Kindly Bent to Free Us

GABRIEL RADANNE, Inria, Paris
HANNES SAFFRICH, University of Freiburg, Germany
PETER THIEMANN, University of Freiburg, Germany

Systems programming often requires the manipulation of resources like file handles, network connections, or dynamically allocated memory. Programmers need to follow certain protocols to handle these resources correctly. Violating these protocols causes bugs ranging from type mismatches over data races to use-after-free errors and memory leaks. These bugs often lead to security vulnerabilities.

While statically typed programming languages guarantee type soundness and memory safety by design, most of them do not address issues arising from improper handling of resources. An important step towards handling resources is the adoption of linear and affine types that enforce single-threaded resource usage. However, the few languages supporting such types require heavy type annotations.

We present Affe, an extension of ML that manages linearity and affinity properties using kinds and constrained types. In addition Affe supports the exclusive and shared borrowing of affine resources, inspired by features of Rust. Moreover, Affe retains the defining features of the ML family: it is an impure, strict, functional expression language with complete principal type inference and type abstraction. Affe does not require any linearity annotations in expressions and supports common functional programming idioms.

1 INTRODUCTION

A large proportion of systems programming is focused on the proper handling of resources, like file handles, network connections, or dynamically allocated memory. Each of these resources comes with a protocol that prescribes the correct use of its API. For examples, a file handle appears as the result of opening a file. If it was opened for reading, then read operations will succeed, but write operations will fail. Once the handle is closed, it cannot be used for reading or writing, anymore. Dynamic allocation of memory is similar. An API call returns a pointer to a memory area, which can then be read and written to until the area is released by another API call.

In both cases, a resource is created in a certain state and a resource handle is returned to the program. Depending on this state, certain API calls can safely be applied to it. Finally, there is another API call to release the resource, which renders the handle invalid. Taken to the extreme, each API call changes the state so that a different set of API calls is enabled afterwards. Ignoring such life cycle protocols is a common source of errors.

Most type systems provide type soundness and memory safety, but neglect the protocol aspect. Systems that can support reasoning about protocols build on linear types [11] and/or uniqueness types [5]. A value of linear type is guaranteed to be consumed exactly once. That is, a file that has been opened must be closed and memory that has been allocated must be released. A value

Authors' addresses: Gabriel Radanne, Inria, Paris, radanne@informatik.uni-freiburg.de; Hannes Saffrich, University of Freiburg, Germany, saffrich@informatik.uni-freiburg.de; Peter Thiemann, University of Freiburg, Germany, thiemann@informatik.uni-freiburg.de.

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of unique type is guaranteed to have a single reference to it. Thus, memory can be reused on consumption of the value.

These systems work well if one is prepared to write programs functionally in resource-passing style. In this style, all operations in the resource's API take the resource as a parameter and return it in possibly modified state [1]. In typestate-oriented programming, they would also modify its type [2]. Functional session types represent a popular example [10, 16].

Explicit resource passing places a heavy burden on the programmer and complicates the program structure. For imperative APIs, resource-passing style is not an option at all. To this end, Boyland and Retert [7] proposed the notion of *borrowing* a resource. The idea is that a linear resource can be borrowed to a function call. The function can work with a borrow of the resource, but it cannot release the resource. Only the original owner of the resource has all rights to it and can release it.

The concepts of ownership and borrowing have grown popular over time and they form the foundation of the type system of the Rust language [17], which considers any memory-allocated data structure a resource. Rust supports two kinds of borrows, shared and exclusive ones. Exclusive borrows enable modification of the data structure whereas shared borrows only grant read access. At any given time, either a single exclusive borrow is active or any number of shared borrows can be active. Moreover, Rust makes sure that the lifetime of a borrow is properly contained in the lifetime of its lender.

The design of Rust is geared towards programmers with a low-level imperative programming background, like C or C++. Its management of lifetimes supports the manual way of memory management customary in these languages very well and makes it safe. However, programmers with a background in managed languages feel alienated from the lack of garbage collected data. They would prefer a setting where automatic memory management with garbage collection was the default, but where they could seemlessly switch to safe, manual resource management if that was required. As a concrete example, consider a functional programmer who wants to safely interact with a C library. Invoking a C function is easy via the existing foreign function interface, but managing the underlying resources like malloc'd storage is not: It cannot be left to the garbage collector, but proper release of the storage via calls to `free()` must be ensured by programming conventions.

Our work provides a safe solution to programmers in this situation. We propose an extended type system for ML-like languages that comes with linear and affine types, exclusive and shared borrows, but all that integrated with full principal type inference, garbage collected data, and automatic placement of borrowing regions. In our system, it is a type error to omit the call to release the storage given a suitably typed API for storage allocation.

The most closely related contenders in this design space are Linear Haskell [6], henceforth LH, Quill [20], and ALMS [36]. Compared to LH and Quill, the goals and means are similar as these systems also permit abstraction over the number of uses of values and retain type inference, but the details are different.

(1) Multiplicities in LH and Quill are either linear or unrestricted whereas we also distinguish affine values.
(2) In Affe and in Quill multiplicities are directly attached to the type of a value. For example, in Affe the function type $\alpha \xrightarrow{\mathtt{lin}} \beta$ denotes the type of a *single-use function* that can be called just once, whereas the multiplicities in LH choose between $\alpha \rightarrow \beta$ and $\alpha \multimap \beta$ where the latter is a function that promises to *use its argument exactly once*.
(3) Affe makes use of multiplicity contraints (like Quill) and kind subsumption (unlike Quill). Kind subsumption results in significantly simpler, more readable inferred types.

```
module File : sig
  type t: lin
  val fopen: path → t
  val write: &!t → string -aff-> unit
  val close: t → unit
end
```

(a) File API

```
1 let main () =
2   let h = File.fopen "foo" in
3   File.write &!h "Hello␣";
4   File.write &!h "world!";
5   File.close h
```

(b) File example

```
1 let main () =
2   let h = File.fopen "foo" in
3   {| File.write &!h "Hello␣" |};
4   {| File.write &!h "world!" |};
5   File.close h
```

(c) File example with regions

Fig. 1. Writing files

(4) Neither LH nor Quill have borrowing whereas Affe supports two flavors: affine (exclusive, mutable) and unrestricted (shared, immutable) borrows.

See Section 7 for further in-depth discussion of these and other related works.

1.1 First examples

As a first, well-known example we consider a simplified API for writing files shown in Fig. 1a. It introduces a linear abstract type `File.t`. A call like `File.fopen "foo"` returns a linear handle to a newly created file, which *must* be released later on with `File.close` as shown in Fig. 1b. Failing to do so is a static type error. To write to the file, we must take an exclusive borrow `&!h` of the handle and pass it to the `File.write` function. Exclusive borrows are affine: they must not be duplicated, but there is no requirement to use them. This affinity shows up in the annotation $\xrightarrow{\text{aff}}$ of the second arrow in the type of `File.write`: a partial application like `File.write &!h` captures the affine borrow and hence the resulting function is also affine. It would be an error to use the affine closure twice as in

```
let w = File.write &!h in w "Hello␣"; w "world!" (*type error*)
```

The remaining arrows in the API are unrestricted and we write $\rightarrow$ instead of the explicitly annotated $\xrightarrow{\text{un}}$.

Every borrow is restricted to a *region*, i.e., a lexically scoped program fragment from which the borrow must not escape. In Fig. 1b, there are two regions visualized in Fig. 1c, one consisting of Line 3 and another consisting of Line 4. Both are fully contained in the scope of the linear handle `h`, hence we can take one exclusive borrow `&!h` in each region. In both regions the borrow is consumed immediately by passing it to `File.write`. Regions are generally inferred by Affe.

This example demonstrates three features of our system:

(1) type and region inference without annotations in user code (Fig. 1b),
(2) types carry multiplicity annotations in the form of kinds,
(3) resource APIs can be written in direct style as linearity is a property of the type `File.t`.

Direct style means that there is a function like `fopen` that creates and returns a linear resource. In contrast, LH forces programmers to use resource-passing style because, in LH, linearity is a property of a function, rather than a property of a value that restricts the way that value can be handled (as in Affe). An LH API analogous to `File` might provide functions like

- `withFile : path → (handle ⊸ Unrestricted r) → r`, which creates a new file handle and takes a continuation that uses the `handle` linearly, but returns an unrestricted value[1],
- `writeFile : string → handle ⊸ handle`, which returns the transformed resource `handle`, and
- `closeFile : handle ⊸ unit`, which consumes the `handle` by closing the file.

[1]For technical reasons, LH requires the programmer to use a type like `Unrestricted` at this point.

In general, kinds can be polymorphic and constrained. Function application and composition are the archetypical examples for functions exploiting that feature.[2] For application, Affe infers the following type.

```
let app f x = f x
# app : (α -κ-> β) → (α -κ-> β)
```

The reading of the inferred type is straightforward. If `f` is a κ-restricted function, then so is `app f`. The multiplicities of α and β play no role. As usual in ML-like languages, we implicitly assume prenex quantification by $\forall\kappa\forall\alpha\forall\beta$. Internally, the type checker also quantifies over the kinds of α and β, but the full prefix $\forall\kappa\kappa_1\kappa_2\forall(\alpha : \kappa_1)\forall(\beta : \kappa_2)$ of the type of `app` is only revealed as much as necessary for understanding the type.

For `compose`, Affe infers this type.

```
let compose f g x = f (g x)
# compose : (κ ≤ κ₁) ⇒ (β -κ-> γ) → (α -κ₁-> β) -κ-> (α -κ₁-> γ)
```

Like in `app`, the multiplicities of the type variables α, β, γ do not matter. However, the multiplicity κ of `f` reappears on the second to last arrow because `compose f` is a closure that inherits `f`'s multiplicity. The multiplicities of `g` and `f` both influence the multiplicity of the last arrow, so we would expect its annotation to be the least upper bound $\kappa \sqcup \kappa_1$. Thanks to subsumption of multiplicities, it is sufficient to assume $\kappa \leq \kappa_1$ and `g`'s actual multiplicity gets subsumed to κ_1. This constraint simplification is part of our type inference algorithm. As before, printing the type scheme only mentions the non-trivial constraint $\kappa \leq \kappa_1$ and omits the prenex quantification over κ, κ_1 as well as the kinds of α, β, γ.

1.2 Contributions

- A polymorphic type system that encodes linearity and affinity with borrowing in lexical regions. Polymorphism covers types and kinds that express multiplicity constraints on the number of uses of a value. This type system is a conservative extension of systems for existing ML-like languages.
- Expressive type soundness theorem with respect to a big-step linearity-aware semantics.
- An extension of the HM(X) framework [22] for constrained type inference to equip the type system with full, principal type inference.
- Soundness proof of the inference algorithm.
- Automatic inference of regions for borrows.
- A prototype implementation of the type inference algorithm, including all constraint simplification and extended with algebraic datatypes and pattern matching, available at https://affe.netlify.com/.

As Affe is built on top of the HM(X) framework, which is a general framework for expressing constraint-based typing and type inference, the extension of our work with features like typeclasses, ad-hoc overloading, traits, etc is possible and orthogonal to the presentation in this paper. While the system is geared towards type inference, it is nonetheless compatible with type annotations and thereby amenable to extensions where type inference may no longer be possible.

2 LINEARITY, AFFINITY, AND BORROWS AT WORK

Affe is amenable to a functional resource-passing style common in functional encodings of session types (e.g., [25]; see also Appendix A.1 in the supplement) as well as other functional resource handling. But it really shines when manipulating mutable resources like buffers or connection pools

[2] Compared to Quill [20] the signatures of application and composition are simpler because Affe supports kind subsumption.

```
module Array : sig
  type (α : κ) t : lin
  val create : (α : un) ⇒ int × α → α t
  val free : (α : aff) ⇒ α t → unit
  val length : &(α t) → int
  val get : (α : un) ⇒ &(α t) × int → α
  val set : (α : aff) ⇒ &!(α t) × int × α → unit
  val map : (α → β) × α t → β t (* destructive *)
  val map_copy : (&α → β) × &(α t) → β t (* copying *)
end
```

Fig. 2. Linear arrays

using a mix of functional and imperative programming styles. To support this usage pattern of linearity, Affe relies on the notion of borrowing [7]. Our first example of linear arrays demonstrates simple borrowing and imperative programming; the second example is a Sudoko solver using a hybrid copy-on-write data structure; the third example demonstrates advanced uses of regions with iterators on linear values. Further examples are available in Appendix A.

2.1 Imperative programming with linear arrays

The API for mutable linear arrays (Fig. 2) aims to safely handle manual allocation and deallocation of arrays that may contain affine elements. A program would first use `create (n, v)` to create an array of size `n` initialized with value `v`. The value `v` must be unrestricted as it is duplicated to initialize all array elements. Using the `map` and `map_copy` functions we can transform the unrestricted elements into linear (affine) ones. The `map` function consumes its input array, whereas `map_copy` preserves the input. The implementation of `map` *might* perform an in-place update, but this is neither required nor guaranteed. The resulting array can be processed further using `map` or `map_copy` (or `set` if affine). The `get` function is only applicable if the element type is unrestricted as one element is duplicated. The `length` function is always applicable. To `free` an array the elements must be affine. An array with linear elements can only be freed by `map`ping an explicit destructor over its elements.

To manage the different kinds of accessing the array we distinguish between constructors, destructors, observers, and mutators. Constructors and destructors like `create` and `free` manipulate the whole array. The constructor `create` yields a linear resource which is consumed by `free`. During the lifetime of the array resource `a`, we can split off *shared borrows* `&a` that provide a read-only view or *exclusive borrows* `&!a` for read-write views. Observer functions such as `length` and `get` expect a shared borrow argument, mutator functions such a `set` expect an exclusive borrow.

Each borrow is tied to a region whose lifetime is properly contained in the lifetime of the resource. In a region, we can split off as many shared borrows of a resource as we like, but we can take only one exclusive borrow. In a subsidiary region, we can take shared borrows of any borrow or we can take an exclusive borrow of an exclusive borrow from an enclosing region. Borrows are confined to their regions. Inside the region, shared borrows are unrestricted (**un**) whereas exclusive borrows are affine (**aff**).

Using the API we can create an array of Fibonacci numbers in an imperative coding style:

```
let mk_fib_array n =
  let a = create (n, 1) in
  for i = 2 to n - 1 do
    let x = get (&a, i-1) + get (&a, i-2) in
    set (&!a, i, x)
  done;
  a
# mk_fib_array : int → int Array.t
```

After creation of the array, the presence of a borrow in the for loop prevents access to the “raw” resource inside the loop’s body. In particular, the resource cannot be freed through a borrow. Line 4 contains two shared borrows in the same expression which forms a region by itself. These borrows are split off the exclusive borrow used in Line 5 which belongs to the next enclosing region corresponding to the loop body. The whole array can be returned in Line 7 because the borrows are no longer in scope. More precisely, here is is an annotated excerpt with regions explicitly marked by braces `{| ... |}`:

```
3 for i = 2 to n - 1 do {|
4   let x = {| get (&a, i-1) + get (&a, i-2) |} in
5   set (&!a, i, x)
6 |} done;
```

One region consists of the header expression of the **`let`** in Line 4. It is contained in another region spanning the body of the **`for`** loop. Affe guarantees that borrows never escape the smallest enclosing region. It employs a system of *indexed kinds* like $\mathbf{aff}_r$ and $\mathbf{un}_r$ where r is a positive integer that corresponds to the lexical nesting depth of regions. For instance, the type of `&!a` in Line 5 has kind $\mathbf{aff}_1$ whereas the type of `&a` in Line 4 has kind $\mathbf{un}_2$ and the typing of the inner region is such that types with kind indexes greater than or equal to 2 cannot escape. In the example, borrows cannot escape because they are consumed immediately by `get` and `set`.

2.2 Solving sudokus with hybrid data-structures

Recently introduced persistent data structures permit transient mutations where non-linear uses lead to degraded performance [8] or to dynamic and static checks [28]. In particular, persistent Hash-Array-Mapped-Tries (HAMT) have been used with similar APIs in several non-pure functional languages (OCaml, Clojure, ...). Affine types help formalize the performance contract between the programmer and the library, while borrows avoid the need to thread state explicitly, as usually required by an API for immutable data types.

In this section, we present a safe API for persistent arrays that support both immutable and mutable features, and use it to implement a backtracking Sudoku solver. The solver maintains an array to represent the state of the game and uses backtracking when there are several choices to proceed. The use of backtracking suggests a persistent data structure for the array. However, only changes that correspond to a choice point need to use the persistence mechanism, others may be implemented as cheap in-place mutations.

