus with $p = *$ in the original typing and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$
The goal typing is then provided by:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\text{NEC}$$

with the goal heap compatbility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

(3) (pullBorrow)

$$\frac{}{H \vdash \mathbf{pull}\,(\&(*v_1), \&(*v_2)) \;\rightsquigarrow_s\; H \vdash \&(*(v_1, v_2))}\;\rightsquigarrow_{\text{PULL}\&}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\,\text{NEC}}{\Gamma_1 \vdash \&(*v_1) : \&_p A}\,\text{BORROW} \qquad \dfrac{\dfrac{\dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\,\text{NEC}}{\Gamma_2 \vdash \&(*v_2) : \&_p B}\,\text{BORROW}}{\Gamma_1 + \Gamma_2 \vdash (\&(*v_1), \&(*v_2)) : (\&_p A \otimes \&_p B)}\,\otimes_I}{\Gamma_1 + \Gamma_2 \vdash \mathbf{pull}\,(\&(*v_1), \&(*v_2)) : \&_p (A \otimes B)}\,\text{PULL}$$

and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$
The goal typing is then provided by:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\,\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\,\text{NEC}}{\Gamma_1 + \Gamma_2 \vdash \&(*(v_1, v_2)) : \&_p (A \otimes B)}\,\text{BORROW}$$

with thus $\Gamma' = \Gamma_1 + \Gamma_2$ and with the goal heap compatbility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

- (split)

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\; t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\;\text{SPLIT}$$

And three possible reductions:

(1) (congSplit)

$$\frac{H \vdash t \;\rightsquigarrow_s\; H' \vdash t'}{H \vdash \mathbf{split}\; t \;\rightsquigarrow_s\; H' \vdash \mathbf{split}\; t'}\;\rightsquigarrow_{\text{SPLIT}}$$

Inductive case as in other inductive rules.

(2) (splitArr)

$$\frac{\#a_1 \qquad \#a_2}{H, id \mapsto \mathbf{arr}, a \mapsto_p id \vdash \mathbf{split}\;(\&(*a)) \;\rightsquigarrow_s\; H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id \vdash (\&(*a_1), \&(*a_2))}\;\rightsquigarrow_{\text{SPLITARR}}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash a : \mathsf{Array}_{id}\, A}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash *a : *B}\,\text{NEC}}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash \&(*a) : \&_p B}\,\text{BORROW}}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A \vdash \mathbf{split}\;(\&(*a)) : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\,\text{SPLIT}$$

and thus heap compatibility is $(H, id \mapsto \mathbf{arr}) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\, A)$

The resulting typing is then given by:

$$\dfrac{\dfrac{\dfrac{0 \cdot \Gamma_1, a_1 : \mathsf{Array}_{id}\ A \vdash a_1 : \mathsf{Array}_{id}\ A}{0 \cdot \Gamma_1, a_1 : \mathsf{Array}_{id}\ A \vdash *a_1 : *(\mathsf{Array}_{id}\ A)}\ \textsc{nec}}{0 \cdot \Gamma_1, a_1 : \mathsf{Array}_{id}\ A \vdash \&(*a_1) : \&_p(\mathsf{Array}_{id}\ A)}\ \textsc{borrow} \quad \dfrac{\dfrac{0 \cdot \Gamma_2, a_2 : \mathsf{Array}_{id'}\ B \vdash a_2 : \mathsf{Array}_{id'}\ B}{0 \cdot \Gamma_2, a_2 : \mathsf{Array}_{id'}\ B \vdash *a_2 : *(\mathsf{Array}_{id'}\ B)}\ \textsc{nec}}{0 \cdot \Gamma_2, a_2 : \mathsf{Array}_{id'}\ B \vdash \&(*a_2) : \&_p(*(\mathsf{Array}_{id'}\ B))}\ \textsc{borrow}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B \vdash (\&(*v_1), \&(*v_2)) : (\&_p(\mathsf{Array}_{id}\ A) \otimes \&_p(\mathsf{Array}_{id}\ B))}\ \otimes_I$$

Goal compatibility is $(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B))$ which refines to

$$(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B)$$

which is constructed by:

$$\dfrac{\dfrac{(H, id \mapsto \mathbf{arr}) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A)}{(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A), a_1 : \mathsf{Array}_{id}\ A}\ \textsc{extRef}}{((H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id), a_2 \mapsto_{\frac{p}{2}} id) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A), a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B}\ \textsc{extRef}$$

satisfying the goal here.

(3) (splitPair)

$$\dfrac{\begin{array}{c} H \vdash \mathbf{split}\ (\&(*v)) \rightsquigarrow_s H' \vdash (\&(*v_1), \&(*v_2)) \\ H' \vdash \mathbf{split}\ (\&(*w)) \rightsquigarrow_s H'' \vdash (\&(*w_1), \&(*w_2)) \end{array}}{H \vdash \mathbf{split}\ (\&(*(v, w))) \rightsquigarrow_s H'' \vdash (\&(*(v_1, w_1)), \&(*(v_2, w_2)))}\ \rightsquigarrow_{\textsc{split}\otimes}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v : A \quad \Gamma_2 \vdash w : B}{\Gamma_1 + \Gamma_2 \vdash (v, w) : A \otimes B}\ \otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v, w) : *(A \otimes B)}\ \textsc{nec}}{\Gamma_1 + \Gamma_2 \vdash \&(*(v, w)) : \&_p(A \otimes B)}\ \textsc{borrow}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{split}\ (\&(*(v, w))) : \&_{\frac{p}{2}}(A \otimes B) \otimes \&_{\frac{p}{2}}(A \otimes B)}\ \textsc{split}$$

and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ By induction on the first premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ and second premise with $\Gamma_0'' = \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1' + s \cdot \Gamma_1$, providing:

(a) Γ_1' with $\Gamma_1' \vdash (\&(*v_1), \&(*v_2)) : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A$ and $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$

(b) Γ_2' with $\Gamma_2' \vdash (\&(*w_1), \&(*w_2)) : \&_{\frac{p}{2}} B \otimes \&_{\frac{p}{2}} B$ and $H'' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1' + s \cdot \Gamma_1 + s \cdot \Gamma_2'$

The resulting goal type derivation is then:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \quad \Gamma_1' \vdash w_1 : B}{\Gamma_1 + \Gamma_1' \vdash (v_1, w_1) : A \otimes B}\ \otimes_I}{\Gamma_1 + \Gamma_1' \vdash *(v_1, w_1) : *(A \otimes B)}\ \textsc{nec}}{\Gamma_1 + \Gamma_1' \vdash \&(*(v_1, w_1)) : \&_{\frac{p}{2}}(A \otimes B)}\ \textsc{borrow} \quad \dfrac{\dfrac{\dfrac{\Gamma_2 \vdash v_2 : A \quad \Gamma_2' \vdash w_2 : B}{\Gamma_2 + \Gamma_2' \vdash (v_2, w_2) : A \otimes B}\ \otimes_I}{\Gamma_2 + \Gamma_2' \vdash *(v_2, w_2) : *(A \otimes B)}\ \textsc{nec}}{\Gamma_2 + \Gamma_2' \vdash \&(*(v_2, w_2)) : \&_{\frac{p}{2}}(A \otimes B)}\ \textsc{borrow}}{\Gamma_1 + \Gamma_1' + \Gamma_2 + \Gamma_2' \vdash (\&(*(v_1, w_1)), \&(*(v_2, w_2))) : \&_{\frac{p}{2}}(A \otimes B) \otimes \&_{\frac{p}{2}}(A \otimes B)}\ \otimes_I$$

The goal compatibility is then $H'' \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_1' + \Gamma_2 + \Gamma_2')$ which is provided by the second induction with distributivity and commutativity of $+$.