Fig. 3 contains an API `HYBARRAY` that enables using mutable and immutable modifications to the board through affine types and borrows. The signature differs slightly from the `Array` signature. As our application requires the `get` function, the array elements must be unrestricted, but the structure itself remains linear so as to be implemented in terms of `Array`. The in-place mutation function `set_mut` with type $\&!(\alpha\ \mathtt{t}) \times \mathtt{int} \times \alpha \rightarrow \mathtt{unit}$ works on a exclusive borrow whereas the persistent `set` operation has type $\&(\alpha\ \mathtt{t}) \times \mathtt{int} \times \alpha \rightarrow \alpha\ \mathtt{t}$. It takes a shared borrow because it only reads from the argument array, but returns a fresh, modified structure. The code in Fig. 3 contains a very simple implementation of `HYBARRAY` that represents hybrid arrays as regular arrays and uses copy-on-write for persistent modifications. The function $\mathtt{mapi}: (\mathtt{int} \times \&\alpha \rightarrow \beta) \times \&(\alpha\ \mathtt{t}) \rightarrow \beta\ \mathtt{t}$ is a simple variation on `Array.map` where the mapping function also takes the position of the element.

Our implementation of a Sudoku solver (Fig. 4) performs modifications that correspond to choice points using `set`, which makes it trivial to come back to the previous version of the array, while other modifications use `set_mut`, which cannot be reverted.

The board is represented as a 2D-matrix (Line 1), where the `Matrix` type uses the same API as `CowArray` but with two indices. Each cell contains an integer set that represents admissible solutions so far. The Sudoku solver iterates over the cells and tries each possible solution (Line 17). When a

```
module type HYBARRAY = sig
  include ARRAY
  val set : &(α t) × int × α → α t
  val set_mut : &!(α t) × int × α → unit
end
```

```
module CowArray : HYBARRAY = struct
  include Array
  let set (a, i0, x0) =
    Array.mapi ((fun (i, x) → if i = i0 then x0 else x), a)
  let set_mut = Array.set
end
```

Fig. 3. Signature and Implementation of hybrid arrays

```
1  type board = IntSet.t Matrix.t
2
3  let propagate_line i0 j0 g n =
4    for j = j0+1 to 8 do
5      let cell = Matrix.get (&g, i0 , j) in
6      let cell' = IntSet.remove n cell in
7      Matrix.set_mut (&!g, i0, j, cell')
8    done
9
10 let propagate i j g n =
11   propagate_line i j &!g n;
12   propagate_column i j &!g n;
13   propagate_square i j &!g n
14 let rec solve i j g =
15   begin if is_solved &g then Matrix.print &g else
16     let (new_i, new_j) = next_pos (i,j) in
17     let try_solution n =
18       let new_g =
19         Matrix.set (&g, i, j, IntSet.singleton n) in
20       propagate i j &!new_g n;
21       if is_valid &new_g then solve new_i new_j new_g
22     in
23     let cell = Matrix.get (&g, i, j) in
24     IntSet.iter try_solution cell;
25   end;
26   free g
```

Fig. 4. Excerpt of the Sudoku solver

value is picked for the current cell, we create a choice point, change the cell with an immutable modification (Line 19), and propagate the changes with the `propagate` function. The `propagate` function uses direct mutation through an exclusive borrow of the matrix as it need not preserve the previous version of the board. The implementation of `propagate` is split into three parts for lines, columns, and square, which are all very similar to function `propagate_lines` (Line 3).

The typing ensures that the mutations do not compromise the state at the choice point, because they operate on a new state `new_g` created for one particular branch of the choice. As the `set` function only requires an unrestricted shared borrow, the closure `try_solution` remains unrestricted even though it captures the borrow `&g`. The price is that `try_solution` cannot escape from `&g`'s region. In this example, the inferred region corresponds to the **begin**/**end** scope. Hence, `try_solution` can be used in the iteration in Line 24. As `g` is linear we must free it outside of the region before returning (Line 26).

While presented for copy-on-write arrays, the API can easily be adapted to other persistent data structures with transient mutability such as Relaxed-Radix Balance Vectors (RRB) [28] or persistent HAMTs [4, 13] to provide a convenient programming style without compromising performance.

2.3 Iterators and regions

In the examples so far, regions do not appear in type signatures. But for certain programming idioms, we want to extend the scope of a region across a function boundary. For instance, how should we fold on an array of linear objects? A naive fold function would only work on unrestricted content:

```
val fold : (α : un) ⇒ (α → β → β) → α Array.t → β → β
```

If we want to work on linear and affine resources, we must make sure those resources are not leaked in the result. content. We obtain the following signature:

$$
\begin{array}{l}
\texttt{val fold :} \\
\quad (\beta{:}\kappa), (\kappa \le \mathbf{aff}_r) \Rightarrow \\
\quad (\&(\mathbf{aff}_{r+1}, \alpha) \to \beta \xrightarrow{\mathtt{aff}_{r+1}} \beta) \to \&(\kappa_1, \alpha\ \texttt{Array.t}) \to \beta \xrightarrow{\kappa_1} \beta
\end{array}
$$

The folded function receives a shared borrow of the element in the array. The typing of the callback ensures that this borrow is neither captured nor returned by the function. This encapsulation is implemented with a universally quantified *kind index variable* r. The signature prescribes the type $\texttt{\&(}\mathbf{aff}_{r+1}\texttt{,}\alpha\texttt{)}$ for the shared borrow of the resource with an affine kind at region nesting $r+1$. The return type of the callback is constrained to kind $\kappa \leq \mathbf{aff}_r$ so that the callback certainly cannot return the borrowed argument. The input of the fold is a shared borrow of the array, which ensures that we have the right to share borrows of the inner content and make multiple concurrent folds.

As an easy example, we fold over an array of files to compute the sum of their sizes:

```
let total_size = fold (fun f s → File.size f + s) &all_files 0
```

This approach is not sufficient if we want to mutate the array during iteration. To this end, we need to take an exclusive borrow of the structure to iterate on:

```
val iter_mut : &!(α Array.t) → (&!(aff_{r+1},α) -un-> unit) -aff-> unit
```

The callback is explicitly annotated with **un** because it will be called multiple times.

While the distinction between mutable and immutable iteration functions seems unfortunate, it is typical of programming with borrows and is also present in the Rust standard library. It enables the programmer to explicitly state how different iterations may be composed and optimized. It also enables different implementations such as using parallel iterations in the immutable case. Affe's region variables ensure that the content iterated on can never be leaked outside of the iteration function. This pattern is essential in many use cases of linearity such as pools of linear objects (see Appendix A.2).

3 THE AFFE LANGUAGE

Affe is a minimal ML-like language with let-polymorphism and abstract types. Its type system manages linearity and borrowing using kinds and (kind-) constrained types. For ease of presentation, we consider a simplified internal language.

- Pattern matching is demonstrated on pairs rather than algebraic datatypes.
- There are separate operators for borrowing and reborrowing (taking the borrow of a borrow); the surface language unifies these operators using ad-hoc polymorphism / typeclasses.
- Regions are explicit in the code and must be annotated using the algorithms presented in Section 3.2.
- Regions are identified using nesting levels instead of region variables.

The rest of this section formalizes Affe: the syntax (Section 3.1), the statics in terms of a region annotation pass (Section 3.2) and syntax-directed typing (Section 3.3), and a dynamics that is linearity- and resource-aware (Section 3.4).

3.1 Syntax

Figure 5 defines the syntax of Affe. Expressions are as usual in an ML-like language. The novel aspects are match specifications, regions, and borrows.

A borrow $\&^b x$ is always taken from a variable x. The *borrow annotation* b indicates whether the borrow is exclusive/affine (**A**) or shared/unrestricted (**U**). If x is already borrowed, then we need to use the reborrow expression $\&\&^b x$. A region $\{\!|e|\!\}^n_{\{x\mapsto b\}}$ is annotated with its nesting n, the variable x that may be borrowed in e, and the kind of borrow b. A match is indexed with a *match specification* ϕ that indicates whether the match operates on borrows ($\phi = \&^b$) or not ($\phi = id$). We consider four primitive operations to manipulate resources: `create`, `observe`, `update` and `destroy`. They serve a prototypes to demonstrate the typing and handling of resources. For a concrete type of resource, there are further arguments and perhaps different versions of the operations. But the

Expressions

$$
\begin{array}{lll}
e & ::= c \mid x \mid (e\ e') \mid \lambda x.e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e' & \\
& \mid (e, e') \mid \mathtt{match}_\phi\ x, y = e\ \mathtt{in}\ e' & \text{(Pairs)} \\
& \mid \{\!|e|\!\}^n_{\{x \mapsto b\}} & \text{(Region)} \\
& \mid \&^b x \mid \&\&^b x & \text{(Borrows)} \\
& \mid \mathtt{create} \mid \mathtt{observe} & \\
& \mid \mathtt{update} \mid \mathtt{destroy} & \text{(Resources)} \\
b & ::= \mathbf{U} \mid \mathbf{A} & \text{(Borrow specification)} \\
\phi & ::= \mathrm{id} \mid \&^b & \text{(Match Specification)}
\end{array}
$$

Types

$$
\begin{array}{lll}
\tau & ::= \alpha \mid \tau \times \tau' \mid \mathrm{T}\ \overline{\tau} & \text{(ML types)} \\
& \mid \tau \xrightarrow{\mathrm{k}} \tau' & \text{(Function types)} \\
& \mid \&^b(k, \tau) & \text{(Borrowed Type)}
\end{array}
$$

Kinds

$$
\begin{array}{llll}
k & ::= \kappa \mid Q_n & \forall n \in \mathbb{N} \cup \{\infty\} & \text{(Kinds)} \\
Q & ::= \mathbf{U} \mid \mathbf{A} \mid \mathbf{L} & & \text{(Quality)}
\end{array}
$$

Constrained type and kind schemes

$$
\begin{array}{lll}
C & ::= \overline{(k \leq k')} & \text{(Constraints)} \\
\sigma & ::= \forall \overline{\kappa} \forall \overline{(\alpha : k)}.(C \Rightarrow \tau) & \text{(Type scheme)} \\
\theta & ::= \forall \overline{\kappa}.(C \Rightarrow \overline{k} \to k) & \text{(Kind scheme)}
\end{array}
$$

Fig. 5. Syntax

typing and behavior of the operations is analogous to the prototype operations. Moreover, `observe` and `update` serve as eliminators for borrow types.

Many types are indexed with kinds. A kind k is either a kind variable κ or a constant (linear **L**, affine **A**, or unrestricted **U**) indexed by a nesting level $n \in \mathbb{N} \cup \{\infty\}$.

A type τ is either a type variable, a pair type, a function type indexed by a kind, a type application $\mathrm{T}\ \overline{\tau}$ of an abstract type constructor T, or a borrowed type $\&^b(k, \tau)$.

Type schemes σ add quantification over kind variables κ, kinded type variables $(\alpha : k)$, and constraints C to a type, where a constraint is a list of inequalities over kinds.

Abstract type constructors possess kind schemes θ which relate the kinds of the type constructors' arguments to the kind of the constructed type.

Generally, we write lists with an overbar and (sometimes) an index as in $\overline{\tau_i}$.

3.2 Automatic region annotation

In the surface language (Section 2) region annotations are optional. In the internal language, regions must be syntactically explicit and annotated with a nesting index and a scoped variable. This section defines a transformation $p \rightsquigarrow p'$ which automatically inserts region annotations in programs. The input p is a program with optional region annotations of the form $\{\!|e|\!\}$. The output p' is a program with explicit annotations of the form $\{\!|e|\!\}^n_{\{x \mapsto b\}}$ such that no borrow occurs outside a region. We give an informal presentation of our code transformation and defer the complete definition to Appendix B. This code transformation aims to find, for each borrow $\&^b x$, the biggest region satisfying the following rules:

(1) The region should contain at least $\&^b x$.
(2) The region must be contained in the scope of x.
(3) An exclusive borrow $\&^{\mathbf{A}} x$ should never share a region with any other borrow of x.
(4) The variable x cannot occur in the region of $\&^b x$.

The transformation starts from each borrow $\&^b x$ and grows its associated region until enlarging it would include the binding for x or lead to a conflicting use of x or a conflicting borrow for x. As an example, consider the following program:

$$\begin{array}{l}\lambda a.\ \texttt{let}\ x = (f\ \&^{\mathbf{U}} a)\ \texttt{in} \\ \quad g\ (\&^{\mathbf{A}} x); \\ \quad f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x)\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\lambda a.\{\!|\ \texttt{let}\ x = (f\ \&^{\mathbf{A}} a)\ \texttt{in} \\ \quad \{\!| g\ (\&^{\mathbf{A}} x) |\!\}^{2}_{\{x \mapsto \mathbf{A}\}}; \\ \quad \{\!| f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x) |\!\}^{2}_{\{x \mapsto \mathbf{U}\}} \\ |\!\}^{1}_{\{a \mapsto \mathbf{A}\}}\end{array}$$

As variable a only has one borrow, its region covers its whole lexical scope. Variable x has multiple conflicting borrows and requires more consideration. We place a first region around the exclusive borrow and its function application, and a second region around both shared borrows. This placement of region is optimal: making any region bigger would cause a borrowing error. Region indices are assigned after placement, so it is trivial to ensure well-nested regions where the inner regions have higher indices than the outer ones.

Programmers may also annotate regions explicitly. The transformation considers an annotation as an upper bound on the contained regions. In the following program, a manual annotation has been inserted to ensure no borrow enters the reference r:

$$\begin{array}{l}\texttt{let}\ r = \mathrm{ref}\ \ 0\ \texttt{in} \\ \lambda a.\ set\ r\ \{\!| g\ (\&^{\mathbf{U}} a) |\!\}; \\ \quad f\ (\&^{\mathbf{U}} a)\end{array} \rightsquigarrow \begin{array}{l}\texttt{let}\ r = \mathrm{ref}\ \ 0\ \texttt{in} \\ \lambda a.\ set\ r\ \{\!| g\ (\&^{\mathbf{U}} a) |\!\}^{1}_{\{a \mapsto \mathbf{U}\}}; \\ \quad \{\!| f\ (\&^{\mathbf{U}} a) |\!\}^{1}_{\{a \mapsto \mathbf{U}\}}\end{array}$$

The rules allow merging the two regions around the borrows $\&^{\mathbf{U}} a$. However the explicit annotation indicates that the region should stay around the closure passed as argument of *set*. This feature is useful to control captures by imperative APIs.

The code transformation is purely syntactic and must be used before typing. It only produces well-nested annotations: if $\{\!| \dots |\!\}^{n}_{b}$ is nested inside $\{\!| \dots |\!\}^{n'}_{b'}$, then $n > n'$. Furthermore, there is at most one region per borrow, and exactly one region per exclusive borrow. In the rest of this article, we assume that all terms have been annotated by this code transformation and respect these properties.

3.3 Typing

To avoid distraction, this section focuses on the essential and novel parts of the type system. A complete description is available in Appendix D. Here we only discuss the following judgments:

$C \mid \Gamma \vdash_{\mathrm{s}} e : \tau$ — Expression e has type τ in environment Γ under constraints C.
$C \mid \Gamma \vdash_{\mathrm{s}} \tau : k$ — Type τ has kind k in environment Γ under constraints C.
$D \vdash_{\mathrm{e}} C$ — Constraint D entails constraint C.

Kinds and constraints. Affe uses kinds and constrained types to indicate linear and affine types. A kind k is either a kind variable, κ, or a constant Q_n. The quality Q describes the use pattern of the type: unrestricted **U**, affine **A**, or linear **L**. The level $n \in \mathbb{N} \cup \{\infty\}$ describes the nested regions in which the value can be used. Level 0 refers to the top-level scope outside any region; we often elide it and write **A** for $\mathbf{A}_0$. Level ∞ refers to an empty region that is infinitely nested. For instance, the constraint $(\kappa \leq \mathbf{U}_\infty)$ indicates that κ must be unrestricted, but can be local to a region. Kinds form a lattice described in Fig. 6. Unrestricted values may be used where affine ones are expected and affine ones are less restricted than linear ones as reflected in Lat-UAL. Values usable at level n may be used at any more deeply nested level n' as defined in the Lat-Level axioms. Constraints are conjunctions of inequality constraints over this kind lattice, i.e., they specify upper or lower bounds for kind variables or relate two kind variables.

$$\textsc{Lat-UAL}\quad \mathbf{U} \le \mathbf{A} \le \mathbf{L}$$

$$\textsc{Lat-Level}\quad \frac{Q \le Q' \qquad n \le n'}{Q_n \le_{\mathcal{L}} Q'_{n'}}$$

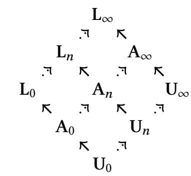

Fig. 6. Lattice ordering – $k \le_{\mathcal{L}} k'$

$$
\begin{array}{lll}
\Gamma ::= \cdot \mid \Gamma; B & & \text{(Environments)}\\
B ::= \emptyset & & \text{(Empty)}\\
\quad \mid (\alpha : \theta) & & \text{(Types)}\\
\quad \mid (x : \sigma) & & \text{(Variables)}\\
\quad \mid [x : \sigma]^n_b & & \text{(Suspended)}\\
\quad \mid (x \div \sigma)^k_b & & \text{(Borrows)}
\end{array}
$$

Fig. 7. Type environments

$$
\begin{array}{rcccccl}
(\sigma \le \mathbf{U}_\infty) \vdash_e & (x : \sigma) & = & (x : \sigma) & \ltimes & (x : \sigma) & \text{(Both)}\\
\cdot \vdash_e & (x \div \sigma)^k_{\mathrm{U}} & = & (x \div \sigma)^k_{\mathrm{U}} & \ltimes & (x \div \sigma)^k_{\mathrm{U}} & \text{(Borrow)}\\
\cdot \vdash_e & B_x & = & B_x & \ltimes & \emptyset & \text{(Left)}\\
\cdot \vdash_e & B_x & = & \emptyset & \ltimes & B_x & \text{(Right)}\\
\cdot \vdash_e & (x : \sigma) & = & [x : \sigma]^n_b & \ltimes & (x : \sigma) & \text{(Susp)}\\
\cdot \vdash_e & (x \div \sigma)^k_b & = & [x : \sigma]^n_{\mathrm{U}} & \ltimes & (x \div \sigma)^k_b & \text{(SuspB)}\\
\cdot \vdash_e & [x : \sigma]^{n'}_b & = & [x : \sigma]^n_{\mathrm{U}} & \ltimes & [x : \sigma]^{n'}_b & \text{(SuspS)}
\end{array}
$$

Fig. 8. Splitting rules for bindings – $C \vdash_e B = B_l \ltimes B_r$

$$\textsc{Instance}\quad \frac{\sigma = \forall \overline{\kappa_i} \forall \overline{(\alpha_j : k_j)}.\, C \Rightarrow \tau \qquad \psi = [\overline{\kappa_i \mapsto k_i}, \overline{\alpha_j \mapsto \tau_j}]}{\psi(C), \psi(\tau) = \mathrm{Inst}(\Gamma, \sigma)}$$

$$\textsc{Var}\quad \frac{(x : \sigma) \in \Gamma \qquad C_x, \tau_x = \mathrm{Inst}(\Gamma, \sigma) \qquad C \vdash_{\mathrm{e}} C_x \wedge (\Gamma \backslash \{x\} \le \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} x : \tau_x}$$

$$\textsc{Abs}\quad \frac{C \mid \Gamma; (x : \tau_2) \vdash_{\mathrm{s}} e : \tau_1 \qquad C \vdash_{\mathrm{e}} (\Gamma \le k)}{C \mid \Gamma \vdash_{\mathrm{s}} \lambda x.e : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$$

$$\textsc{App}\quad \frac{C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_2 \xrightarrow{\mathrm{k}} \tau_1 \qquad C \mid \Gamma_2 \vdash_{\mathrm{s}} e_2 : \tau'_2 \qquad C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \vdash_{\mathrm{e}} (\tau'_2 \le \tau_2)}{C \mid \Gamma \vdash_{\mathrm{s}} (e_1\ e_2) : \tau_1}$$

$$\textsc{Region}\quad \frac{[x : \tau_x]^n_b \in \Gamma \qquad C \vdash_e \Gamma \rightsquigarrow^x_n \Gamma' \qquad C \mid \Gamma' \vdash_{\mathrm{s}} e : \tau \qquad C \vdash_{\mathrm{e}} (\tau \le \mathbf{L}_{n-1})}{C \mid \Gamma \vdash_{\mathrm{s}} \{\!\!| e |\!\!\}^n_{\{x \mapsto b\}} : \tau}$$

$$\textsc{Borrow}\quad \frac{(x \div \sigma)^k_b \in \Gamma \qquad C_x, \tau_x = \mathrm{Inst}(\Gamma, \sigma) \qquad C \vdash_{\mathrm{e}} C_x \wedge (\Gamma \backslash \{x\} \le \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \&^b x : \&^b(k, \tau)}$$

$$\textsc{Reborrow}\quad \frac{C \mid \Gamma \vdash_{\mathrm{s}} x : \&^b(k, \tau)}{C \mid \Gamma \vdash_{\mathrm{s}} \&\&^b x : \&^b(k, \tau)}$$

$$\textsc{BorrowBinding}\quad \frac{C \vdash_{\mathrm{e}} (b_n \le k) \wedge (k \le b_\infty) \qquad b \in \{\mathbf{U}, \mathbf{A}\}}{C \vdash_e [x : \sigma]^n_b \rightsquigarrow_n (x \div \sigma)^k_b}$$

Fig. 9. Selected typing rules ($C \mid \Gamma \vdash_{\mathrm{s}} e : \tau$) and borrowing rules ($C \vdash_e \Gamma \rightsquigarrow^x_n \Gamma'$)

Environments and bindings. Affe controls the use of variables by supporting new modes of binding in type environments Γ, as defined in Fig. 7. Environments contain standard bindings of type variables to kind schemes, $(\alpha : \theta)$, value bindings $(x : \sigma)$, but also suspended and borrow bindings. A suspended binding, $[x : \sigma]^n_b$, indicates that x is earmarked for a borrowing use in a nested region marked with x but cannot be used directly. A borrow binding, $(x \div \sigma)^k_b$, replaces such a suspended binding on entry to the x-region. It indicates that the borrow $\&^b x$ can be used directly. Kind k restricts the lifetime of the borrow to the region (see rules Region and BorrowBinding in Fig. 9 and the upcoming discussion of these rules).

Constraints on an environment control substructural properties by restricting the types of variables. The constraint $(\Gamma \leq k)$ stands for the conjunction of $(\sigma \leq k)$, for all $(x : \sigma)$ in Γ, which in turn means that $k' \leq k$ where k' is the kind of the variable's type scheme σ. Borrow bindings follow the same rules, but suspended bindings are forbidden in an environment Γ constrained like that. This intuitive explanation is sufficient to understand the Var and Abs rules shown in Fig. 9.

Rule Var looks up the type scheme of the variable x in the environment Γ and instantiates it using the substitution ψ. The rule also checks that the other bindings in Γ can be safely discarded by imposing the constraint $(\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)$. It enforces that all remaining bindings (except x) are affine or unrestricted and can therefore be discarded.

Rule Abs ensures the kind annotation on the arrow type $(\tau_2 \xrightarrow{\mathrm{k}} \tau_1)$ reflects the restrictions on captured variables via the constraint $(\Gamma \leq k)$. If, for instance, any binding in Γ is affine, it gives rise to the constraint $(\mathbf{A}_n \leq k)$ and the arrow kind is at least affine at nesting level n. Capturing a borrow is perfectly fine: the kind of the borrow is also a lower bound of the arrow kind k which restricts the closure to the region of the borrow. Capturing a suspended binding is forbidden.

Copying and Splitting. The App typing rule in Fig. 9 demonstrates how Affe deals with duplication and dropping of values. The splitting $C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2$ in the rule decomposes the type environment Γ in two parts, Γ_1 and Γ_2, which are used to typecheck the components of the application.

Fig. 8 shows the action of splitting rules on single bindings. If x's type is unrestricted, rule Both indicates that we can duplicate it. Similarly, unrestricted borrows can be duplicated with rule Borrow. Left and Right rules are always applicable and move a binding either to the left or right environment. The rules Susp, SuspB and SuspS split off suspended bindings to the left while conserving access to the binding on the right. A suspended binding can later be turned into a borrow inside a region. Splitting of suspended bindings is asymmetric. It must follow the order of execution from left to right, which means that a resource can be used first as a borrow on the left and then later as a full resource on the right. The Susp rule works with a full resource, the SuspB with a borrow and SuspS with a suspended binding.

Splitting applies whenever an expression has multiple subexpressions: function applications, let bindings and pairs. In the expression $\texttt{let}\ a = \mathrm{create}\ 8\ x\ \texttt{in}\ f\ \{\!|\mathrm{length}\ \&^{\mathrm{U}} a|\!\}_{\{a \mapsto \mathrm{U}\}}\ a$, the rule Susp splits off a borrow from the resource a to use it in the left argument. As usual, a borrow cannot be active in the same scope as its resource. The *region* around its use ensures that the borrow in the left argument does not escape, which brings us to the next topic.

Regions. Borrowing is crucial to support an imperative programming style. To guarantee the validity of a borrow, its lifetime must be properly contained in its ancestor's lifetime. Affe ensures proper nesting of lifetimes by using regions. The expression $\{\!|e|\!\}^n_{\{x \mapsto b\}}$ indicates a region at nesting level n in which a b-borrow can be taken of x.

The typing for a region (rule Region in Fig. 9) replaces suspended bindings by borrow bindings (rule BorrowBinding), typechecks the body of the region, and ensures that the borrow does not leak outside. This last check is done with indices that correspond to the nesting level of the region. The kind k of the borrow is indexed with the level n corresponding to its region, thanks to the constraint $(b_n \leq k)$. The constraint $(\tau \leq \mathbf{L}_{n-1})$ ensures that the return type of the region must live at some enclosing, lower level.

As an example, consider the expression $\{\!|f(\&^{\mathrm{U}} c)|\!\}^n_{\{x \mapsto \mathrm{U}\}}$ where c is a linear channel in an environment Γ. The first step is to check that $[c : \mathrm{channel}]_{\mathrm{U}}$ is in Γ. When entering the region, rule Region imposes $C \vdash_e \Gamma \rightsquigarrow^x_n \Gamma'$, which defines Γ' corresponding to Γ where the suspended binding is replaced by the borrow binding $(c \div \mathrm{channel})^k_{\mathrm{U}}$. To constrain the borrow to this region we impose the constraint $C \vdash_{\mathrm{e}} (\mathbf{U}_n \leq k) \wedge (k \leq \mathbf{U}_\infty)$, which affirms that the borrow is unrestricted, but can

Elaborated expressions

$$
\begin{array}{rll}
e ::= & x \mid x[\overline{k};\overline{\tau}] \mid \lambda^k x.e \mid (e\ e')_{sp} & \\
 & \mid (e,e')^k_{sp} \mid \mathtt{match}_\phi\ x,y =_{sp} e \mathtt{\ in\ } e' & \text{(Pairs)} \\
 & \mid \mathtt{let}\ x =_{sp} e \mathtt{\ in\ } e' & \text{(Mono let)} \\
 & \mid \mathtt{letfun}\ (x:\sigma) =_{sp} \lambda^k y.e \mathtt{\ in\ } e' & \text{(Poly let)} \\
 & \mid \{\!|e|\!\}^n_{\{x \mapsto b\}} & \text{(Region)} \\
 & \mid \&^b x \mid \&\&^b x & \text{(Borrows)} \\
 & \mid \mathtt{create} \mid \mathtt{observe} & \\
 & \mid \mathtt{update} \mid \mathtt{destroy} & \text{(Resources)}
\end{array}
$$

Splittings

$$sp : (C \vdash_e \Gamma = \Gamma_l \ltimes \Gamma_r)$$

Storables

$$
\begin{array}{rll}
w ::= & \mathrm{STPOLY}(\gamma, \overline{\kappa}, C, k, x, e) & \text{(Poly Closures)} \\
 & \mid \mathrm{STCLOS}(\gamma, k, x, e) & \text{(Closures)} \\
 & \mid \mathrm{STPAIR}(k, r, r') & \text{(Pairs)} \\
 & \mid \mathrm{STRSRC}(r) & \text{(Resources)} \\
 & \mid \bullet & \text{(Freed Resource)}
\end{array}
$$

Environment

$$
\begin{array}{rll}
\rho ::= & \overline{\mathbf{U}}\,\overline{\mathbf{A}}\,\ell & \text{(Locations)} \\
\pi ::= & \{\} \mid \pi + \rho & \text{(Permissions)} \\
r ::= & \rho \mid c & \text{(Results)} \\
\gamma ::= & \cdot \mid \gamma(x \mapsto r) & \text{(Enviroments)} \\
\delta ::= & \cdot \mid \delta(\ell \mapsto w) & \text{(Stores)}
\end{array}
$$

Fig. 10. Syntax of internal language

only be used in nesting levels n and higher. Rule Region also imposes the constraint $(\tau \leq \mathbf{L}_{n-1})$, which prevents the borrow, having kind k of level $\geq n$, from escaping the region's body of type τ.

Pattern matching. Elimination of pairs is done using a matching construct $(\mathtt{match}_\phi\ x, x' = e_1 \mathtt{\ in\ } e_2)$. This construct is mostly standard, except it can operate both on a normal pair and a borrow of a pair. The intuition is as follows: A syntactic marker ϕ indicates if it applies to a pair $(\phi = \mathrm{id})$ or a borrow $(\phi = \&^b)$. If $\phi = \mathrm{id}$, the typing simplifies to the usual elimination of a pair. Otherwise, e_1 is expected to be a borrow of type $\&^b(k, \tau_1 \times \tau_1')$ and the variables x and x' have type $\&^b(k, \tau_1)$ and $\&^b(k, \tau_1')$, respectively. Thus, the borrow of a pair is considered as a pair of borrows of its components.

Resource management. To demonstrate how Affe deals with resources, we introduce an abstract type $\mathrm{R}\ \tau$ whose content of type τ must be unrestricted $(\mathbf{U}_0)$ and which is equipped with the four operations introduced in Section 3.1:

- $\mathtt{create}$: $\forall \kappa_\alpha(\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \alpha \to \mathrm{R}\ \alpha$
- $\mathtt{observe}$: $\forall \kappa \kappa_\alpha(\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \&^{\mathbf{U}}(\kappa, \mathrm{R}\ \alpha) \to \alpha$
- $\mathtt{update}$: $\forall \kappa \kappa_\alpha(\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \&^{\mathbf{A}}(\kappa, \mathrm{R}\ \alpha) \to \alpha \xrightarrow{\mathbf{A}} \mathtt{Unit}$
- $\mathtt{destroy}$: $\forall \kappa_\alpha(\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \mathrm{R}\ \alpha \to \mathtt{Unit}$

3.4 Semantics

It is straightforward to give a standard semantics for Affe, but such a semantics would not be very informative. In this section, we give a big-step semantics that performs explicit bookkeeping of the number of times a value is used and of the mode in which a reference to a resource is used (e.g., borrowed or not). This bookkeeping is based on a set of permissions that regulate the currently allowed mode of access to resources and closures. It enables us to state and prove a highly informative type soundness result (see Section 5) with expressive invariants that ensure proper resource usage.

The dynamics of Affe is given in big-step functional style [3, 24, 30]. A function eval manipulates the semantic objects defined in Fig. 10. The semantics is defined in terms of *elaborated expressions* e with kind, constraint, and splitting annotations inserted by the typechecker. A splitting sp is evidence of the splitting relation for type environments used in the typing rules.

Let-polymorphism in the surface language gives rise to elaborated `letfun` expressions annotated with a type scheme σ and a kind k indicating their usage restriction (linear, affine, etc) relative to the variables and constraints of σ. Their use gives rise to explicit instantiation of the kind and type variables. Pairs come with a kind tag k indicating the usage restriction.

Addresses ρ are composed of a raw location ℓ, which is just a pointer into a store, and a stack of modifiers that indicates the borrows that have been taken from the original object. From the raw location, we may take affine borrows and reborrows. Once we have taken an unrestricted borrow (from a raw location or a borrowed one), then we can take further unrestricted borrows from it, but no more affine ones.

A permission π is a set of addresses that may be accessed during evaluation. A well-formed permission contains at most one address for each raw location.

Non-trivial results are boxed in the semantics. So, a result r is either an address or a primitive constant (e.g., a number).

A value environment γ maps variables to results.

A storable w describes the content of a location in the store. There are five kinds of storables. A *poly closure* represents a polymorphic function. It consists of an environment and the components of an elaborated abstraction. A *closure* represents a monomorphic function in the usual way. A *resource* contains a result and the *hole* • fills a released location.

A store δ is a partial map from raw locations to storables. The function salloc : store → storable → (loc * store) is such that salloc delta w allocates an unused location in delta and fills it with w. It returns the location and the extended store.

The evaluation function is indexed by a step count i so that each invocation is guaranteed to terminate either with an error, a timeout, or a result. Its return type is a monad α sem which combines error reporting and timeout:

```
type α sem = Error of string | TimeOut | Ok of α
val eval: store→perm→venv→int→exp→(store *perm *result) sem
```

Function eval evaluates the given expression in the context of an initial store, a permission to use addresses in the store, a value environment, and a step count. If successful, it returns the final store, the remaining permissions, and the actual result.

We give some excerpts of the definition of eval in Fig. 11 and leave the full definition for Appendix F. The definition uses OCaml syntax with extensive pretty printing. The pervasive **let**∗ operator acts as monadic bind for the sem monad. The operator **let**∗? : bool → (unit → α sem) → α sem converts a boolean argument into success or failure in the monad.

```
let (let*?) : bool → (unit → β sem) → β sem =
  fun b f → if b then f () else Error ("test␣failed")
```

The function header of eval checks whether time is up and otherwise proceeds processing the expression.

The Varinst case corresponds to instantiation. It obtains the variable's value, checks that it is a location, checks the permission (the **let**∗? clause), obtains the storable w at that location, and checks that it is a poly closure (STPOLY). Next, it updates the permission: if the poly closure is unrestricted, then the location remains in the permission set, otherwise it is removed. Finally, we allocate a new monomorphic closure, add it to the permissions, and return the pointer as the result along with the updated store and permissions.