- (join)

$$\frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

And four possible reductions:

(1) (congJoinL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ t_1'\ t_2}\ \rightsquigarrow_{\textsc{joinL}}$$

Inductive case as in other inductive rules.

(2) (congJoinR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\textsc{joinR}}$$

Inductive case as in other inductive rules.

(3) (joinArr)

$$\frac{\#a}{H, id \mapsto \mathbf{arr}, a_1 \mapsto_p id, a_2 \mapsto_q id \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) \rightsquigarrow_s H, id \mapsto \mathbf{arr}, a \mapsto_{(p+q)} id \vdash \&(*a)}\ \rightsquigarrow_{\textsc{joinArr}}$$

Dualising the splitArr cases exactly.

(4) (joinPair)

$$\frac{\begin{array}{c} H \vdash \mathbf{join}\ (\&(*v_1))\ (\&(*v_2)) \rightsquigarrow_s H' \vdash (\&(*v)) \\ H' \vdash \mathbf{join}\ (\&(*w_1))\ (\&(*w_2)) \rightsquigarrow_s H'' \vdash (\&(*w)) \end{array}}{H \vdash \mathbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) \rightsquigarrow_s H'' \vdash \&(*(v, w))}\ \rightsquigarrow_{\textsc{join}\otimes}$$

Dualising the joinPair cases exactly.

□

D UNIQUENESS AND BORROW SAFETY PROOFS

Lemma D.1 (Borrow safety). *For a well-typed term $\Gamma \vdash t_1 : A$, consider all types $\&_p A' \in A$ (subformulas of A with the form $\&_p A'$ for some A'). Then for all $id \in A'$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$, and given a single-step reduction $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$ then for all $a \in \mathsf{arrRefs}(t_1)$ (array references in t_1) such that $H(a) = id$ we have:*

$$\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1 \quad \vee \quad \sum_p a \mapsto_p id \in H' = 0$$

i.e., either *any array references contributing to the final term have their total fraction of* 1 *preserved from the incoming heap to the resulting term,* or *the total fraction in the resulting term is* 0 *(we have stopped tracking ownership information for the given identifier).*

Also, for all $a \in \mathsf{arrRefs}(t_1')$ (array references in t_1') such that $a \notin \mathsf{dom}(H)$ we have:

$$\exists id'.\ \sum_q a \mapsto_q id' \in H' = 1$$

i.e., any new array references contributing to the final term also have total fractions summing to 1.

Proof. By induction on typing $\Gamma \vdash t : A$.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

Then we also have a heap H such that $H \bowtie (0 \cdot \Gamma, x : A)$.
By inversion of heap compatibility, which implies that there exists a subheap H_1 such that $H = H_1, x{\mapsto_r}(\Gamma' \vdash v : A)$ (where there exists some r' such that $r' + 1 \sqsubseteq r$, and since $\downarrow v = \emptyset$):

$$\frac{H \bowtie 0 \cdot \Gamma + \Gamma' + \downarrow v \qquad x \notin \mathsf{dom}(H_1) \qquad \Gamma' \vdash v : A \qquad \exists r'.\ 1 + r' \equiv r}{(H_1, x{\mapsto_r}(\Gamma' \vdash v : A)) \bowtie (0 \cdot \Gamma, x : A)}\ \textsc{extLin}$$

and we have the reduction:

$$\frac{\exists r'.\ r' + 1 \sqsubseteq r}{H_1, x{\mapsto_r} v : A \vdash x \rightsquigarrow_s H_1, x{\mapsto_r} v : A \vdash v_{x:[A]_s}}\ \rightsquigarrow_{\textsc{var}}$$

Then for all $id \in A$ and all $a \in \mathsf{arrRefs}(v)$ such that $H(a) = id$ we have:

$$\sum_p a{\mapsto_p} id \in H = \sum_p a{\mapsto_p} id \in H' = 1$$

trivially as no array references were modified or manipulated here (just change in variables), and

$$(a \notin \mathsf{dom}(H)) \implies \exists id'. \sum_q a{\mapsto_q} id' \in H' = 1$$

trivially since the premise must always be false.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

Is trivial since there are no possible reductions.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and two reductions (beta and application congruence on the left) then several possible reductions following from primitive applications:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\textsc{appL}}$$

Then the goal follows by induction.

(2) Alternatively $t_1 = \lambda x.t_1'$ such that typing is:

$$\frac{\dfrac{\Gamma_1, x : A \vdash t_1' : B}{\Gamma_1 \vdash t_1' : A \multimap B}\ \textsc{abs} \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_1')\ t_2 : B}\ \textsc{app}$$

and we have reduction:

$$\frac{\Gamma_2 \vdash t_2 : A}{H \vdash (\lambda x.t_1')\ t_2 \rightsquigarrow_s H, x{\mapsto_s}(\Gamma_2 \vdash t_2 : A) \vdash t_1'}\ \rightsquigarrow_\beta$$

Then for all $id \in A$ and all $a \in \text{arrRefs}(t_1')$ such that $H(a) = id$ we have:

$$\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1$$

trivially as the right hand side does not refer to array references and the heap is preserved, and:

$$a \notin \text{dom}(H) \implies \exists id'. \sum_q a \mapsto_q id' \in H' = 1$$

is trivially true as the antecedent is always false.

(3) $t_1 = \textbf{newArray}$ therefore $A = \mathbb{N}$
Therefore we induct on the second argument:
- t_2 is a value and therefore by the value lemma (Lemma C.2) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \textbf{newArray}\ n : *(\text{Array}_{id}\ \mathbb{F})}\ \text{TyDerivednewArray}$$

with $H \bowtie (\Gamma_0 + \Gamma)$.
Thus there is a reduction as follows:

$$\frac{a \# H \qquad id \# H}{H \vdash \textbf{newArray}\ n \rightsquigarrow_s H, a \mapsto_1 id, id \mapsto \text{init} \vdash \textbf{pack}\ \langle id, *a \rangle}\ \rightsquigarrow_{\text{newArray}}$$

where $a \notin \text{dom}(H)$ but we have that $a \mapsto_1 id \in H, a \mapsto_1 id$ satisfying the goal.
- t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{newArray}\ t_2 \rightsquigarrow_s H' \vdash \textbf{newArray}\ t_2'}\ \rightsquigarrow_{\text{prim}}$$

Then the result holds by induction.