The App case implements (elaborated) function application. We first apply the splitting sp to gamma and evaluate subterm e_1 with its part of the environment and the decremented timer i'. The result must be a location that we are permitted to use. Moreover, there must be a monomorphic STCLOS stored at that location. The permission to further use this closure remains in force only if

```
let rec eval
  (δ:store) (π:perm) (γ:venv) i e
  : (store × perm × result) sem =
  if i=0 then TimeOut else
  let i' = i − 1 in
  match e with
  | Region (e, n, x, τ_x, b) →
     let+ ρ = γ(x) in
     let* ρ' = b.ρ in
     let* π' = reach ρ τ_x δ in
     let* π'' = b.π' in
     let γ' = γ(x ↦ρ') in
     let π = (π ∪ π'') \ π' in
     let* (δ1, π1, r1) = eval δ π γ' i' e in
     let π1 = (π1 \ π'') ∪ π' in
     Ok (δ1, π1, r1)

  | Borrow (b, x) →
     let+ ρ = γ(x) in
     let*? () = ρ ? b && ρ ∈ π in
     Ok (δ, π, ρ)

| Varinst (x, k̄) →
  let* rx = γ(x) in
  let* ℓ = getloc rx in
  let*? () =  ℓ ∈ π in
  let* w = δ(ℓ) in
  let* (γ', κ̄', C', k', x', e') = getstpoly w in
  let π' =
    if C'{k̄\>κ̄'} =e [(k' ≤ U)]{k̄\>κ̄'}
    then π
    else π −  ℓ
  in
  let w = STCLOS (γ', k'{k̄\>κ̄'}, x', e'{k̄\>κ̄'}) in
  let (ℓ', δ') = salloc δ w in
  Ok (δ', π' +  ℓ',  ℓ')

| App (e1, e2, sp) →
  let (γ1, γ2) = vsplit γ sp in
  let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
  let* ℓ1 = getloc r1 in
  let*? () =  ℓ1 ∈ π1 in
  let* w = δ1(ℓ1) in
  let* (γ', k', x', e') = getstclos w in
  let π1' = if k' ≤ U then π1 else π1 −  ℓ1 in
  let* δ1' = δ1(ℓ1) ← (if k' ≤ U then w else •) in
  let* (δ2, π2, r2) = eval δ1' π1' γ2 i' e2 in
  let* (δ3, π3, r3) = eval δ2 π2 γ'(x'↦ r2) i' e' in
  Ok (δ3, π3, r3)
```

Fig. 11. Big-step interpretation

the closure is unrestricted. Finally, we evaluate the argument, then the function body, and return its result.

The Region case implements a region. It obtains the address for x, the suspended binding, and extends it with the intended borrow b. This extension may fail if we try to take an affine borrow of an unrestricted borrow. Next, we rebind x to the borrow's address, extend the permission accordingly, and execute the region's body. Finally, we withdraw the permission and return the result.

The Borrow case obtains the address for x, checks that it is a borrow of the correct mode b and whether it is permitted to use it. It just returns the address.

4 INFERENCE

An important contribution of Affe is its principal type inference. Our type inference algorithm is based on the HM(X) framework [22], a Hindley-Milner type system for a language with constrained types where constraints are expressed in an arbitrary theory X. If X has certain properties, then HM(X) guarantees principal type inference. We apply HM(X) to a concrete constraint language which we name $C_{\mathcal{L}}$. We adapt and extend HM(X)'s rules to support kind inference, track linearity, and handle borrows and regions. Finally, we formulate constraint solving and simplification algorithms for $C_{\mathcal{L}}$.

4.1 Preliminaries

In the context of inference, it is critical to know which elements are input and output of inference judgments. In the following, when presenting a new judgment, we write input parameters in **bold blue**. The remaining parameters are output parameters.

Usage Environments. To determine if a variable is used in an affine manner, we track its uses and the associated kinds. In the expression $f\ x\ x$, x is used twice. If x is of type τ, which is of kind k, we add the constraint $(k \leq \mathbf{U})$. To infer such constraints, our inference judgment not only takes an environment as parameter but also returns a *usage environment*, denoted Σ, which summarizes usages of variables and borrows. Usage environments are defined like normal environments. In Section 3.3, we use relations to split environments and to transform suspended bindings into borrows inside a region. These relations take a constraint parameter which validates the transformations. In the context of inference, we define new judgments which *infer* the constraints.

- $C \Lleftarrow \Sigma = \boldsymbol{\Sigma_1} \ltimes \boldsymbol{\Sigma_2}$. Given two usage environments $\boldsymbol{\Sigma_1}$ and $\boldsymbol{\Sigma_2}$, we return Σ, the merged environment, and C, a set of constraints that must be respected.
- $C \Lleftarrow \Sigma \rightsquigarrow^{\boldsymbol{x}}_{\boldsymbol{n}} \boldsymbol{\Sigma'}$. Given a usage environment $\boldsymbol{\Sigma'}$, a nesting level $\boldsymbol{n}$, and a variable name $\boldsymbol{x}$, we return Σ where the borrow binding of x in Σ', if it exists, is replaced by a suspended binding. We also return the constraints C.

Both relations are total and non-ambiguous in term of their input (i.e., functions), and use the rules presented in Section 3.3. The relations used for syntax-directed typing can trivially be defined in terms of these new relations by using constraint entailment. All relations are fully described in Appendix D.2.

Constraint Normalization. The HM(X) framework assumes the existence of a function "normalize" which takes a constraint $\boldsymbol{C}$ and a substitution $\boldsymbol{\psi}$ and returns a simplified constraint C' and an updated substitution ψ'. Normalization returns a normal form such that ψ' is a most general unifier. For now, we simply assume the existence of such a function for our constraint system and defer details to Section 4.3.

4.2 Type Inference

We write $\Sigma \mid (C, \psi) \mid \boldsymbol{\Gamma} \vdash_{\mathrm{w}} \boldsymbol{e} : \tau$ when $\boldsymbol{e}$ has type τ in $\boldsymbol{\Gamma}$ under the constraints C and unifier ψ with a usage environment Σ. $\boldsymbol{\Gamma}$ and $\boldsymbol{e}$ are the input parameters of our inference algorithm. Unlike in the syntax-directed version, Γ contains only regular and type bindings. Suspended and borrow bindings can only be present in Σ. We revisit some of the syntax-directed rules presented in Section 3.3 to highlight the novelties of our inference algorithm and the differences with the syntax-directed system in Fig. 12. The complete type inference rules are shown in Appendix E.

Environments and Bindings. In the syntax-directed system, the Var rule ensure that linear variables are not discarded at the *leaves*. In the inference algorithm, we operate in the opposite direction: we collect data from the leaves and enforce linearity at *binders*. This policy is reflected in the $\textsc{Var}_I$ and $\textsc{Abs}_I$ rules. Typing for variables is very similar to traditional Hindley-Milner type inference. To keep track of linearity, we record that x was used with the scheme σ by returning a usage environment $\Sigma = \{(x : \sigma)\}$. This usage environment is in turn used at each binder to enforce proper usage of linear variable via the Weak property as shown for lambda expressions in the $\textsc{Abs}_I$ rule. First, we typecheck the body of the lambda and obtain a usage environment Σ_x. As in the syntax-directed type system, we introduce the constraint $(\Sigma \backslash \{x\} \leq \kappa)$ which properly accounts for captures in the body of the lambda expression. We then introduce the constraint $\mathrm{Weak}_{(x:\sigma)}(\Sigma)$, which fails if we try to abandon a linear variable. The Weak constraint is introduced at each binding

$$\textsc{Var}_I\quad\frac{\begin{array}{c}(x:\sigma)\in\Gamma \qquad \sigma = \forall\kappa_i\forall(\alpha_j:k_j).C_x \Rightarrow \tau \\ \overline{\kappa'_i},\overline{\alpha'_j}\ \text{fresh} \qquad \psi' = [\kappa_i\mapsto\kappa'_i, \alpha_j\mapsto\alpha'_j] \\ (C,\psi) = \text{normalize}(C_x,\psi')\end{array}}{(x:\sigma)\,|\,(C,\psi|_{\text{fv}(\Gamma)})\,|\,\Gamma\vdash_{\text{w}} x : \psi\tau}$$

$$\textsc{Abs}_I\quad\frac{\begin{array}{c}\alpha,\kappa\ \text{fresh} \qquad \Sigma_x\,|\,(C',\psi')\,|\,\Gamma;(x:\alpha)\vdash_{\text{w}} e:\tau \\ \Sigma = \Sigma_x\backslash\{x\} \qquad D = C'\wedge(\Sigma\leq\kappa)\wedge\text{Weak}_{(x:\alpha)}(\Sigma_x) \\ (C,\psi) = \text{normalize}(D,\psi')\end{array}}{\Sigma\,|\,(C,\psi\backslash\{\alpha,\kappa\})\,|\,\Gamma\vdash_{\text{w}}\lambda x.e : \psi(\alpha)\xrightarrow{\psi(\kappa)}\tau}$$

$$\textsc{Region}_I\quad\frac{\begin{array}{c}\Sigma'\,|\,(C',\psi')\,|\,\Gamma\vdash_{\text{w}} e:\tau \qquad (C_\tau,\psi_\tau)\,|\,\Gamma\vdash_{\text{w}}\tau:k_\tau \\ C_r \Lleftarrow \Sigma \rightsquigarrow^x_n \Sigma' \qquad D = C'\wedge C_\tau\wedge(k_\tau\leq\mathbf{L}_{n-1})\wedge C_r \\ (C,\psi) = \text{normalize}(D,\psi'\sqcup\psi_\tau)\end{array}}{\Sigma\,|\,(C,\psi)\,|\,\Gamma\vdash_{\text{w}}\{\!|e|\!\}^n_{\{x\mapsto b\}}:\tau}$$

$$\textsc{App}_I\quad\frac{\begin{array}{c}\alpha,\kappa\ \text{fresh} \qquad \Sigma_1\,|\,(C_1,\psi_1)\,|\,\Gamma\vdash_{\text{w}} e_1:\tau_1 \qquad \Sigma_2\,|\,(C_2,\psi_2)\,|\,\Gamma\vdash_{\text{w}} e_2:\tau_2 \\ C_s\Lleftarrow\Sigma = \Sigma_1\ltimes\Sigma_2 \qquad D = C_1\wedge C_2\wedge(\tau_1\leq\tau_2\xrightarrow{\kappa}\alpha)\wedge C_s \\ \psi' = \psi_1\sqcup\psi_2 \qquad (C,\psi) = \text{normalize}(D,\psi')\end{array}}{\Sigma\,|\,(C,\psi)\,|\,\Gamma\vdash_{\text{w}}(e_1\ e_2):\psi(\alpha)}$$

$$\text{Weak}_{(x:\sigma)}(\Sigma) = \text{if } (x\in\Sigma) \text{ then True else } (\sigma\leq\mathbf{A}_\infty)$$

Fig. 12. Selected inference rules – $\Sigma\,|\,(C,\psi)\,|\,\Gamma\vdash_{\text{w}} e:\tau$

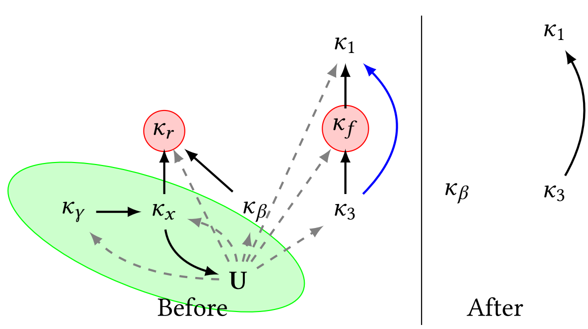

Fig. 13. Graph representing the example constraints

construct. Finally, we normalize constraints to ensure that the inference algorithm always return the simplest possible constraints and unifiers.

Splitting and Regions. Inference versions of the App and Region rules are similar to the original ones, but now *return* the usage environment Σ. As such, we use the "inference" version of the relations on the environment, $C\Lleftarrow\Sigma = \Sigma_1\ltimes\Sigma_2$ and $C\Lleftarrow\Sigma\rightsquigarrow^x_n\Sigma'$, which returns the necessary constraints. We then collect all constraints and normalize them.

4.3 Constraint Solving

For concision, we demonstrate the constraint solving algorithm with an example. The complete constraint system is defined in Appendix C.

Consider the expression $\lambda f.\lambda x.((f\ x), x)$. The inference algorithm yields the following constraints:

$$\Gamma = (\alpha_f : \kappa_f)(\alpha_x : \kappa_x)\dots$$
$$C = (\alpha_f \leq \gamma \xrightarrow{\kappa_1} \beta) \land (\gamma \leq \alpha_x) \land (\beta * \alpha_x \leq \alpha_r) \land (\kappa_x \leq \mathbf{U})$$

The first step of the algorithm uses Herbrand unification to obtain a type skeleton.

$$\alpha_r = (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

In addition, we obtain the following kind constraints:

$$(\kappa_x \leq \mathbf{U}) \land (\kappa_\gamma \leq \kappa_x) \land (\kappa_x \leq \kappa_r) \land (\kappa_\beta \leq \kappa_r) \land (\kappa_3 \leq \kappa_f) \land (\kappa_f \leq \kappa_1)$$

We translate these constraints into a relation whose graph is shown in Fig. 13. The algorithm then proceeds as follow:

- From the constraints above, we deduce the graph shown with plain arrows on the left of Fig. 13.
- We add all the dashed arrows by saturating lattice inequalities. For clarity, we only show **U**.
- We identify the connected component circled in green. We deduce $\kappa_\gamma = \kappa_x = \mathbf{U}$.
- We take the transitive closure, which adds the arrow in blue from κ_3 to κ_1.
- We remove the remaining nodes not present in the type skeleton (colored in red): κ_r and κ_f.
- We clean up the graph (transitive reduction, remove unneeded constants, ...), and obtain the graph shown on the right. We deduce $\kappa_3 \leq \kappa_1$.

The final constraint is thus

$$\kappa_\gamma = \kappa_x = \mathbf{U} \land \kappa_3 \leq \kappa_1$$

If we were to generalize, we would obtain the type scheme:

$$\forall \kappa_\beta \kappa_1 \kappa_2 \kappa_3 (\gamma : \mathbf{U})(\beta : \kappa_\beta).\ (\kappa_3 \leq \kappa_1) \Rightarrow (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

We can further simplify this type by exploiting variance. As κ_1 and κ_2 are only used in covariant position, they can be replaced by their lower bounds, κ_3 and **U**. By removing the unused quantifiers, we obtain a much simplified equivalent type:

$$\forall \kappa (\gamma : \mathbf{U}).(\gamma \xrightarrow{\kappa} \beta) \rightarrow \gamma \xrightarrow{\kappa} \beta * \gamma$$

Our algorithm respect the properties of HM(X), computes principal normal forms and simplifies constraints significantly. All the simplification mechanisms presented here, including the variance-based one, are complete. It is also possible to add "best-effort" simplification rules which help reduce the size of inferred signatures even further [32].

4.4 Soundness and Principality

The extended inference algorithm still satisfies the soundness and completeness properties of HM(X). The first theorem states that our inference algorithm is sound with respect to the syntax-directed type system.

Theorem 4.1 (Soundness of inference). *Given a type environment Γ containing only value bindings, $\Gamma|_\tau$ containing only type bindings, and a term e:*
if $\Sigma \mid (C, \psi) \mid \Gamma; \Gamma_\tau \vdash_{\mathrm{w}} e : \tau$
then $C \mid \psi(\Sigma; \Gamma_\tau) \vdash_{\mathrm{s}} e : \tau$, $\psi C = C$ *and* $\psi\tau = \tau$

The syntax-directed derivation holds with the usage environment Σ instead of the originally provided environment Γ. Indeed, Γ does not contain suspended and borrow bindings, since those are discovered on the fly and recorded in Σ. The type binding, however, are directly taken from the syntax-directed derivation.

The second theorem states that our algorithm is complete: for any given syntax-directed typing derivation, our inference algorithm can find a derivation that gives a type at least as general.