(4) $t_1 = \textbf{readArray}$ therefore $A = \&_p(\text{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\text{Array}_{id}\ \mathbb{F})$
and there is a reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{readArray}\ t_2 \rightsquigarrow_s H' \vdash \textbf{readArray}\ t_2'}\ \rightsquigarrow_{\text{prim}}$$

Therefore the borrow safety result holds by induction.

(5) $t_1 = \textbf{readArray}\ (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes *(\text{Array}_{id}\ \mathbb{F})$
- t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma C.2) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \text{Array}_{id}\ \mathbb{F} \vdash a : (\text{Array}_{id}\ \mathbb{F})\ \text{ARR}}{[\Gamma_1], a : \text{Array}_{id}\ \mathbb{F} \vdash *a : *(\text{Array}_{id}\ \mathbb{F})}\ *\text{ARR} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \text{Array}_{id}\ \mathbb{F} \vdash \textbf{readArray}\ (*a)\ i : \mathbb{F} \otimes *(\text{Array}_{id}\ \mathbb{F})}\ \text{TyDerivedreadArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \text{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \textbf{arr}$.
Then there is a reduction as follows:

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a \mapsto_p id, id \mapsto \textbf{arr}[i] = v \vdash \textbf{readArray}\ (*a)\ i \rightsquigarrow_s H, a \mapsto_p id, id \mapsto \textbf{arr}[i] = v \vdash (v, *a)}\ \rightsquigarrow_{\text{readArray}}$$

As the entirety of the heap is preserved (including the array reference a being read from here), the goal follows trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \leadsto_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\ (*a)\ t_2 \leadsto_s H' \vdash \mathbf{readArray}\ (*a)\ t_2'} \leadsto_{\text{PRIM}}$$

And therefore the borrow safety result holds by induction.

(6) $t_1 = \mathbf{readArray}\ (\&(*a))$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

– t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma C.2) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{ARR}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \&\text{ARR} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\ (\&(*a))\ i : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\text{TyDerivedreadArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\frac{\emptyset \vdash v : \mathbb{F}}{H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\ (\&(*a))\ i \leadsto_s H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash (v, \&(*a))} \leadsto_{\text{read\&Arra}}$$

As the entirety of the heap is preserved (including the array reference a being read from here), the goal follows trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \leadsto_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\ (\&(*a))\ t_2 \leadsto_s H' \vdash \mathbf{readArray}\ (\&(*a))\ t_2'} \leadsto_{\text{PRIM}}$$

And therefore the borrow safety result holds by induction.

(7) $t_1 = \mathbf{writeArray}$ therefore $A = \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F})$ with reduction

$$\frac{H \vdash t_2 \leadsto_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ t_2 \leadsto_s H' \vdash \mathbf{writeArray}\ t_2'} \leadsto_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

(8) $t_1 = \mathbf{writeArray}\ (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}_{id}\ \mathbb{F})$ with reduction:

$$\frac{H \vdash t_2 \leadsto_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ t_2 \leadsto_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ t_2'} \leadsto_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

(9) $t_1 = \mathbf{writeArray}\ (\&(*a))$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap \&_1(\mathsf{Array}_{id}\ \mathbb{F})$ with reduction:

$$\frac{H \vdash t_2 \leadsto_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ t_2 \leadsto_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ t_2'} \leadsto_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

(10) $t_1 = \mathbf{writeArray}\ (*a)\ i$ therefore $A = \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})$

– t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma C.2) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{arr}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : *(\mathsf{Array}_{id}\ \mathbb{F})}\ {}^{*}\textsc{arr} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\ (*a)\ i\ f : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{TyDerivedwriteArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\dfrac{}{H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (*a)\ i\ v \rightsquigarrow_s H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr}[i] = v \vdash *a}\ \rightsquigarrow_{\textsc{writeArray}}$$

Here, the only change in the heap is to the array value that id is pointing to. As the remainder of the heap is preserved, both parts of the goal follow trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (*a)\ i\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (*a)\ i\ t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

Therefore the borrow safety result holds by induction.

(11) $t_1 = \mathbf{writeArray}\ (\&(*a))\ i$ therefore $A = \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})$

– t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma C.2) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{arr}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \&\textsc{arr} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ f : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{TyDerivedwriteArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr}$.
Then there is a reduction as follows:

$$\dfrac{}{H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\ (\&(*a))\ i\ v \rightsquigarrow_s H, a{\mapsto_p}\, id, id \mapsto \mathbf{arr}[i] = v \vdash \&(*a)}\ \rightsquigarrow_{\textsc{write\&Array}}$$

Here, the only change in the heap is to the array value that id is pointing to. As the remainder of the heap is preserved, both parts of the goal follow trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\ (\&(*a))\ i\ t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

Therefore the borrow safety result holds by induction.

(12) $t_1 = \mathbf{deleteArray}$ therefore $A = *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}$
The result type here is 1, so cannot contain any types of the form $\&_p A$, and therefore the result holds trivially.

• (pr)

$$\dfrac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \square_r A}\ \textsc{pr}$$

with a heap H such that $H \bowtie (r \cdot \Gamma)$ and reduction:

$$H \vdash [t] \rightsquigarrow_s H' \vdash t''$$

which has only one possible derivation:

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t' \quad \Gamma \vdash [t] : \Box_r A}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']} \rightsquigarrow_{\Box}$$

Thus, induction provides the goal.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 \rightsquigarrow_s H' \vdash t'$$

which has two possible derivations:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 \rightsquigarrow_s H' \vdash \textbf{let}\ [x] = t_1'\ \textbf{in}\ t_2} \rightsquigarrow_{\text{LET}\Box}$$

Then induction provides the goal.

(2) Alternatively $t_1 = [v]$ such that the typing is:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v] : \Box_r A}\ \text{PR} \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = [v]\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

and we have reduction:

$$\frac{r \cdot \Gamma_1 \vdash [v] : \Box_r A}{H \vdash \textbf{let}\ [x] = [v]\ \textbf{in}\ t_2 \rightsquigarrow_s H, x \mapsto_{s*r} v : A \vdash t_2} \rightsquigarrow_{\Box\beta}$$

Which trivially satisfies the goal since all array references are then in H and we get the conditions trivially (since no array references are manipulated) similar to the β proof above.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{DER}$$

with a heap H such that $H \bowtie (\Gamma, x : [A]_1)$ and reduction:

$$H \vdash t \rightsquigarrow_s H' \vdash t'$$

As the term t is unchanged from the premise, the goal holds by induction regardless of how this reduction is derived.