Definition 4.2 (Instance relation). Given a constraint C and two schemes $\sigma = \forall\overline{\alpha}.D \Rightarrow \tau$ and $\sigma' = \forall\overline{\alpha}'.D' \Rightarrow \tau'$. Then $C \vdash_e \sigma \leq \sigma'$ iff $C \vdash_e D[\alpha \rightarrow \tau'']$ and $C \wedge D' \vdash_e (\tau[\alpha \rightarrow \tau''] \leq \tau')$

Definition 4.3 (Flattened Environment). A flattened environment, written as $\Downarrow\Gamma$, is the environment where all the binders are replaced by normal ones. More formally:

$$\Downarrow\Gamma = \left\{(x : \tau) \in \Gamma \mid \vee(\&^b x : \&^b(k, \tau)) \in \Gamma \vee [x : \tau]^n_b \in \Gamma\right\} \cup \{(\alpha : k) \mid (\alpha : k) \in \Gamma\}$$

Theorem 4.4 (Principality). *Let* True $\mid \Gamma \vdash_s e : \sigma$ *a closed typing judgment. Then* $\Sigma \mid (C, \psi) \mid \Downarrow \Gamma \vdash_w e : \tau$ *such that:*

$$(\mathrm{True}, \sigma_o) = gen(C, \psi\Gamma, \tau) \qquad\qquad \vdash_e \sigma_o \leq \sigma$$

5 METATHEORY

There are several connections between the type system and the operational semantics, which we state as a single type soundness theorem. The theorem relies on several standard notions like store typing $\vdash \delta : \Delta$ and agreement of the results in the value environment with the type environment $\Delta \vdash \gamma : \Gamma$ that we define formally in Appendix G where we also present selected cases of the proofs. The non-standard part is the handling of permissions. With $\mathrm{getloc}(\pi)$ we extract the underlying raw locations from the permissions as in $\mathrm{getloc}(\overline{\mathbf{U}}\,\overline{\mathbf{A}}\,\ell) = \ell$ and with $\mathrm{reach}_\delta(\gamma)$ we transitively trace the addresses reachable from γ in store δ. We write $\Delta \leq \Delta'$ and $\delta \leq \delta'$ for extending the domain of the store type and of the store, respectively. The permission set contains the set of addresses that can be used during evaluation. It is managed by the region expression as well as by creation and use of resources as shown in Section 3.4. We distinguish several parts of the value environment γ that correspond to the different kinds of bindings in the type environment: $\gamma^{\mathbf{L}}$ for active entries of direct references to linear resources, closures, etc; $\gamma^{\mathbf{A}}$ for affine borrows or resources; $\gamma^{\mathbf{U}}$ for unrestricted values including unrestricted borrows; and $\gamma_\#$ for suspended entries. The judgment $\Delta \vdash \gamma : \Gamma$ is defined in terms of this structure. We treat $\mathrm{reach}_\delta(\gamma)$ as a multiset to properly discuss linearity and affinity. We use the notation $M(x)$ for the number of times x occurs in multiset M.

Theorem 5.1 (Type Soundness). *Suppose that*

(A1) $C \mid \Gamma \vdash_s e : \tau$
(A2) $\Delta \vdash \gamma : \Gamma$
(A3) $\vdash \delta : \Delta$
(A4) π *is wellformed and* $\mathrm{getloc}(\pi) \subseteq \mathrm{dom}(\delta) \setminus \delta^{-1}(\bullet)$
(A5) $\mathrm{reach}_0(\gamma) \subseteq \pi$, $\mathrm{reach}_\delta(\gamma) \subseteq \downarrow\pi$.
(A6) $\mathrm{getloc}(\gamma^{\mathbf{L}})$, $\mathrm{getloc}(\gamma^{\mathbf{A}})$, $\mathrm{getloc}(\gamma^{\mathbf{U}})$, *and* $\mathrm{getloc}(\gamma_\#)$ *are all disjoint*
(A7) Incoming Resources:
  (a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_\delta(\gamma))$, $\delta(\ell) \neq \bullet$.
  (b) $\forall \ell \in \Theta = \mathrm{getloc}(\mathrm{reach}_\delta(\gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_\#))$, $\Theta(\ell) = 1$.

For all $i \in \mathbb{N}$, *if* R' = eval δ π γ i e *and* $R' \neq \mathrm{TimeOut}$, *then* $\exists\, \delta', \pi', r', \Delta'$ *such that*

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$
(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
(R3) $\Delta' \vdash r' : \tau$
(R4) π' *is wellformed and* $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.

(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ *it must be that*
- $\delta'(\ell) = \delta(\ell)$ *and*
- *for any* ρ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, *it must be that* $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ *and* $\rho \in \pi'$.

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, *it must be that* $\ell \in \mathrm{dom}(\delta)$. *If* $\rho \neq \ell$, *then* $\delta'(\ell) \neq \bullet$. *If* $\rho \in \mathrm{reach}_{\delta'}(\gamma_\#^{\mathbf{A}})$, *then* $\rho \in \pi'$.

(R9) Resources: Let $\Theta = \mathrm{reach}_{\delta}(\gamma^{\mathbf{L}})$. *Let* $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ *it must be that* $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, *and if* $\delta(\ell)$ *is a resource, then* $\delta'(\ell) = \bullet$.

(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.

The proof of the theorem is by functional induction on the evaluation judgment, which is indexed by the strictly decreasing counter i.

The assumptions A1-A3 and results R1-R3 state the standard soundness properties for lambda calculi with references.

The rest of the statement accounts for the substructural properties and borrowing in the presence of explicit resource management. Incoming resources are always active (i.e., not freed). Linear and affine resources as well as suspended affine borrows have exactly one pointer in the environment. The Frame condition states that only store locations reachable from the current environment can change and that all permissions outside the reachable locations remain the same. Unrestricted values, resources, and borrows do not change their underlying resource and do not spend their permission. Affine borrows and resources may or may not spend their permission. Borrows are not freed, but resources may be freed. Incoming suspended borrows have no permission attached to them and their permission has been retracted on exit of their region. A linear resource is always freed. Outgoing permissions are either inherited from the caller or they refer to newly created values.

6 EXTENSIONS

6.1 Algebraic Datatypes

Algebraic data types are a staple of functional programming and fit nicely in our paradigm. Indeed, it is sufficient to ensure that the kinds of the constructor arguments are less than or equal to the kind of the datatype. Hence, it is forbidden to include affine elements in an unrestricted datatype, whereas the elements in a linear list may be linear or unrestricted. Here is the definition of a datatype for linear lists.

```
type (α : κ) llist : lin = Nil | Cons of (α × α llist)
```

This extension is implemented in our prototype.

6.2 Ad-hoc Polymorphism and Borrows

In our formalization, we use two operators, $\&^b x$ and $\&\&^b x$ to distinguish between borrows and borrows of borrows. Such a distinction is inconvenient for programming. Using a typeclass-like mechanism, we can replace these operators by a single overloaded operator, $\&^b x$, which expects x to be `Borrowable` and would then desugar to the more precise operators. A similar solution is used in Rust through the `Borrow` and `Defer` traits. This approach also enables method calls on objects without explicit borrows, such as `foo.len()` where len expects a shared borrow.

Ad-hoc polymorphism fits demonstrably in the HM(X) framework of constrained types and preserves all properties of our language such as principal type inference. Its soundness is orthogonal to linear types and has been explored in the literature [23].

6.3 Branching constructs

Our formalization of Affe does not contain branching constructs. As a consequence, subexpressions are always executed sequentially. This is of great importance for the splitting relation introduced in Section 3.3, as it must coincide with the order of evaluation. To account for borrows and linear values in different branches, we need a new (symmetric) join relation. This relation immediately follows from the semantics of borrows and linearity shown so far, and allow to easily define branching constructs such as if-then-else and pattern matching. This extension is implemented in our prototype.

6.4 Non-Lexical Regions

A recent important addition to Rust is the notion of non-lexical lifetimes. With this feature code is acceptable even if borrowing does not respect lexical scoping as in this example:

```
let a = &x in
f a;
g (&!x)
```

Such code patterns are dynamically safe since a is not used after the second line. Non-lexical lifetimes handle this by removing expressions that do not mention a from its region; in this example, the expression on the last line. In Affe, regions are purely lexical and marked by the expression $\{\!|e|\!\}^n_b$. During inference, we introduce kind constraints to prevent escaping from a region.

To add support for non-lexical lifetimes, we could replace the lexical region by an annotation on each expression indicating which borrows are live in this expression. When exploring a subexpression, we would compare the annotations, and automatically apply the Region rule when they differ. This approach is equivalent to inlining the Region rule in all the other rules.

Applied to the program above, only the first two lines would be annotated to be “in the region associated with &x”, but not the last line. Thanks to these annotations, when type checking the sequence we would check that the borrow does not escape the left-hand side (i.e., the second line).

7 RELATED WORK

7.1 Substructural type-systems in functional languages

Many systems propose combinations of functional programming and linear types in a practical setting. The goal of Affe is to combine key ingredients from these proposals while still preserving complete type inference. Many of the following languages support linear or affine types, but rarely both. In many cases, it is easy to adapt a system to support both, as Affe does. None of the following languages support borrows.

System F° [19] extends System F with kinds to distinguish between linear and unrestricted types. The authors provide a linearity-aware semantics with a soundness proof. Unlike Affe, System F° does not allow quantification over kinds which limits its expressivity. For instance, it does not admit a most general type for function composition. Being based on System F, it does not admit principal type inference.

Quill [20] is a Haskell-like language with linear types. Quill does not expose a kind language, but uses the framework of qualified types to govern linearity annotations on arrows. Its type inference algorithm is proven sound and complete. Affe infers type signatures for all Quill examples, but often with simpler types because Quill does not support subkinding. Quill comes with a linearity-aware semantics and soundness proof. Quill does not support borrows.

Alms [36] is an ML-like language with rich, kind-based affine types and ML modules, similar to Affe. Alms examples often rely on existential types to track the identity of objects. For instance `Array.create : int →α→∃β. (α, β)array` where β uniquely identifies the array. Due to the reliance on existentials, Alms does not support complete type inference. Furthermore, Alms does not support borrows and often relies on explicit capability passing. In our experience, Affe's limited support for existential types through regions is sufficient to express many of Alms' examples and leads to a more convenient programming style for imperative code. Alms kind structure features unions, intersections and dependent kinds while Affe uses constrained types. We believe most of Alms' kind signatures can be expressed equivalently in our system: for instance the pair type constructor has kind $\Pi\alpha\Pi\beta.\langle\alpha\rangle \sqcup \langle\beta\rangle$ (where α and β are types and Π is the dependent function) in Alms compared to $\kappa \rightarrow \kappa \rightarrow \kappa$ in Affe thanks to subkinding. Finally, Alms provides excellent support for abstraction through modules by allowing to keep some type unrestricted inside a module, but exposing it as affine. Affe supports such programming style thanks to subsumption.

The goal of Linear Haskell [6] (LH) is to retrofit linear types to Haskell. Unlike the previously discussed approaches, LH relies on "linear arrows", written $\multimap$ as in linear logic, which are functions that *use* their argument exactly once. This design is easy to retrofit on top of an existing compiler such as GHC, but has proven quite controversial[3]. Most relevant to Affe:

- LH does not admit subtyping for arrows and requires η-expansion to pass unrestricted functions in linear contexts. This approach is acceptable in a non-strict language such as Haskell but changes the semantics in a strict setting.
- While the LH paper specifies a full type system along with a linearity-aware soundness proof, there is neither formal description of the type inference algorithm nor a proof of the properties of inference. Subsequent work [18] formalizes the inference for rank 1 qualified-types. However, there is an implementation of the inference as part of GHC.
- LH promotes a continuation-passing style with functions such as `withFile : path →(file ⊸Unrestricted r)⊸r` to ensure linear use of resources. This style leads to problems with placing the annotation on, e.g., the IO monad. Affe follows System F°, Quill, and Alms, all of which support resource handling in direct style, where types themselves are described as affine or linear. (Of course, continuation-passing style is also supported.) We expect that the direct approach eases modular reasoning about linearity. In particular, using abstraction through modules, programmers only need to consider the module implementation to ensure that linear resources are properly handled.

Mezzo [27] is an ML-like language with a rich capability system which is able to encode numerous properties akin to separation logic [29]. Mezzo explores the boundaries of the design space of type systems for resources. Hence, it is more expressive than Affe, but much harder to use. The Mezzo typechecker relies on explicit annotations and it is not known whether type inference for Mezzo is possible.

Munch-Maccagnoni [21] presents an extension of OCaml for resource management in the style of C++'s RAII and Rust's lifetimes. This system assumes the existence of a linear type system and develops the associated compilation and runtime infrastructure. We believe our approach is complementary and aim to combine them in the future.

[3]See the in-depth discussion attached to the GHC proposal for LH on GitHub: https://github.com/ghc-proposals/ghc-proposals/pull/111#issuecomment-403349707.

7.2 Other substructural type-systems

Affe uses borrows and regions which were initially developed in the context of linear and affine typing for imperative and object-oriented programming [7, 12].

Rust [17] is the first mainstream language to popularize the idea of borrowing and ownership for safe low-level programming. Affe is inspired by Rust's borrowing system and transfers some of its ideas to a functional setting with type inference, garbage collection, and an ML-like module system. Rust's lifetime system is more explicit and more expressive than Affe, but Rust does not provide type inference and only provides *partial* lifetime inference. Recently, Weiss et al. [38] formalized Rust's ownership discipline, including non-lexical lifetimes. Another important difference is that Rust allows to precisely specify the memory layout of objects and to pass arguments by value. These features are crucial for the efficiency goals of Rust. In Affe, we assume that all arguments are passed by reference and that a garbage collector is available. This forgoes numerous issues regarding interior mutability and algebraic data types. In particular, it allows us to easily nest mutable references inside objects, regardless whether they are linear or unrestricted.

Vault [9] and Plaid [2] leverage typestate and capabilities to express rich properties in objects and protocols. These systems are designed for either low-level or object-oriented programming and do not immediately lend themselves to a more functional style. While these systems are much more powerful than Affe's, they require programmer annotations and do not support inference. It would be interesting to extend Affe with limited forms of typestate as a local, opt-in feature to provide more expressivity at the cost of inference.

7.3 Type-system features

Affe relies on constrained types to introduce the kind inequalities required for linear types. HM(X) [22] allows us to use constrained types in an ML-like language with complete type inference. HM(X) has been shown to be compatible with subtyping, bounded quantification and existentials [31], GADTs [33], and there exists a syntactic soundness proof [34]. These results make us confident that the system developed in Affe could be applied to larger and more complex languages such as OCaml and the full range of features based on ad-hoc polymorphism.

Affe's subtyping discipline is similar to structural subtyping, where the only subtyping (or here, subkinding) is at the leaves. Such a discipline is known to be friendly to inference and has been used in many contexts, including OCaml, and has been combined with constraints [22, 37]. It also admits classical simplification rules [26, 32] which we partially use in our constraint solving algorithm. Affe's novelty is a kind language sufficiently simple to make all simplification rules complete, which allows us to keep type signatures simple.

8 CONCLUSIONS

Affe is an ML-like language extended with sound handling of linear and affine resources. Its main novel feature is the combination of full type inference and a practically useful notion of shared and exclusive borrowing of linear and affine resources. Although the inferred types are much richer internally than plain ML types, most of that complexity can be hidden from user-level programmers. On the other hand, programmers of libraries dealing with resources have sufficient expressiveness at their fingertips to express many resource management schemes.

The main restriction of the current system is that the lifetime of borrows is determined by lexical scoping. Overcoming this restriction is subject of future work and will probably require extending the type system by some notion of effect, which is currently discussed in the OCaml community. Moreover, other systems rely on existential types for extra expressiveness. We chose not to include existentials to preserve complete type inference, but our design can be extended in this direction.

Finally, our matching construct is very simplistic. Our implementation supports full algebraic data types and we believe it can be further extended to support manipulating borrows of data-structures and internal mutability.

REFERENCES

[1] Peter Achten and Marinus J. Plasmeijer. 1995. The Ins and Outs of Clean I/O. *J. Funct. Program.* 5, 1 (1995), 81–110. https://doi.org/10.1017/S0956796800001258

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1015–1022. https://doi.org/10.1145/1639950.1640073

[3] Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs With Definitional Interpreters. In *POPL*. ACM, 666–679.

[4] Phil Bagwell. 2001. Ideal Hash Trees.

[5] Erik Barendsen and Sjaak Smetsers. 1995. Uniqueness Type Inference. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings (Lecture Notes in Computer Science)*, Manuel V. Hermenegildo and S. Doaitse Swierstra (Eds.), Vol. 982. Springer, 189–206. https://doi.org/10.1007/BFb0026821

[6] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *PACMPL* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

[7] John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adoption. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 283–295. https://doi.org/10.1145/1040305.1040329

[8] Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A Persistent Union-Find Data Structure. In *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, Claudio V. Russo and Derek Dreyer (Eds.). ACM, 37–46. https://doi.org/10.1145/1292535.1292541

[9] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *PLDI*. ACM, 59–69.

[10] Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

[11] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[12] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. https://doi.org/10.1145/512529.512563

[13] Rich Hickey. 2017. clojure/PersistentHashMap.java. https://github.com/richhickey/clojure/blob/master/src/jvm/clojure/lang/PersistentHashMap.java

[14] Kohei Honda. 1993. Types for Dyadic Interaction. In *Proceedings of 4th International Conference on Concurrency Theory (LNCS)*, Eike Best (Ed.). Springer Verlag, 509–523.

[15] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science)*, Chris Hankin (Ed.), Vol. 1381. Springer, 122–138. https://doi.org/10.1007/BFb0053567

[16] Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioral Types: From Theory to Tools*, Simon Gay and António Ravara (Eds.). River Publishers.

[17] Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

[18] Kazutaka Matsuda. 2019. A Modular Inference of Linear Types for Multiplicity-Annotated Arrows. *CoRR* abs/1911.00268 (2019). arXiv:1911.00268 http://arxiv.org/abs/1911.00268

[19] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F°. In *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, Andrew Kennedy and Nick Benton (Eds.). ACM, 77–88. https://doi.org/10.1145/1708016.1708027

[20] J. Garrett Morris. 2016. The Best of Both Worlds: Linear Functional Programming Without Compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 448–461. https://doi.org/10.1145/2951913.2951925

[21] Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. *CoRR* abs/1803.02796 (2018). arXiv:1803.02796 http://arxiv.org/abs/1803.02796

[22] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *TAPOS* 5, 1 (1999), 35–55.

[23] Martin Odersky, Philip Wadler, and Martin Wehr. 1995. A Second Look at Overloading. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 135–146. https://doi.org/10.1145/224164.224195

[24] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 589–615.

[25] Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

[26] François Pottier and Vincent Simonet. 2002. Information Flow Inference for ML. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 319–330. https://doi.org/10.1145/503272.503302

[27] Jonathan Protzenko. 2014. *Mezzo: a typed language for safe effectful concurrent programs. (Mezzo: un langage typé pour programmer de manière concurrent et sure en présence d'effets).* Ph.D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01086106

[28] Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-Vectors in a Systems Language. *PACMPL* 1, ICFP (2017), 16:1–16:28. https://doi.org/10.1145/3110260

[29] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[30] Jeremy Siek. 2013. Type Safety in Three Easy Lemmas. http://siek.blogspot.de/2013/05/type-safety-in-three-easy-lemmas.html.

[31] Vincent Simonet. 2003. An extension of HM(X) with bounded existential and universal data-types. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 39–50. https://doi.org/10.1145/944705.944710

[32] Vincent Simonet. 2003. Type Inference with Structural Subtyping: A Faithful Formalization of an Efficient Constraint Solver. In *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings (Lecture Notes in Computer Science)*, Atsushi Ohori (Ed.), Vol. 2895. Springer, 283–302. https://doi.org/10.1007/978-3-540-40018-9_19

[33] Vincent Simonet and François Pottier. 2007. A Constraint-Based Approach to Guarded Algebraic Data Types. *ACM Trans. Program. Lang. Syst.* 29, 1 (2007), 1. https://doi.org/10.1145/1180475.1180476

[34] Christian Skalka and François Pottier. 2002. Syntactic Type Soundness for HM(X). *Electr. Notes Theor. Comput. Sci.* 75 (2002), 61–74. https://doi.org/10.1016/S1571-0661(04)80779-5

[35] Martin Sulzmann. 1997. *Proofs of soundness and completeness of type inference for HM (X).* Technical Report. Research Report YALEU/DCS/RR-1102, Yale University, Department of Computer Science.

[36] Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436

[37] Valery Trifonov and Scott F. Smith. 1996. Subtyping Constrained Types. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot and David A. Schmidt (Eds.), Vol. 1145. Springer, 349–365. https://doi.org/10.1007/3-540-61739-6_52

[38] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

A FURTHER EXAMPLES

A.1 A session on linearity

Session typing [14, 15] is a type discipline for checking protocols statically. A session type is ascribed to a communication channel and describes a sequence of interactions. For instance, the type `int!int!int?end` specifies a protocol where the program must send two integers, receive an integer, and then close the channel. In this context, channels must be used linearly, because every use of a channel "consumes" one interaction and changes the type of the channel. That is, after sending two integers on the channel, the remaining channel has type `int?end`. Here are some standard type operators for session types $\mathcal{S}$:

$\tau!\mathcal{S}$	Send a value of type τ then continue with $\mathcal{S}$.
$\tau?\mathcal{S}$	Receive a value of type τ then continue with $\mathcal{S}$.
$\mathcal{S} \oplus \mathcal{S}'$	Internal choice between protocols $\mathcal{S}$ and $\mathcal{S}'$.
$\mathcal{S} \,\&\, \mathcal{S}'$	Offer a choice between protocols $\mathcal{S}$ and $\mathcal{S}'$.

Padovani [25] has shown how to encode this style of session typing in ML-like languages, but his implementation downgrades linearity to a run-time check for affinity. Building on that encoding we can provide a safe API in Affe that statically enforces linear handling of channels:

```
1 type 𝒮 st : lin
2 val receive: (α ? 𝒮) st → α × 𝒮 st
3 val send : (α ! 𝒮) st → α ─lin→ 𝒮 st
4 val create : unit → 𝒮 st × (dual 𝒮) st
5 val close : end st → unit
```

Line 1 introduces a parameterized abstract type `st` which is linear as indicated by its kind `lin`. Its low-level implementation would wrap a handle for a socket, for example. The `receive` operation in Line 2 takes a channel that is ready to receive a value of type α and returns a pair of the value a the channel at its residual type $\mathcal{S}$. It does not matter whether α is restricted to be linear, in fact `receive` is polymorphic in the kind of α. This kind polymorphism is the default if no constraints are specified. The `send` operation takes a linear channel and returns a single-use function that takes a value of type α suitable for sending and returns the channel with updated type. The `create` operation returns a pair of channel endpoints. They follow dual communication protocols, where the `dual` operator swaps sending and receiving operations. Finally, `close` closes the channel.

In Fig. 14 we show how to use these primitives to implement client and server for an addition service. No linearity annotations are needed in the code, as all linearity properties can be inferred from the linearity of the `st` type.

The inferred type of the server, `add_service`, is `(int ! int ! int ? end)st →unit`. The client operates by sending two messages and receiving the result. This code is polymorphic in both argument and return types, so it could be used with any binary operator. Moreover, the `op_client` function can be

```
let add_service ep =
  let x, ep = receive ep in
  let y, ep = receive ep in
  let ep = send ep (x + y) in
  close ep
# add_service : (int ! int ! int ? end) st → unit
```

(a) Addition server

```
let op_client ep x y =
  let ep = send ep x in
  let ep = send ep y in
  let result, ep = receive ep in
  close ep;
  result
# op_client :
   (α₁ ? α₂ ? β ! end) st → α₁ ─lin→ α₂ ─lin→ β
```

(b) Binary operators client

Fig. 14. Corresponding session type programs in Affe

partially applied to a channel. Since the closure returned by such a partial application captures the channel, it can only be used once. This restriction is reflected by the arrow of kind **lin**, $\xrightarrow{\text{lin}}$, which is the type of a single-use function. The general form of arrow types in Affe is $\xrightarrow{k}$, where k is a kind that restricts the number of uses of the function. For convenience, we shorten $\xrightarrow{\text{un}}$ to $\rightarrow$. Affe infers the single-use property of the arrows without any user annotation. In fact, the only difference between the code presented here and Padovani's examples [25] is the kind annotation on the type definition of `st`.

To run client and server, we can create a channel and apply `add_service` to one end and `op_client` to the other. Failure to consume either channel endpoints (`a` or `b`) would result in a type error.

```
let main () =
  let (a, b) = create () in
  fork add_service a;
  op_client b 1 2
# main : unit → int
```

A.2 Pool of linear resources

We present an interface and implementation of a pool of linear resources where the extended scope of the region enforces proper use of the resources.

Fig. 15a contains the interface of the `Pool` module. A pool is parameterized by its content. The kind of the pool depends on the content: linear content implies a linear pool while unrestricted content yields an unrestricted pool. The functions `Pool.create` and `Pool.consume` build/destroy a pool given creators/destructors for the elements of the pool. The function `Pool.use` is the workhorse of the API, which borrows a resource from the pool to a callback. It takes a shared borrow of a pool (to enable concurrent access) and a callback function. The callback receives a exclusive borrow of an arbitrary resource from the pool. The typing of the callback ensures that this borrow is neither captured nor returned by the function.

This encapsulation is implemented with a universally quantified *kind index variable* r. The signature prescribes the type $\texttt{\&!}(\textbf{aff}_{r+1},\alpha_1)$ for the exclusive borrow of the resource with an affine kind at region nesting $r+1$. The return type of the callback is constrained to kind $\kappa_2 \leq \textbf{aff}_r$ so that the callback certainly cannot return the borrowed argument. In a specific use of `Pool.use`, the index r gets unified with the current nesting level of regions so that the region for the callback effectively gets "inserted" into the lexical nesting at the callsite. Fig. 15b shows a simple example using the `Pool` module.

The implementation in Fig. 15c represents a bag of resources using a concurrent queue with atomic add and remove operations. The implementation of the `Pool.create` and `Pool.consume` functions is straightforward. The function `Pool.use` first draws an element from the pool (or creates a fresh element), passes it down to the callback function `f`, and returns it to the pool afterwards. For clarity, we explicitly delimit the region in Line 11 to ensure that the return value of `f &!o` does not capture `&!o`. In practice, the type checker inserts this region automatically.

B AUTOMATIC REGION ANNOTATION

We now define our automatic region annotation which is presented in Section 3.2. First, we extend the region annotation to $\{\!|E|\!\}^n_S$ where S is a map from variables to borrow indicator b. This annotation, defined below, is equivalent to nested region annotations for each individual variable.

$$\{\!|e|\!\}^n_{\{x\mapsto b\};S} = \{\!|\{\!|e|\!\}^n_S|\!\}^n_{\{x\mapsto b\}} \qquad\qquad \{\!|e|\!\}^n_\emptyset = e$$

Figure 16 define a rewriting relation $e \rightsquigarrow e'$ which indicates that an optionally annotated term e

```
type (α:κ) pool : κ
create : (unit → α) → α pool
consume : (α → unit) → α pool → unit
use : (α₁:κ₁),(α₂:κ₂),(κ₂ ≤ aff_r) ⇒
  &(α₁ pool) → (&!(aff_{r+1},α₁) -lin→ α₂) -κ₁→ α₂
```

(a) Signature

```
(*Using the pool in queries.*)
let create_user pool name =
  Pool.use &pool (fun connection →
    Db.insert "users" [("name", name)] connection)

let uri = "postgresql://localhost:5432"
let main users =
  (*Create a database connection pool.*)
  let pool = Pool.create (fun _ → Db.connect uri) in
  List.parallel_iter (create_user &pool) users;
  Pool.consume (fun c → Db.close c)
```

(b) Example of use

```
type (α:κ) pool : κ =
  { spawn: unit → α; queue: α CQueue.t }
let create spawn =
  { spawn ; queue = CQueue.create () }
let consume f c = CQueue.iter f c.queue
let use { spawn ; queue } f =
  let o = match CQueue.pop &queue with
    | Some x → x
    | None () → spawn ()
  in
  let r = {| f &!o |} in
  CQueue.push o &queue;
  r
```

(c) Implementation

Fig. 15. The Pool module

$$
\begin{array}{rclll}
\{x \mapsto b\} \oplus \cdot & = & \cdot, \{x \mapsto b\}, \cdot & & \text{AnnotRegion-Left} \\
\cdot \oplus \{x \mapsto b\} & = & \cdot, \{x \mapsto b\}, \cdot & & \text{AnnotRegion-Right} \\
\{x \mapsto \mathbf{U}\} \oplus \{x \mapsto \mathbf{U}\} & = & \cdot, \{x \mapsto \mathbf{U}\}, \cdot & & \text{AnnotRegion-Immut} \\
\{x \mapsto \mathbf{U}\} \oplus \{x \mapsto \mathbf{A}\} & = & \{x \mapsto \mathbf{U}\}, \{x \mapsto \mathbf{A}\}, \cdot & & \text{AnnotRegion-MutLeft} \\
\{x \mapsto \mathbf{A}\} \oplus \{x \mapsto b\} & = & \{x \mapsto \mathbf{A}\}, \cdot, \{x \mapsto b\} & & \text{AnnotRegion-MutRight}
\end{array}
$$

$$
\frac{e = \&^b x \mid \&\&^b x}{e \rightsquigarrow_n e, \{x \mapsto b\}}
\qquad
\frac{e = c \mid x}{e \rightsquigarrow_n e, \cdot}
\qquad
\frac{\forall i,\ e_i \rightsquigarrow_{n+1} e'_i, B_i \qquad B_1 \oplus B_2 = S_1, S, S_2}{(e_1\ e_2) \rightsquigarrow_n (\{\!|e'_1|\!\}^{n+1}_{S_1}\ \{\!|e'_2|\!\}^{n+1}_{S_2}), S}
$$

$$
\frac{\forall i,\ e_i \rightsquigarrow_{n+1} e'_i, B_i \qquad B_1 \oplus (B_2 \backslash \{x\}) = S_1, S, S_2 \qquad S'_2 = S_2 \cup B_2\big|_x}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \rightsquigarrow_n \mathtt{let}\ x = \{\!|e'_1|\!\}^{n+1}_{S_1}\ \mathtt{in}\ \{\!|e'_2|\!\}^{n+1}_{S'_2}, S}
\qquad
\textsc{Rewrite-Lam}\ \frac{e \rightsquigarrow_{n+1} e', B \qquad B_x = B\big|_x}{\lambda x.e \rightsquigarrow_n \lambda x.\{\!|e'|\!\}^{n+1}_{B_x}, B\backslash\{x\}}
$$

$$
\frac{\forall i,\ e_i \rightsquigarrow_{n+1} e'_i, B_i \qquad B_1 \oplus (B_2 \backslash \{x, y\}) = S_1, S, S_2 \qquad S'_2 = S_2 \cup B_2\big|_{x,y}}{\mathtt{match}_\phi\ x, y = e_1\ \mathtt{in}\ e_2 \rightsquigarrow_n \mathtt{match}_\phi\ x, y = \{\!|e'_1|\!\}^{n+1}_{S_1}\ \mathtt{in}\ \{\!|e'_2|\!\}^{n+1}_{S'_2}, S}
\qquad
\textsc{Rewrite-Region}\ \frac{e \rightsquigarrow_{n+1} e', B}{\{\!|e|\!\} \rightsquigarrow_n \{\!|e'|\!\}^{n+1}_B, \cdot}
$$

$$
\textsc{Rewrite-Pair}\ \frac{\forall i,\ e_i \rightsquigarrow_{n+1} e'_i, B_i \qquad B_1 \oplus B_2 = S_1, S, S_2}{(e_1, e_2) \rightsquigarrow_n (\{\!|e'_1|\!\}^{n+1}_{S_1}, \{\!|e'_2|\!\}^{n+1}_{S_2}), S}
\qquad
\textsc{AnnotRegion}\ \frac{\forall i,\ b_i \oplus b'_i = S^l_i, S^m_i, S^r_i}{\overline{b_i} \oplus \overline{b'_i} = \cup_i S^l_i, \cup_i S^m_i, \cup_i S^r_i}
\qquad
\textsc{Rewrite-Top}\ \frac{e \rightsquigarrow_1 e', S}{e \rightsquigarrow \{\!|e'|\!\}^1_S}
$$

Fig. 16. Automatic region annotation — $e \rightsquigarrow e'$

can be rewritten as a fully annotated term e'. Through the rule Rewrite-Top, this is defined in term of an inductively defined relation $e \rightsquigarrow_n e', S$ where n is the current nesting and S is a set of variable that are not yet enclosed in a region. The base cases are constants, variables and borrows. The general idea is to start from the leaves of the syntax tree, create a region for each borrow, and enlarge the region as much as possible. This is implemented by a depth-first walk of the syntax tree

$$C ::= (\tau_1 \leq \tau_2) \mid (k_1 \leq k_2) \mid C_1 \wedge C_2 \mid \exists \alpha.C \mid \exists \kappa.C$$

Fig. 17. The constraint language

$$\frac{l \leq_{\mathcal{L}} l'}{\vdash_e (l \leq l')} \qquad \frac{\forall i,\ C \vdash_e (l_i \leq k)}{C \vdash_e (\wedge_i l_i \leq k)} \qquad \frac{\forall i,\ C \vdash_e (k \leq l_i)}{C \vdash_e (k \leq \vee_i l_i)} \qquad \frac{C \vdash_e (k \leq k') \wedge (\tau \leq \tau')}{C \vdash_e (\&^b(k,\tau) \leq \&^b(k',\tau'))}$$

$$\frac{C \vdash_e (\tau_1' \leq \tau_1) \qquad C \vdash_e (\tau_2 \leq \tau_2') \qquad C \vdash_e (k \leq k')}{C \vdash_e (\tau_1 \xrightarrow{k} \tau_2 \leq \tau_1' \xrightarrow{k'} \tau_2')} \qquad \frac{\forall i,\ C \vdash_e (\tau_i = \tau_i)}{C \vdash_e (\mathrm{T}\ \overline{\tau_i} \leq \mathrm{T}\ \overline{\tau_i'})} \qquad \leq \text{ transitive, reflexive}$$

Fig. 18. Base entailment rules – $C \vdash_e D$

which collects each variable that has a corresponding borrow. At each step, it rewrites the inner subterms, consider which borrow must be enclosed by a region now, and return the others for later enclosing. Binders force immediate enclosing of the bound variables, as demonstrated in rule Rewrite-Lam. For nodes with multiple children, we use a scope merge operator to decide if regions should be placed and where. This is shown in rule Rewrite-Pair. The merge operator, written $B_l \oplus B_r = (S_l, S, S_r)$, takes the sets B_l and B_r returned by rewriting the subterms and returns three sets: S_l and S_r indicates the variables that should be immediately enclosed by a region on the left and right subterms and S indicates the set of the yet-to-be-enclosed variables. As an example, the rule AnnotRegion-MutLeft is applied when there is an shared borrow and a exclusive borrow. In that case, a region is created to enclose the shared borrow, while the exclusive borrow is left to be closed later. This is coherent with the rules for environment splitting and suspended bindings from Section 3.3. Explicitly annotated regions are handled specially through rule Rewrite-Region. In that case, we assume that all inner borrows should be enclosed immediately.