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \text{APPROX}$$

with a heap H such that $H \bowtie (\Gamma, x : [A]_s, \Gamma')$ and reduction:

$$H \vdash t \rightsquigarrow_s H' \vdash t'$$

As the term t is unchanged from the premise, the goal holds by induction regardless of how this reduction is derived.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)}\ \rightsquigarrow_{\otimes \mathrm{L}}$$

Here, induction on t_1 provides the goal.

(2) Otherwise, $t_1 = v$ for some value v and we can perform the reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')}\ \rightsquigarrow_{\otimes \mathrm{R}}$$

Here, induction on t_2 provides the goal.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 : C}\ \otimes_E$$

Two possible reductions:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ (x, y) = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\mathrm{LET}\otimes}$$

and induction provides the goal.

(2) otherwise $t_1 = (v_1, v_2)$ such that the typing is:

$$\frac{\dfrac{\Gamma_3 \vdash v_1 : A \quad \Gamma_4 \vdash v_2 : B}{\Gamma_3 + \Gamma_4 \vdash (v_1, v_2) : (A \otimes B)}\ \otimes_I \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_3 + \Gamma_4 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = (v_1, v_2)\ \mathbf{in}\ t_2 : C}\ \otimes_E$$

and we have reduction:

$$\frac{\Gamma_1 \vdash t_1' : A \quad \Gamma_2 \vdash t_1'' : B}{H \vdash \mathbf{let}\ (x, y) = (t_1', t_1'')\ \mathbf{in}\ t_3 \rightsquigarrow_s H, x \mapsto_s (\Gamma_1 \vdash t_1' : A), y \mapsto_s (\Gamma_2 \vdash t_1'' : B) \vdash t_3}\ \rightsquigarrow_{\otimes\beta}$$

Then for all $id \in A$ and all $a \in \mathrm{arrRefs}(t_3)$ such that $H(a) = id$ we have:

$$\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1$$

and also

$$a \notin \mathrm{dom}(H) \implies \exists id'. \sum_q a \mapsto_q id' \in H' = 1$$

where the first statement holds trivially since there is no manipulation of the array references here and the second statement is always true.

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

The result type here is 1, so cannot contain any types of the form $\&_p A$, and therefore the result holds trivially.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 : B}\ 1_E$$

Following essentially the same structure as the tensor proof where array reference counting is not used so induction provides the goal.

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\ t : \Box_r A}\ \textsc{share}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{share}\ t \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

(1)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\ t \rightsquigarrow_s H' \vdash \mathbf{share}\ t'}\ \rightsquigarrow_{\textsc{share}}$$

Here, induction provides the goal.

(2) t has the form $*v$ for some value v, and we can reduce:

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\textsc{share}\beta}$$

Here, for any instance of $a \mapsto_p id$ that exists in the initial heap for any a or id, we now instead have $a \mapsto_0 id$ in the resulting heap. Hence it is clear for all $id \in A$ and all $a \in \mathsf{arrRefs}(t)$ such that $H(a) = id$:

$$\sum_p a \mapsto_p id \in H' = 0$$

holds by default.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \mathbf{clone'}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone'}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\ t_1'\ \mathbf{as}\ x\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\textsc{clone}}$$

Here, induction provides the goal.

(2) t_1 has the form $[v]$ for some value v, and we can reduce:

$$\frac{\Gamma \vdash [v] : \Box_r A \qquad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H')}{H, H' \vdash \mathbf{clone}\ [v]\ \mathbf{as}\ x\ \mathbf{in}\ t_2 \rightsquigarrow_s H, H', H'', x \mapsto_s \mathbf{pack}\ \langle \overline{id}, *(\theta(v)) \rangle : *A \vdash t_2}\ \rightsquigarrow_{\textsc{clone}\beta}$$

Here, all of the array references from the original heap H (separated out as H') are copied and given a new identifier, to form H''. In other words, for every $a \mapsto_p id$, $id \mapsto \mathbf{arr}$ in H, there now exists a new a' and id' in H'' such that $a' \mapsto_p id'$, $id' \mapsto \mathbf{arr}$.

Therefore, for all $id \in A$ and all $a \in \text{arrRefs}(t_1)$ such that $H(a) = id$ we have:

$$\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1$$

because none of the references in the preexisting heap have been modified in the new heap, and there are no new references with the same identifiers as any references appearing in H'' have fresh identifiers.
We also have that:

$$a' \notin \text{dom}(H) \implies \exists id'. \sum_q a' \mapsto_q id' \in H' = 1$$

because any $a' \notin \text{dom}(H)$ must be in H'', and so it matches up with another reference a appearing in H, for which we have $\sum_p a \mapsto_p id \in H = 1$ by the above argument.

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ t_1\ t_2 : *B}\ \text{WITH\&}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \textbf{withBorrow}\ f\ t \rightsquigarrow_s H' \vdash t''$$

which has three possible derivations:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\text{WITH\&L}}$$

Here, induction on t_1 provides the goal.

(2) f has the form $(\lambda x.t_1)$, and we can reduce:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\text{WITH\&R}}$$

Here, induction on t_2 provides the goal.

(3) As above, but t also has the form $(*v)$, and we can reduce:

$$\frac{}{H \vdash \textbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H \vdash \textbf{unborrow}\ ([\&(*v)/x]t)}\ \rightsquigarrow_{\text{WITH\&}}$$

Here, the goal is trivial since the heap H is preserved.

- (split)

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \textbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \text{SPLIT}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \textbf{split}\ t \rightsquigarrow_s H' \vdash t''$$

which has three possible derivations:

(1)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{split}\ t \rightsquigarrow_s H' \vdash \textbf{split}\ t'}\ \rightsquigarrow_{\text{SPLIT}}$$

Here, induction provides the goal.

(2) t has the form $\&(*a)$ such that the typing is:

$$\frac{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{\&arr}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{split}\ (\&(*a)) : \&_{\frac{p}{2}}(\mathsf{Array}_{id}\ \mathbb{F}) \otimes \&_{\frac{p}{2}}(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{split}$$

and we have reduction:

$$\frac{\#a_1 \qquad \#a_2}{H, id \mapsto \mathbf{arr}, a \mapsto_p id \vdash \mathbf{split}\ (\&(*a)) \rightsquigarrow_s H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id \vdash (\&(*a_1), \&(*a_2))}\ \rightsquigarrow_{\textsc{splitArr}}$$

Here, for all $id \in A$ and all $a' \in \mathsf{arrRefs}(t)$ such that $H(a') = id$ we have:

$$\sum_{p}^{'} a' \mapsto_{p'} id \in H = \sum_{p}^{'} a' \mapsto_{p'} id \in H' = 1$$

because the sum of p' over all references in H is equal to $p + x$ for some x (potentially zero) representing any remaining references besides a, and the sum of p' in H' is now equal to $\frac{p}{2} + \frac{p}{2} + x$, which is equivalent.