C CONSTRAINTS

We place our constraint system in a more general setting. We define the constraint solver in terms of an arbitrary commutative bounded lattice $(\mathcal{L}, \leq_{\mathcal{L}})$, i.e., a lattice which has a minimal and a maximal element ($l^\top$ and $l^\perp$) and where meet and joins are commutative. We write lattice elements as l and $\bigwedge_i l_i$ (resp. $\bigvee_i l_i$) for the greatest lower bound (resp. least upper bound) in $\mathcal{L}$. The lattice for Affe (see Section 3.3) is a bounded lattice with $l^\top = \mathbf{L}_\infty$ and $l^\perp = \mathbf{U}_0$.

Let $C_{\mathcal{L}}$ be the set of constraints in such a lattice $\mathcal{L}$. The full grammar of constraints is shown in Fig. 17. Constraints are made of kind inequalities, conjunctions and projections along with type unification constraints. Since types might contain kinds (for instance, on the arrows), type unification is oriented and written as $\leq$. For simplicity, we consider all type constructors invariant in their parameters and define $(\tau = \tau')$ as $(\tau \leq \tau') \wedge (\tau' \leq \tau)$.

Entailment is denoted by $C \vdash_e D$, where D is a consequence of the constraints C. We say that C and D are equivalent, $C =_e D$, when $C \vdash_e D$ and $D \vdash_e C$. Following HM(X), we define a cylindric constraint system as a system which respects the following properties for any variable x:

$$C \vdash_e \exists x.C \qquad\qquad C \vdash_e D \implies \exists x.C \vdash_e \exists x.D$$
$$\exists x.(C \wedge \exists x.D) =_e \exists x.C \wedge \exists x.D \qquad\qquad \exists x.\exists y.D =_e \exists y.\exists x.D$$

Furthermore, we define a term rewriting system as a system where, for every types τ,τ', there exists an equality predicates $(\tau = \tau')$ which is symmetric, reflexive, transitive, stable under substitution and such that, for any predicate P:

$$(x = y) \wedge \exists x.C \wedge (x = y) \vdash_e C$$
$$P[x \to \tau] =_e \exists x.P \wedge (x = \tau) \text{ where } x \notin \mathrm{fv}(\tau)$$

$\mathcal{C}_{\mathcal{L}}$ is defined as the smallest cylindric term constraint system that satisfies the axiom shown in Fig. 18.

We define the set of solved formed $\mathcal{S}$ as the quotient set of $\mathcal{C}_{\mathcal{L}}$ by $=_e$. We will show later that such constraints are in fact only composed of kind inequalities, and thus correspond to the syntactic constraints used in type and kind schemes. We now define our normalization procedure normalize(C_0, ψ_0), where $C_0 \in \mathcal{C}_{\mathcal{L}}$ is a set of constraints and ψ_0 is a substitution. It returns a constraint $C \in \mathcal{S}$ in solved form and a unifier ψ. The main idea of the algorithm is to first remove all the type equalities by using regular Herbrand unification. After that, we only have a set of inequalities among kinds, which we can consider as a relation. We can then saturate the relation, unify all kinds that are in the same equivalence classes to obtain a most general unifier on kind variables, remove all existentially quantified variables and then minimize back the relation and apply various simplification rules to make the resulting type easier to understand to users.

More precisely, we apply the following steps:

(1) Solve all type equality constraints through Herbrand unification and gather all existential quantifications at the front of the constraint. We obtain a constraint $C^k = \exists \overline{\kappa},\ (k_j \leq k'_j)_j$ and a substitution ψ_τ.
We write $\mathcal{R}$ for the relation $(k_j \leq k'_j)_j$, $\mathcal{G}$ the underlying directed graph and V its vertices.
(2) Saturate the lattice equalities in $\mathcal{R}$.
More precisely, for each kind variable $\kappa \in V$, for each constant l_i (resp. l_j) such that there is a path from l_i to κ (resp. from κ to l_j) in $\mathcal{G}$, add an edge from $\bigvee l_i$ to κ (resp. from κ to $\bigwedge l_j$). This step is well defined since $\mathcal{L}$ is a bounded lattice and $\bigvee \emptyset$ and $\bigwedge \emptyset$ are well defined.
We also complement $\mathcal{R}$ with $(\leq)$ by adding an edge between related constants.
(3) At this point, we can easily check for satisfiability: A constraint is satisfiable (in the given environment) if and only if, for any constants l_1 and l_2 such that there is a path from l_1 to l_2 in $\mathcal{G}$, then $l_1 \leq_{\mathcal{L}} l_2$. If this is not the case, we return **fail**.
(4) For each strongly connected component in $\mathcal{G}$, unify all its vertices and replace it by a representative. We write ψ_k for the substitution that replaces a kind variable by its representative. The representative of a strongly connected component g can be determined as follows:
   - If g does not contain any constant, then the representative is a fresh kind variable.
   - If g contains exactly one constant, it is the representative.
   - Otherwise, the initial constraint C_0 is not satisfiable.

   Note that this step will also detect all unsatisfiable constraints.
(5) Take the transitive closure of $\mathcal{R}$.
(6) Remove all the vertices corresponding to the kind variables $\overline{\kappa}$ that are existentially quantified in C^k.
(7) Take the transitive reduction of $\mathcal{R}$.
(8) Remove the extremums of $\mathcal{L}$ and the edges of $(\leq)$ from $\mathcal{R}$.
(9) Return $C = \{k \leq k' \mid k\,\mathcal{R}\,k'\}$ and $\psi = \psi_\tau \sqcup \psi_k$.

An example of this algorithm in action is shown in Section 4.3. Our algorithm is complete, computes principal normal forms, and already simplifies constraints significantly (thanks to steps 6, 7 and 8). It can be extended with further simplification phases. In particular, our implementation and all the signatures presented in Section 2 use a variance-based simplification where all covariant (resp. contravariant) variables are replaced by their lower (resp. upper) bounds. All the simplification mechanisms presented here, including the variance-based one, are complete. It is also possible to add "best-effort" simplification rules which help reduce the size of inferred signatures even further [32].

C.1 Principal constraint system

We now prove that $C_{\mathcal{L}}$ supports all the properties necessary for principal type inference, as defined by HM(X). We first prove that constraint solving does compute normal forms, and that such normal forms are unique.

Lemma C.1 (Principal normal form). *Given a constraint $D \in C_{\mathcal{L}}$, a substitution ϕ and $(C, \psi) =$ normalize(D, ϕ), then $\phi \leq \psi$, $C =_{\mathrm{e}} \psi D$ and $\psi C = C$.*

Proof. Let us partition ϕ into a part which affects type variables, ϕ_τ, and a part which affects kind variables, ϕ_k.

We write (C^k, ψ_τ) for the result of the modified Herbrand unification on (D, ϕ) in step (1). Herbrand unification computes the most general unifier. Our modified Herbrand unification only output additional kind constraints for kind on the arrows and does not change the result of the unification. Thus, we have $\phi_\tau \leq \psi_\tau$, $C^k =_{\mathrm{e}} \psi_\tau D$ and $\psi_\tau C^k = C^k$.

Let C^{k+} be the result after step (2), we trivially have that $\mathrm{fv}(C^{k+}) = \mathrm{fv}(C^k)$ and that $C^{k+} =_{\mathrm{e}} C^k$.

Let C^A and ψ_k be the results after step (4). By definition, we have $\psi_k C^{k+} =_{\mathrm{e}} C^A$ and $\psi_k C^A = C^A$. Since ϕ_k has already be applied to C before unifying the strongly connected components, we have that $\phi_k \leq \psi_k$.

Let $\psi = \psi_\tau \sqcup \psi_k$. Since ψ_τ and ψ_k have disjoint supports, we have $C^A = \psi_\tau C^A =_{\mathrm{e}} \psi C^{k+} =_{\mathrm{e}} \psi D$ and $\psi C^A = C^A$. Furthermore, $\phi_\tau \sqcup \phi_k \leq \psi_\tau \sqcup \psi_k$.

Steps (5) to (9) all preserve the free variables and the equivalence of constraints, which concludes. □

Lemma C.2 (Uniqueness). *Given (C_1, ψ_1) and (C_2, ψ_2) such that $\psi_1 C_1 =_{\mathrm{e}} \psi_2 C_2$, then* normalize$(C_1, \psi_1)$ *and* normalize(C_2, ψ_2) *are identical up to α-renaming.*

Proof. In Lemma C.1, we have showed that all the steps of the normalization procedure preserve equivalence. Since $\psi_1 C_1 =_{\mathrm{e}} \psi_2 C_2$, equivalence between the two results of the normalization procedures is preserved for all steps.

We write $P(C_a)$ if for all $C = (k, k)'$ such that $C_a \vdash_{\mathrm{e}} C$ and $\nvdash_e C$, we have $C \in \mathcal{R}_a$.

Let us write C'_1 and C'_2 for the constraints after step (4). $P(C'_1)$ and $P(C'_2)$ hold. Indeed, since C'_1 and C'_2 are only composed of existential quantifications and kind inequalities, the only rules that applies are transitivity and lattice inequalities. After step (2) and (5), the associated relations are fully saturated for these two rules, hence all inequalities that can be deduced from C'_a are already present in the relation.

The property P is preserved by step (6) since we only remove inequalities that involve existentially quantified variables. Such inequalities could not be picked in P.

Let us write C''_a for $a \in \{1, 2\}$ the constraints after step (5). Since there are no more existential variables, we have $C''_a = (k_i, k'_i)_i = \mathcal{R}''_a$. For any $C = (k, k')$ such that $\vdash_{\mathrm{e}} C$ and $C''_a \vdash_{\mathrm{e}} C$, then $C \in (\leq) \subset \mathcal{R}''_a$. Indeed, the only trivial inequalities in our system are equalities of the form (κ, κ), which were removed in step (4) and the lattice inequalities.

Let us consider $C = (k, k') \in \mathcal{R}''_1$. Since $C''_1 =_{\mathrm{e}} C''_2$, we have $C''_2 \vdash_{\mathrm{e}} C$. If $\nvdash_e C$, by $P(C''_2)$ we have that $C \in R''_2$. If $\vdash_{\mathrm{e}} C$, then $C \in (\leq) \subset R''_2$. We conclude that $R''_1 \subset R''_2$. By symmetry, $R''_1 = R''_2$ and $C''_1 = C''_2$.

$$\textsc{KVar}\ \frac{(\alpha : k) \in \Gamma}{C \mid \Gamma \vdash_{\mathrm{s}} \alpha : k} \qquad \textsc{KApp}\ \frac{\begin{array}{c}(\mathrm{T} : \forall \overline{\kappa_i}.\, D \Rightarrow (\overline{k'_j}) \rightarrow k') \in \Gamma \\ \psi = [\overline{\kappa_i} \rightarrow \overline{k_i}] \quad C \vdash_{\mathrm{e}} \psi D \quad \forall j \quad C \mid \Gamma \vdash_{\mathrm{s}} \tau_j : k_j \quad C \mid \Gamma \vdash_{\mathrm{s}} k_j \leq \psi k'_j\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} \mathrm{T}\ \overline{\tau_j} : \psi k'}$$

$$\textsc{KPair}\ \frac{\forall i \quad C \mid \Gamma \vdash_{\mathrm{s}} \tau_i : k_i \quad C \mid \Gamma \vdash_{\mathrm{s}} k_i \leq k}{C \mid \Gamma \vdash_{\mathrm{s}} \tau_1 \times \tau_2 : k} \qquad \textsc{KBorrow}\ C \mid \Gamma \vdash_{\mathrm{s}} \&^{b}(k, \tau) : k \qquad \textsc{KArr}\ C \mid \Gamma \vdash_{\mathrm{s}} \tau_1 \xrightarrow{\mathrm{k}} \tau_2 : k$$

Fig. 19. Syntax-directed kinding rule – $C \mid \Gamma \vdash_{\mathrm{s}} \tau : k$

This equality is preserved by step (7) and (8) since the transitive reduction of a directed acyclic graph is unique, which concludes. □

We can now prove all the necessary high level properties.

Lemma C.3. *For all $C \in \mathcal{S}$, $C \vdash_{\mathrm{e}} x = x$ implies $\vdash_{\mathrm{e}} x = x$.*

Proof. By definition of normalize, We have $C = \overline{(k \leq k')}$ such that the underlying relation has no cycles. Thus, we can not deduce neither kind nor type equalities from C. □

Property 1 (Regular constraint system). *$C_{\mathcal{L}}$ is regular, ie, for x, x' two types or kinds, $\vdash_{\mathrm{e}}(x = x')$ implies $\mathrm{fv}(x) = \mathrm{fv}(x')$*

Proof. The only equalities possibles are between variables (via symmetry) or between constants. □

Finally, we can conclude with all the properties we need for $\mathrm{HM}(X)$:

Theorem C.4 (Principal constraints). *$C_{\mathcal{L}}$ has the principal constraint property,* normalize *computes principal normal forms for $C_{\mathcal{L}}$ and $C_{\mathcal{L}}$ is regular.*

This is sufficient to show that $HM(C_{\mathcal{L}})$ is principal. However, we do not use $\mathrm{HM}(X)$ directly but an extended version with kind inference, linear and affine types, and borrow. We extend the proofs of $\mathrm{HM}(X)$ to such a system in Appendix E.

D SYNTAX-DIRECTED TYPING

D.1 Kinding

We write $C \mid \Gamma \vdash_{\mathrm{s}} \tau : k$ if τ has kind k in environment Γ under constraints C. The rules are shown in Fig. 19. Kinds and types follow a small calculus with variables ($\alpha, \ldots$), functions (type constructors t), application ($\mathrm{T}\ \overline{\tau}$) and primitives such as types for arrows ($\tau \xrightarrow{\mathrm{k}} \tau'$) and borrows ($\&^{b}(k, \tau)$). Kind checking can thus be done in a fairly straightforward, syntax-directed fashion by simply following the syntax of the types. Kind arrows can only appear when looking up the kind scheme of a type constructor t. Kind arrows are forbidden in any other contexts.

D.2 Environments

In Section 3.3, we only gave a partial description of the splitting and borrowing relations on environments, $C \vdash_{e} \Gamma = \Gamma \ltimes \Gamma$ and $C \vdash_{e} \Gamma \rightsquigarrow^{x}_{n} \Gamma$. The complete definitions are shown on Figs. 20 and 21. All the definitions are made in term of the inference version, which returns fresh constraints.

$$\textsc{ESplit-Check}\ \frac{D \Lleftarrow \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \vdash_{\mathrm{e}} D}{C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2} \qquad \textsc{ESplit-Empty}\ \cdot \Lleftarrow \cdot = \cdot \ltimes \cdot \qquad \textsc{ESplit-Nonempty}\ \frac{C_1 \Lleftarrow \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C_2 \Lleftarrow b = b_1 \ltimes b_2}{C_1 \wedge C_2 \Lleftarrow \Gamma; b = \Gamma_1; b_1 \ltimes \Gamma_2; b_2}$$

$$\begin{array}{rcccccr}
(\sigma \leq \mathbf{U}_\infty) \vdash_e & (x : \sigma) & = & (x : \sigma) & \ltimes & (x : \sigma) & \text{(Both)} \\
\cdot \vdash_e & (x \div \sigma)^k_{\mathrm{U}} & = & (x \div \sigma)^k_{\mathrm{U}} & \ltimes & (x \div \sigma)^k_{\mathrm{U}} & \text{(Borrow)} \\
\cdot \vdash_e & B_x & = & B_x & \ltimes & \emptyset & \text{(Left)} \\
\cdot \vdash_e & B_x & = & \emptyset & \ltimes & B_x & \text{(Right)} \\
\cdot \vdash_e & (x : \sigma) & = & [x : \sigma]^n_b & \ltimes & (x : \sigma) & \text{(Susp)} \\
\cdot \vdash_e & (x \div \sigma)^k_b & = & [x : \sigma]^n_{\mathrm{U}} & \ltimes & (x \div \sigma)^k_b & \text{(SuspB)} \\
\cdot \vdash_e & [x : \sigma]^{n'}_b & = & [x : \sigma]^n_{\mathrm{U}} & \ltimes & [x : \sigma]^{n'}_b & \text{(SuspS)}
\end{array}$$

Fig. 20. Splitting — environments $C \vdash_e \Gamma = \Gamma_l \ltimes \Gamma_r$; inference $C \Lleftarrow \Gamma = \boldsymbol{\Gamma_l} \ltimes \boldsymbol{\Gamma_r}$; binders $C \Lleftarrow b = \boldsymbol{b_r} \ltimes \boldsymbol{b_l}$

$$\textsc{EBorrow}\ \frac{C_r \Lleftarrow [x : \tau]^n_b \rightsquigarrow_n b}{C_r \Lleftarrow \Gamma; [x : \tau]^n_b \rightsquigarrow^x_n \Gamma; b} \qquad \textsc{EBorrow-Check}\ \frac{C \vdash_{\mathrm{e}} D \qquad D \Lleftarrow \Gamma; [x : \tau]^n_b \rightsquigarrow^x_n \Gamma; b}{C \vdash_{\mathrm{e}} \Gamma; [x : \tau]^n_b \rightsquigarrow^x_n \Gamma; b}$$

$$\textsc{EBorrow-Binder}\ \frac{b \in \{\mathbf{U}, \mathbf{A}\} \qquad C = (b_n \leq k) \wedge (k \leq b_\infty)}{C \Lleftarrow [x : \sigma]^n_b \rightsquigarrow_n (x \div \sigma)^k_b}$$

Fig. 21. Borrowing — environments $C \vdash_e \Gamma \rightsquigarrow^x_n \Gamma'$; inference $C \Lleftarrow \Gamma \rightsquigarrow^{\boldsymbol{x}}_{\boldsymbol{n}} \boldsymbol{\Gamma'}$; binders $C \Lleftarrow b \rightsquigarrow^{\boldsymbol{x}}_{\boldsymbol{n}} \boldsymbol{b'}$

The solving version then simply uses entailment, as shown in rule ESplit-Check and EBorrow-Check. The remaining new rules are dedicated to iterating over the environment.