(3) t has the form $\&(*(v, w))$ with reduction:

$$\frac{\begin{array}{l} H \vdash \mathbf{split}\ (\&(*v)) \rightsquigarrow_s H' \vdash (\&(*v_1), \&(*v_2)) \\ H' \vdash \mathbf{split}\ (\&(*w)) \rightsquigarrow_s H'' \vdash (\&(*w_1), \&(*w_2)) \end{array}}{H \vdash \mathbf{split}\ (\&(*(v, w))) \rightsquigarrow_s H'' \vdash (\&(*(v_1, w_1)), \&(*(v_2, w_2)))}\ \rightsquigarrow_{\textsc{split}\otimes}$$

Induction over the two premises gives us the result here, by threading the theorem's implications from H to H' via the first premise and then H' to H'' via the second.

- (join)

$$\frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash t''$$

which has four possible derivations:

(1)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ t_1'\ t_2}\ \rightsquigarrow_{\textsc{joinL}}$$

Here, induction over t_1 provides the goal.

(2) $t_1 = v$ for some value v, and we can reduce:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\textsc{joinR}}$$

Here, induction over t_2 provides the goal.

(3) As above, but t_2 has the form $(\&(*a_2))$, which restricts v to have the form $(\&(*a_1))$ such that the typing is:

$$\frac{[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a_1) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{\&arr} \qquad [\Gamma], a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \&(*a_2) : \&_q(\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{\&arr}}{[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F}, a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) : \&_{p+q}(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{join}$$

and we have reduction:

$$\frac{\#a}{H, id \mapsto \mathbf{arr}, a_1 \mapsto_p id, a_2 \mapsto_q id \vdash \mathbf{join}\ (\&(*a_1))\ (\&(*a_2)) \rightsquigarrow_s H, id \mapsto \mathbf{arr}, a \mapsto_{(p+q)} id \vdash \&(*a)}\ \rightsquigarrow_{\textsc{joinArr}}$$

Here, for all $id \in A$ and all $a' \in \text{arrRefs}(t_1)$ and $\text{arrRefs}(t_2)$ such that $H(a') = id$ we have:

$$\sum_{p}^{'} a' \mapsto_{p'} id \in H = \sum_{p}^{'} a' \mapsto_{p'} id \in H' = 1$$

because the sum of p' over all references in H is equal to $p + q + x$ for some x (potentially zero) representing any remaining references besides a_1 and a_2, and the sum of p' in H' is now equal to $(p + q) + x$, which is equivalent.

(4) t_2 has the form $(\&(*(v_2, w_2)))$, which restricts t_1 to have the form $(\&(*(v_1, w_1)))$, allowing the reduction:

$$\frac{\begin{array}{l} H \vdash \textbf{join}\ (\&(*v_1))\ (\&(*v_2)) \rightsquigarrow_s H' \vdash (\&(*v)) \\ H' \vdash \textbf{join}\ (\&(*w_1))\ (\&(*w_2)) \rightsquigarrow_s H'' \vdash (\&(*w)) \end{array}}{H \vdash \textbf{join}\ (\&(*(v_1, w_1)))\ (\&(*(v_2, w_2))) \rightsquigarrow_s H'' \vdash \&(*(v, w))}\ \rightsquigarrow_{\text{JOIN}\otimes}$$

Induction over the two premises gives us the result here, by threading the theorem's implications from H to H' via the first premise and then H' to H'' via the second.

- (push)

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \textbf{push}\ t : (\&_p A) \otimes (\&_p B)}\ \text{PUSH}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \textbf{push}\ t \rightsquigarrow_s H' \vdash t''$$

which has three possible derivations:

(1)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{push}\ t \rightsquigarrow_s H' \vdash \textbf{push}\ t'}\ \rightsquigarrow_{\text{PUSH}}$$

Here, induction provides the goal.

(2) t has the form $*(v_1, v_2)$ and we can reduce:

$$\frac{}{H \vdash \textbf{push}\ *(v_1, v_2) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\ \rightsquigarrow_{\text{PUSH}*}$$

Here, the goal is trivial since the heap H is preserved.

(3) t has the form $\&(*(v_1, v_2))$ and we can reduce:

$$\frac{}{H \vdash \textbf{push}\ \&(*(v_1, v_2)) \rightsquigarrow_s H \vdash (\&(*v_1), \&(*v_2))}\ \rightsquigarrow_{\text{PUSH}\&}$$

Here, the goal is trivial since the heap H is preserved.

- (pull)

$$\frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \textbf{pull}\ t : \&_p(A \otimes B)}\ \text{PULL}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \textbf{pull}\ t \rightsquigarrow_s H' \vdash t''$$

which has three possible derivations:

(1)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{pull}\ t \rightsquigarrow_s H' \vdash \textbf{pull}\ t'}\ \rightsquigarrow_{\text{PULL}}$$

Here, induction provides the goal.

(2) t has the form $(*v_1, *v_2)$ and we can reduce:

$$\frac{}{H \vdash \mathbf{pull}\,(*v_1, *v_2) \;\leadsto_s\; H \vdash *(v_1, v_2)} \;\leadsto_{\text{PULL}*}$$

Here, the goal is trivial since the heap H is preserved.

(3) t has the form $(\&(*v_1), \&(*v_2))$ and we can reduce:

$$\frac{}{H \vdash \mathbf{pull}\,(\&(*v_1), \&(*v_2)) \;\leadsto_s\; H \vdash \&(*(v_1, v_2))} \;\leadsto_{\text{PULL}\&}$$

Here, the goal is trivial since the heap H is preserved.

- (newArray) (readArray) (writeArray) (deleteArray)
  All trivial as they have no reductions.
- (pack)

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\,\langle id', t\rangle : \exists id.A[id/id']} \;\text{PACK}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{pack}\,\langle id, t\rangle \;\leadsto_s\; H' \vdash t''$$

which has one possible derivation:

$$\frac{H \vdash t \;\leadsto_s\; H \vdash t'}{H \vdash \mathbf{pack}\,\langle id, t\rangle \;\leadsto_s\; H \vdash \mathbf{pack}\,\langle id, t'\rangle} \;\leadsto_{\text{PACK}}$$

Here, the goal is trivial since the heap H is preserved.

- (unpack)

$$\frac{\begin{array}{l}\Gamma_1 \vdash t_1 : \exists id.A \\ \quad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1 \,\mathbf{in}\, t_2 : B} \;\text{UNPACK}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1 \,\mathbf{in}\, t_2 \;\leadsto_s\; H' \vdash t''$$

which has two possible derivations:

(1)

$$\frac{}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = \mathbf{pack}\,\langle id', v\rangle \,\mathbf{in}\, t \;\leadsto_s\; H[id'/id], x \mapsto_r v \vdash t} \;\leadsto_{\exists\beta}$$

Here, the heap is preserved with the exception of x and some renaming of identifiers. Since no array references are affected, the goal is achieved directly.

(2)

$$\frac{H \vdash t_1 \;\leadsto_s\; H \vdash t_1'}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1 \,\mathbf{in}\, t_2 \;\leadsto_s\; H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1' \,\mathbf{in}\, t_2} \;\leadsto_{\text{UNPACK}}$$

Here, the goal is trivial since the heap H is preserved.

□

Lemma D.2 (Borrow safety over multi-reduction). *For a well-typed term $\Gamma \vdash t_1 : *A$, then for all $id \in A$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$, and given a multi-step reduction $H \vdash t_1 \;\Rightarrow_s\; H' \vdash v$ then for all $a \in \mathsf{arrRefs}(t_1)$ (array references in t_1) such that $H(a) = id$ we have:*

$$\sum_p a \mapsto_p id \in H = 1 \implies \exists a'.a' \mapsto_1 id \in H'$$

i.e., any array references contributing to the final term have their total fraction of 1 *preserved from the incoming heap to the resulting term, with this fraction now contained in a single reference.*

Also, for all $a \in \mathsf{arrRefs}(t_1')$ *(array references in* t_1'*) such that* $a \notin \mathsf{dom}(H)$ *we have:*

$$\exists id'.\, a \mapsto_1 id' \in H'$$

i.e., any new array references contributing to the final term uniquely point to their identifier, and thus are annotated with the fraction 1.

Proof. By induction on the structure of the multi-reduction $H \vdash t_1 \implies_s H' \vdash v$.

- (refl)

$$\frac{}{H \vdash v \implies_s H \vdash v}\ \textsc{refl}$$

Since v is a value which we know has type $*A$, then by the unique value lemma there are two possibilities for the form of v.

(1) $A = \mathsf{Array}_{id}\ \mathbb{F}$, and so v has the form $*a$. This restricts the typing as follows:

$$\frac{}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : *(\mathsf{Array}_{id}\ A)}\ *\textsc{arr}$$

Then we also have a heap H such that $H \bowtie (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ \mathbb{F})$ which by inversion of heap-compatibility for array references implies that there exists a subheap H_1 such that $H = H_1, a \mapsto_1 id, id \mapsto \mathbf{arr}$.

As we have a single reference in the heap annotated with 1, both the premise and the goal of the implication in the theorem hold.

(2) $A = A' \otimes B$, and so v has the form (v_1, v_2). This restricts the typing to two possible derivations:

$$\frac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash v_2 : B}{\gamma_1 + \gamma_2 \vdash (v_1, v_2) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \textsc{nec}$$

$$\frac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A'}{\gamma_1 \vdash *v_1 : *A'}\ \textsc{nec} \quad \dfrac{\gamma_2 \vdash v_2 : B}{\gamma_2 \vdash *v_2 : *B}\ \textsc{nec}}{\gamma_1 + \gamma_2 \vdash (*v_1, *v_2) : *A' \otimes *B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \textsc{pull}$$

In either case, by the unique value lemma we know that v_1 and v_2 are both restricted: they can either be of the form a, meaning that one of the types in the product is $\mathsf{Array}_{id}\ \mathbb{F}$, or they can be of the form (v_3, v_4), meaning that one of the types is $A'' \otimes B'$.

Proceed by inspecting each of v_1 and v_2 in turn.

If the value is of the form a, then as in the above case by inversion of heap compatibility the heap must contain a unique array reference $a \mapsto_1 id, id \mapsto \mathbf{arr}$, which must satisfy the theorem as it is annotated with the fraction 1.

If the value is of the form (v_3, v_4), then we can restrict the typing of this subpart of the derivation by exactly the above argument, and then inspect the form of v_3 and v_4 using the unique value lemma following the same logic. This proceeds inductively; as typing derivations are finite trees, this must eventually terminate in an array reference which satisfies the theorem at every leaf of the tree.

- (ext)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_2 \qquad H' \vdash t_2 \Rightarrow_s H'' \vdash t_3}{H \vdash t_1 \Rightarrow_s H'' \vdash t_3}\ \textsc{ext}$$

First, consider the first premise, which is a single-step reduction of the form $H \vdash t_1 \rightsquigarrow_s H' \vdash t_2$.
From Theorem D.1, we know that for all $id \in A$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$, then for all $a \in \text{arrRefs}(t_1)$ (array references in t_1) such that $H(a) = id$ we have:

$$\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1$$
$$\vee \sum_p a \mapsto_p id \in H' = 0$$

and also that for all $a \in \text{arrRefs}(t_2)$ (new array references in t_2) such that $a \notin \text{dom}(H)$ we have:

$$\exists id'. \sum_q a \mapsto_q id' \in H' = 1$$

Now, we induct over the second premise, which is a multi-reduction of the form $H' \vdash t_2 \Rightarrow_s H'' \vdash t_3$. Induction using the present theorem tells us that for all $id \in A$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$, then for all $a \in \text{arrRefs}(t_2)$ (array references still present in t_2) such that $H(a) = id$ we have:

$$\sum_p a \mapsto_p id \in H' = 1 \implies \exists a'. a' \mapsto_1 id \in H''$$

From our knowledge about the first premise (the single-step reduction), there are two cases.

- $\sum_p a \mapsto_p id \in H = \sum_p a \mapsto_p id \in H' = 1$.
  Then we know that fractions must sum to 1 for both array references preserved from t_1 and also new array references preserved from t_2, but via the implication we obtained from induction on the second premise, we know $\exists a'. a' \mapsto_1 id \in H''$, which is exactly the required result.
- $\sum_p a \mapsto_p id \in H' = 0$.
  Then we only need to concern ourselves with new array references preserved from t_2, but via the same implication we obtained via induction on the second premise, we know $\exists a'. a' \mapsto_1 id \in H''$, again exactly as required.

□

Corollary D.3 (Uniqueness). *For a well-typed term $\Gamma \vdash t : *A$ and all Γ_0 and H such that $H \bowtie (\Gamma_0 + \Gamma)$ and given a multi-reduction to a value $H \vdash t \Rightarrow_s H' \vdash *v$, for all $a \in \text{arrRefs}(v)$ (array references in v) we have:*

$$a \mapsto_1 id \in H \implies a \mapsto_1 id \in H' \quad \wedge \quad a \notin \text{dom}(H) \implies \exists id'. a \mapsto_1 id' \in H'$$

i.e., any array references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new array references contributing to the final term are also unique.

Proof. Follows directly from Lemma D.2, in the subcase where only one reference exists in the initial heap (since $\sum_p a \mapsto_p id \in H = 1$ must hold if there exists a single reference such that $a \mapsto_1 id \in H$). □

e

E SOUNDNESS OF HEAP MODEL WRT. EQUATIONAL THEORY

Theorem E.1 (Soundness with respect to the equational theory). *For all t_1, t_2 such that $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_2 : A$ and $t_1 \equiv t_2$ and given H such that $H \bowtie \Gamma$, there exists a value (irreducible term) v such that there are full β-reductions to the same value*

$$H \vdash t_1 \Rightarrow_\beta H' \vdash v \quad \wedge \quad H \vdash t_2 \Rightarrow_\beta H'' \vdash v$$

where full β-reduction includes all congruences to evaluate inside λs, etc.

Proof. • (unitL)

$$\frac{}{\mathbf{clone}\ (\mathbf{share}\ v)\ \mathbf{as}\ x\ \mathbf{in}\ t' \equiv [\mathbf{pack}\ \langle \overline{id}, v\rangle/x]t'}\ \textsc{EqUnitL}$$

With typing derivation for the LHS:

$$\frac{\dfrac{\Gamma_1, \overline{id} \vdash v : *A}{\Gamma_1, \overline{id} \vdash \mathbf{share}\ v : \Box_r A}\ \textsc{Tyreturn} \qquad \mathrm{noIDs}(\Gamma_1) \quad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t : \Box_r B \quad 1 \sqsubseteq r}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\ (\mathbf{share}\ v)\ \mathbf{as}\ x\ \mathbf{in}\ t : \Box_r B}\ \textsc{clone}$$

And typing derivation for the RHS:

$$(\Gamma_1 + \Gamma_2), \overline{id} \vdash [\mathbf{pack}\ \langle \overline{id}, v\rangle/x]t : \Box_r B$$

By the value lemma (Lemma C.2) we know that $v = *v'$ therefore we know that we can perform the following reduction (where we elide the output binding context and usage context as they are not needed in the proof):

$$\begin{array}{rcl} & & H_1, H_2 \vdash \mathbf{clone}\ (\mathbf{share}\ (*v'))\ \mathbf{as}\ x\ \mathbf{in}\ t \\ \rightsquigarrow_{\textsc{clone}} + \rightsquigarrow_{\textsc{share}\beta} & \longrightarrow & [H_1]_0, H_2 \vdash \mathbf{clone}\ [v']\ \mathbf{as}\ x\ \mathbf{in}\ t \\ \rightsquigarrow_{\textsc{clone}\beta} & \rightsquigarrow & [H_1]_0, H_2, H_3, x \mapsto_r \mathbf{pack}\ \langle \overline{id}, \theta(v)\rangle : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t \end{array}$$

where $\mathrm{dom}(H_1) \equiv \mathrm{arrRefs}(v)$ and $(H_3, \theta, \overline{id}) = \mathrm{copy}(H_1)$

• (assoc)

$$\frac{x \# t_3}{\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ (\mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3) \equiv \mathbf{clone}\ (\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)\ \mathbf{as}\ y\ \mathbf{in}\ t_3}\ \textsc{EqAssoc}$$

With typing for the LHS:

$$\frac{\Gamma_1, \overline{id_1} \vdash t_1 : \Box_{r_1} A_1 \qquad \dfrac{\Gamma_2, \overline{id_2}, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) \vdash t_2 : \Box_{r_2} A_2 \quad \Gamma_3, y : \exists \overline{id_2'}.*(A_2[\overline{id_2'}/\overline{id_2}]) \vdash t_3 : \Box_{r_3} A_3}{\Gamma_2, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) + \Gamma_3 \vdash \mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3 : \Box_{r_3} A_3}\ \textsc{Tybind}}{(\Gamma_1 + \Gamma_2 + \Gamma_3), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ (\mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3) : \Box_{r_3} A_3}\ \textsc{Tybind}$$

where $\mathrm{noIDs}(\Gamma_1)$ and $\mathrm{noIDs}(\Gamma_2)$ and $1 \sqsubseteq r_1$ and $1 \sqsubseteq r_2$.
And with RHS typing, with the same conditions:

$$\frac{\dfrac{\Gamma_1, \overline{id_1} \vdash t_1 : \Box_{r_1} A_1 \quad \Gamma_2, \overline{id_2}, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) \vdash t_2 : \Box_{r_2} A_2}{(\Gamma_1 + \Gamma_2), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_{r_2} A_2}\ \textsc{Tybind} \qquad \Gamma_3, y : \exists \overline{id_2'}.*(A_2[\overline{id_2'}/\overline{id_2}]) \vdash t_3 : \Box_{r_3} A_3}{(\Gamma_1 + \Gamma_2 + \Gamma_3), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ (\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)\ \mathbf{as}\ y\ \mathbf{in}\ t_3 : \Box_{r_3} A_3}\ \textsc{Tybind}$$

There are four possibilities depending on the reduction of t_1 and t_2.

(1) Divergence: $t_1 \rightarrow^{\omega}$ (i.e., t_1 diverges). In which case then $H \vdash \textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ (\textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3) \rightsquigarrow_{\text{CLONE}}{}^{\omega}$ (diverging) and also $H \vdash \textbf{clone}\ (\textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2)\ \textbf{as}\ y\ \textbf{in}\ t_3 \rightsquigarrow_{\text{CLONE}}{}^{\omega}$ and so the the equation is trivially satisfied as both sides diverge.
If t_1 converges, but t_2 diverges then both sides diverge by similar reasoning. Similarly if both diverge then overall both sides diverge.

(2) Convergence: t_1 reduces to a value v_1 and t_2 reduces to a value v_2. By the typing and the value lemma (Lemma C.2) then $\exists v_1'.\ v_1 = [v_1']$ and then $\exists v_2'.\ v_2 = [v_2']$.
Then we can reduce as follows on the LHS:

$$
\begin{array}{rcll}
 & & H \vdash \textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ (\textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3) & \\
 & \rightsquigarrow * & H' \vdash \textbf{clone}\ [v_1']\ \textbf{as}\ x\ \textbf{in}\ (\textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3) & \\
\rightsquigarrow_{\text{CLONE}\beta} & \rightsquigarrow & H_1, H_1', H_1'', x \mapsto_r \textbf{pack}\ \langle \overline{id_1}, *\theta(v_1)\rangle \vdash \textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3 & \text{dom}(H_1') \equiv \text{arrRefs}(v_1)\ \wedge\ (H_1'', \theta, \overline{id_1}) = \text{copy}(H_1') \\
 & \rightsquigarrow * & H'' \vdash \textbf{clone}\ [v_2]\ \textbf{as}\ y\ \textbf{in}\ t_3 & \\
\rightsquigarrow_{\text{CLONE}\beta} & \rightsquigarrow & H_2, H_2', H_2'', y \mapsto_r \textbf{pack}\ \langle \overline{id_2}, *\theta'(v_2)\rangle \vdash t_3 & \text{dom}(H_2') \equiv \text{arrRefs}(v_2)\ \wedge\ (H_2'', \theta, \overline{id_2}) = \text{copy}(H_2')
\end{array}
$$

and on the RHS:

$$
\begin{array}{rcll}
 & & H \vdash \textbf{clone}\ (\textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2)\ \textbf{as}\ y\ \textbf{in}\ t_3 & \\
 & \rightsquigarrow * & H' \vdash \textbf{clone}\ (\textbf{clone}\ [v_1]\ \textbf{as}\ x\ \textbf{in}\ t_2)\ \textbf{as}\ y\ \textbf{in}\ t_3 & \\
 & \rightsquigarrow & H_1, H_1', H_1'', x \mapsto_r \textbf{pack}\ \langle \overline{id_1}, *\theta(v_1)\rangle \vdash \textbf{clone}\ t_2\ \textbf{as}\ y\ \textbf{in}\ t_3 & \text{dom}(H_1') \equiv \text{arrRefs}(v_1)\ \wedge\ (H_1'', \theta, \overline{id_1'}) = \text{copy}(H_1') \\
 & \rightsquigarrow * & H'' \vdash \textbf{clone}\ [v_2]\ \textbf{as}\ y\ \textbf{in}\ t_3 & \\
\rightsquigarrow_{\text{CLONE}\beta} & & H_2, H_2', H_2'', y \mapsto_r \textbf{pack}\ \langle \overline{id_2}, *\theta'(v_2)\rangle \vdash t_3 & \text{dom}(H_2') \equiv \text{arrRefs}(v_2)\ \wedge\ (H_2'', \theta, \overline{id_2'}) = \text{copy}(H_2')
\end{array}
$$

matching in both sides.

- (&unit)

$$\frac{}{\textbf{withBorrow}\ (\lambda x.x)\ t \equiv t}\ \text{EqBorrowUnit}$$

With typing derivation for the LHS:

$$\frac{\Gamma_1 \vdash t : *A \qquad \dfrac{\dfrac{}{0 \cdot \Gamma_2, x : \&_1 A \vdash x : \&_1 A}\ \text{VAR}}{0 \cdot \Gamma_2 \vdash (\lambda x.x) : \&_1 A \multimap \&_1 A}\ \text{ABS}}{\Gamma_1 + 0 \cdot \Gamma_2 \vdash \textbf{withBorrow}\ (\lambda x.x)\ t : *A}\ \text{WITH\&}$$

and typing derivation for the RHS:

$$\Gamma_1 + 0 \cdot \Gamma_2 \vdash t : *A$$

There are two possibilities depending on the reduction of t.

(1) If the reduction of t diverges, then the reduction on the LHS also diverges, since we must repeatedly apply the $\rightsquigarrow_{\text{WITH\&R}}$ rule. Hence, the equation is trivially satisfied as both sides diverge.

(2) Otherwise, t reduces to a value v. By the typing and the value lemma (Lemma C.2) then $\exists v'.\ v = *v'$.
Then we can reduce as follows on the LHS:

$$
\begin{array}{rcl}
 & & H \vdash \textbf{withBorrow}\ (\lambda x.x)\ t \\
 & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x.x)\ (*v') \\
\rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H' \vdash \textbf{unborrow}\ ([\&(*v')/x]x) \\
 & \rightsquigarrow * & H' \vdash \textbf{unborrow}\ (\&(*v')) \\
\rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H' \vdash *v'
\end{array}
$$

and as follows on the RHS:

$$\begin{array}{rl} & H \vdash t \\ \rightsquigarrow * & H' \vdash *v' \end{array}$$

- (&assoc)

$$\frac{}{\textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ t \equiv \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t)}\ \textsc{EqborrowAssoc}$$

If any of the terms f, g or t diverge, then the LHS and RHS both diverge, and so the equation is trivially satisfied.
Otherwise, all three terms must reduce to values v_1, v_2 and v_3.
By the typing and the value lemma (Lemma C.2) then $v_1 = \lambda x_1.t_1$, $v_2 = \lambda x_2.t_2$, and $\exists v_3'.\ v_3 = *v_3'$.
Then we can reduce as follows on the LHS:

$$\begin{array}{rll} & & H \vdash \textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ t \\ & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ (*v_3') \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H' \vdash \textbf{unborrow}\ ([\&(*v_3')/x](f\ (g\ x))) \\ & \rightsquigarrow * & H' \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ ((\lambda x_2.t_2)\ (\&(*v_3')))) \\ \rightsquigarrow_{\beta} & \rightsquigarrow & H', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ t_2)) \end{array}$$

If t_2 diverges, then again both sides diverge. Otherwise, t_2 must reduce to a value v_4, and by the value lemma $\exists v_4'.\ v_4 = \&(*v_4')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H'', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ (\&(*v_4')))) \\ \rightsquigarrow_{\beta} & \rightsquigarrow & H'', x_2 \mapsto_s (\&(*v_3')), x_1 \mapsto_{s'} (\&(*v_4')) \vdash \textbf{unborrow}\ (t_1) \end{array}$$

If t_1 diverges, then again both sides diverge. Otherwise, t_1 must reduce to a value v_5, and by the value lemma $\exists v_5'.\ v_5 = \&(*v_5')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H''', x_2 \mapsto_s (\&(*v_3')), x_1 \mapsto_{s'} (\&(*v_4')) \vdash \textbf{unborrow}\ (\&(*v_5')) \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H''', x_2 \mapsto_s (\&(*v_3')), x_1 \mapsto_{s'} (\&(*v_4')) \vdash *v_5' \end{array}$$

and on the RHS:

$$\begin{array}{rll} & & H \vdash \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t) \\ & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{withBorrow}\ (\lambda x_2.t_2)\ (*v_3')) \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H' \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ ([\&(*v_3')/x_2]t_2)) \\ & \sim & H', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ (t_2)) \end{array}$$

If t_2 diverges, then again both sides diverge. Otherwise, t_2 must reduce to a value v_4, and by the value lemma $\exists v_4'.\ v_4 = \&(*v_4')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H'', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ (\&(*v_4'))) \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H'', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (*v_4') \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H'', x_2 \mapsto_s (\&(*v_3')) \vdash \textbf{unborrow}\ ([\&(*v_4')/x_1]t_1) \\ & \sim & H'', x_2 \mapsto_s (\&(*v_3')), x_1 \mapsto_{s'} (\&(*v_4')) \vdash \textbf{unborrow}\ (t_1) \end{array}$$

If t_1 diverges, then again both sides diverge. Otherwise, t_1 must reduce to a value v_5, and by the value lemma $\exists v_5'.\ v_5 = \&(*v_5')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H'', x_2 \mapsto_s (\&(*v_3')), x_1 \mapsto_{s'} (\&(*v_4')) \vdash \textbf{unborrow}\ (\&(*v_5')) \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H'', x_2 \mapsto_s (\&(*v_3')) \vdash *v_5' \end{array}$$

□