Fig. 22 defines the rewriting relation on environment constraints, $(\Gamma \leq k) \rightsquigarrow C$, which rewrites a constraint of the form $(\Gamma \leq k)$ into C. It proceeds by iterating over the environment and expanding the constraints for each binding. Suspended bindings are rejected (ConstrSusp). Borrow bindings directly use the annotated kind (ConstrBorrow). Other bindings use the underlying type scheme (ConstrBinding). Type schemes are constrained by first inferring the kind, and then emitting the constraint (ConstrSD and ConstrI).

D.3 Typing

The rules for syntax-directed typing are shown in Fig. 23 and follow the presentation given in Section 3.3. As usual in HM type systems, introduction of type-schemes is included in the Let rule via generalization. We define $\mathrm{gen}(C, \Gamma, \tau) = (\exists \overline{\kappa}, \overline{\alpha}.C, \forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow \tau)$ where $\overline{\kappa}, \overline{\alpha} = (\mathrm{fv}(\tau) \cup \mathrm{fv}(C)) \setminus \mathrm{fv}(\Gamma)$. The typing rules specific to the internal language are shown in Fig. 24.

$$(\cdot \leq k) \rightsquigarrow \cdot \qquad \frac{(\Gamma \leq k) \rightsquigarrow C \qquad \Gamma \vdash (B \leq k) \rightsquigarrow D}{(\Gamma ; B \leq k) \rightsquigarrow C \wedge D} \qquad \textsc{ConstrBinding}\ \frac{\Gamma \vdash (\sigma \leq k) \rightsquigarrow C}{\Gamma \vdash ((x : \sigma) \leq k) \rightsquigarrow C}$$

$$\textsc{ConstrBorrow}\ \frac{}{\Gamma \vdash ((\&^b x : \&^b(k', \tau)) \leq k) \rightsquigarrow (k' \leq k)} \qquad \textsc{ConstrSusp}\ \frac{}{\Gamma \vdash ([x : \sigma]_b^n \leq k) \rightsquigarrow \mathrm{False}}$$

$$\textsc{ConstrSD}\ \frac{C \wedge C_x \mid \Gamma \vdash_{\mathrm{w}} \tau : k' \qquad D = C \wedge C_x \wedge (k' \leq k)}{\Gamma \vdash_{\mathrm{w}} ((\forall \kappa_i \forall (\alpha_j : k_j).\, C_x \Rightarrow \tau) \leq k) \rightsquigarrow D} \qquad \textsc{ConstrI}\ \frac{C \wedge C_x \mid \Gamma \vdash_{\mathrm{s}} \tau : k' \qquad D = C \wedge C_x \wedge (k' \leq k)}{\Gamma \vdash_{\mathrm{s}} (\forall \kappa_i \forall (\alpha_j : k_j).\, C_x \Rightarrow \tau \leq k) \rightsquigarrow D}$$

Fig. 22. Rewriting constraints on environments — $(\Gamma \leq k) \rightsquigarrow C$

$$\textsc{Var}\ \frac{(x : \sigma) \in \Gamma \qquad C_x, \tau_x = \mathrm{Inst}(\Gamma, \sigma) \qquad C \vdash_{\mathrm{e}} C_x \wedge (\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} x : \tau_x} \qquad \textsc{Pair}\ \frac{C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_1 \qquad C \mid \Gamma_2 \vdash_{\mathrm{s}} e_2 : \tau_2}{C \mid \Gamma \vdash_{\mathrm{s}} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\textsc{Region}\ \frac{[x : \tau_x]_b^n \in \Gamma \qquad C \vdash_e \Gamma \rightsquigarrow_n^x \Gamma' \qquad C \mid \Gamma' \vdash_{\mathrm{s}} e : \tau \qquad C \vdash_{\mathrm{e}} (\tau \leq \mathbf{L}_{n-1})}{C \mid \Gamma \vdash_{\mathrm{s}} \{\!\!| e |\!\!\}^n_{\{x \mapsto b\}} : \tau}$$

$$\textsc{Const}\ \frac{C \vdash_{\mathrm{e}} (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} c : \mathrm{CType}(c)} \qquad \textsc{Abs}\ \frac{C \mid \Gamma ; (x : \tau_2) \vdash_{\mathrm{s}} e : \tau_1 \qquad C \vdash_{\mathrm{e}} (\Gamma \leq k)}{C \mid \Gamma \vdash_{\mathrm{s}} \lambda x.e : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$$

$$\textsc{Borrow}\ \frac{(x \div \sigma)_b^k \in \Gamma \qquad C_x, \tau_x = \mathrm{Inst}(\Gamma, \sigma) \qquad C \vdash_{\mathrm{e}} C_x \wedge (\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \&^b x : \&^b(k, \tau)} \qquad \textsc{Reborrow}\ \frac{C \mid \Gamma \vdash_{\mathrm{s}} x : \&^b(k, \tau)}{C \mid \Gamma \vdash_{\mathrm{s}} \&\&^b x : \&^b(k, \tau)}$$

$$\textsc{App}\ \frac{C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_2 \xrightarrow{\mathrm{k}} \tau_1 \qquad C \mid \Gamma_2 \vdash_{\mathrm{s}} e_2 : \tau_2' \qquad C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \vdash_{\mathrm{e}} (\tau_2' \leq \tau_2)}{C \mid \Gamma \vdash_{\mathrm{s}} (e_1\ e_2) : \tau_1}$$

$$\textsc{Let}\ \frac{C \wedge D \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_1 \qquad (C_\sigma, \sigma) = \mathrm{gen}(D, \Gamma, \tau_1) \qquad C \vdash_{\mathrm{e}} C_\sigma \qquad C \mid \Gamma ; (x : \sigma) \vdash_{\mathrm{s}} e_2 : \tau_2 \qquad C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{MatchPair}\ \frac{C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \phi(\tau_1 \times \tau_1') \qquad C \mid \Gamma_2 ; (x : \phi(\tau_1)) ; (x' : \phi(\tau_1')) \vdash_{\mathrm{s}} e_2 : \tau_2 \qquad C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{match}_\phi\ x, x' = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{Create}\ \frac{C \mid \Gamma \vdash_{\mathrm{s}} \tau : k \qquad C \vdash_{\mathrm{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{create} : \tau \rightarrow \mathrm{R}\ \tau}$$

$$\textsc{Observe}\ \frac{C \mid \Gamma \vdash_{\mathrm{s}} \tau : k \qquad C \vdash_{\mathrm{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{observe} : \&^{\mathrm{U}}(k', \mathrm{R}\ \tau) \rightarrow \tau} \qquad \textsc{Update}\ \frac{C \mid \Gamma \vdash_{\mathrm{s}} \tau : k \qquad C \vdash_{\mathrm{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{update} : \&^{\mathrm{A}}(k', \mathrm{R}\ \tau) \rightarrow \tau \xrightarrow{\mathrm{A}} \texttt{Unit}}$$

$$\textsc{Destroy}\ \frac{C \mid \Gamma \vdash_{\mathrm{s}} \tau : k \qquad C \vdash_{\mathrm{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{destroy} : \mathrm{R}\ \tau \rightarrow \texttt{Unit}}$$

Fig. 23. Syntax-directed typing rules – $C \mid \Gamma \vdash_{\mathrm{s}} e : \tau$

$$\textsc{Var}\;\frac{\begin{array}{c}(x : \tau) \in \Gamma \\ C \vdash_{\mathrm{e}} (\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} x : \tau}
\qquad
\textsc{VarInst}\;\frac{\begin{array}{c}(x : \forall \overline{\kappa_i} \forall (\overline{\alpha_j : k_j}).\, C_x \Rightarrow \tau) \in \Gamma \\ \psi = [\overline{\kappa_i \mapsto k_i}, \overline{\alpha_j \mapsto \tau_j}] \\ C \vdash_{\mathrm{e}} \psi(C_x) \wedge (\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} x[\overline{k_i}; \overline{\tau_j}] : \psi\tau}
\qquad
\textsc{App}\;\frac{\begin{array}{c}sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \\ C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_2 \xrightarrow{\mathrm{k}} \tau_1 \\ C \mid \Gamma_2 \vdash_{\mathrm{s}} e_2 : \tau_2' \\ C \vdash_{\mathrm{e}} (\tau_2' \leq \tau_2)\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} (e_1\ e_2)_{sp} : \tau_1}$$

$$\textsc{Pair}\;\frac{\begin{array}{c}sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \\ C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_1 \\ C \mid \Gamma_2 \vdash_{\mathrm{s}} e_2 : \tau_2\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} (e_1, e_2)^k_{sp} : \tau_1 \times \tau_2}
\qquad
\textsc{Abs}\;\frac{C \mid \Gamma; (x : \tau_2) \vdash_{\mathrm{s}} e : \tau_1 \qquad C \vdash_{\mathrm{e}} (\Gamma \leq k)}{C \mid \Gamma \vdash_{\mathrm{s}} \lambda^k x.e : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$$

$$\textsc{Let}\;\frac{sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \tau_1 \qquad C \mid \Gamma; (x : \tau_1) \vdash_{\mathrm{s}} e_2 : \tau_2}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{let}\ x =_{sp} e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{PLet}\;\frac{\begin{array}{c}sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad \sigma_1 = \forall \overline{\kappa_i} (\overline{\alpha_j} : \overline{k_j}).\, D \Rightarrow \tau_2 \xrightarrow{\mathrm{k}} \tau_1 \qquad C \wedge D \mid \Gamma_1; (\overline{\alpha_j} : \overline{k_j}); (x : \tau_2) \vdash_{\mathrm{s}} e_1 : \tau_1 \\ C \wedge D \vdash_{\mathrm{e}} (\Gamma_1 \leq k) \qquad C \vdash_{\mathrm{e}} \exists (\overline{\kappa_i}, \overline{\alpha_j}).D \qquad C \mid \Gamma; (f : \sigma_1) \vdash_{\mathrm{s}} e_2 : \tau_2\end{array}}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{letfun}\ (f : \sigma_1) =_{sp} \lambda^k x.e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{MatchPair}\;\frac{sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_{\mathrm{s}} e_1 : \phi(\tau_1 \times \tau_1') \qquad C \mid \Gamma_2; (x : \phi(\tau_1)); (y : \phi(\tau_1')) \vdash_{\mathrm{s}} e_2 : \tau_2}{C \mid \Gamma \vdash_{\mathrm{s}} \texttt{match}_\phi\ x, x' =_{sp} e_1\ \texttt{in}\ e_2 : \tau_2}$$

Fig. 24. Syntax-directed typing rules for internal language – $C \mid \Gamma \vdash_{\mathrm{s}} e : \tau$

$$\textsc{KArr}\ \frac{}{(\mathrm{True}, \emptyset) \mid \Gamma \vdash_{\mathrm{w}} \tau_1 \xrightarrow{\mathrm{k}} \tau_2 : k} \qquad \textsc{KBorrow}\ \frac{}{(\mathrm{True}, \emptyset) \mid \Gamma \vdash_{\mathrm{w}} \&^b(k, \tau) : k} \qquad \textsc{KVar}\ \frac{(\alpha : k) \in \Gamma}{(\mathrm{True}, \emptyset) \mid \Gamma \vdash_{\mathrm{w}} \alpha : k}$$

$$\textsc{KPair}\ \frac{\forall i,\ (C_i, \psi_i) \mid \Gamma \vdash_{\mathrm{w}} \tau_i : k_i \qquad \kappa \text{ fresh} \qquad (C, \psi) = \mathrm{normalize}(\overline{C_i} \land \overline{(k_i \leq \kappa)_i}, \overline{\psi_i})}{(C, \psi) \mid \Gamma \vdash_{\mathrm{w}} \tau_1 \times \tau_2 : \psi\kappa}$$

$$\textsc{KApp}\ \frac{(\mathrm{T} : \forall \overline{\kappa_i}.\ C_0 \Rightarrow (\overline{k'_j}) \rightarrow k') \in \Gamma \qquad \overline{\kappa'_i} \text{ fresh} \qquad \psi_0 = [\overline{\kappa_i} \rightarrow \overline{\kappa'_i}] \qquad \forall j,\ (C_j, \psi_j) \mid \Gamma \vdash_{\mathrm{w}} \tau_j : k'_j \qquad (C, \psi) = \mathrm{normalize}(C_0 \land \overline{C_j} \land \overline{(k'_j = k_j)_j}, \psi_0 \sqcup \overline{\psi_j})}{(C, \psi|_{\mathrm{fv}(\Gamma)}) \mid \Gamma \vdash_{\mathrm{w}} \mathrm{T}\ \overline{\tau_j} : \psi k}$$

Fig. 25. Kind inference rules – $(C, \psi) \mid \mathbf{\Gamma} \vdash_{\mathrm{w}} \boldsymbol{\tau} : k$

E TYPE INFERENCE

In this appendix, we provide the complete type inference rules and show that our type inference algorithm is sound and complete. The constraints rules are already shown in Section 4. Kind inference is presented in Appendix E.1 and the detailed treatment of let-bindings in Appendix E.2. The type inference rules are shown in Fig. 26. The various theorems and their proofs are direct adaptations of the equivalent statements for HM(X) [35].

E.1 Kind Inference

We write $(C, \psi) \mid \mathbf{\Gamma} \vdash_{\mathrm{w}} \boldsymbol{\tau} : k$ when type τ has kind k in environment Γ under constraints C and unifier ψ. $\mathbf{\Gamma}$ and $\boldsymbol{\tau}$ are the input parameters of our inference procedure. We present the kind inference algorithm as a set of rules in Fig. 25. Higher-kinds are not generally supported and can only appear by looking-up the kind scheme of a type constructor, for use in the type application rule KApp. Type variables must be of a simple kind in rule KVar. Kind schemes are instantiated in the KVar rules by creating fresh kind variables and the associated substitution. KArr and KBorrow simply returns the kind of the primitive arrow and borrow types. The normalize function is used every time several constraints must be composed in order to simplify the constraint and return a most general unifier.

E.2 Generalization and constraints

The Let rule combines several ingredients previously seen: since let expressions are binders, we use Weak on the bound identifier x. Since let-expressions contain two subexpressions, we use the environment splitting relation, $C_s \Leftarrow \Sigma = \Sigma_1 \ltimes (\Sigma_2 \backslash \{x\})$. We remove the x from the right environment, since it is not available outside of the expression e_2, and should not be part of the returned usage environment.

As per tradition in ML languages, generalization is performed on let-bindings. Following HM(X), we write $(C_\sigma, \sigma) = \mathrm{gen}(C, \Gamma, \tau)$ for the pair of a constraint and a scheme resulting from generalization. The definition is provided in Fig. 12. The type scheme σ is created by quantifying over all the appropriate free variables and the current constraints. The generated constraint C_σ uses a new projection operator, $\exists x.D$ where x can be either a type or a kind variable, which allow the creation

$$\textsc{Var}_I\quad \frac{\begin{array}{c}(x : \sigma) \in \Gamma \qquad \sigma = \forall \kappa_i \forall (\alpha_j : k_j).C_x \Rightarrow \tau \\ \overline{\kappa_i'}, \overline{\alpha_j'} \text{ fresh} \qquad \psi' = [\kappa_i \mapsto \kappa_i', \alpha_j \mapsto \alpha_j'] \\ (C, \psi) = \mathrm{normalize}(C_x, \psi')\end{array}}{(x : \sigma) \,|\, (C, \psi|_{\mathrm{fv}(\Gamma)}) \,|\, \Gamma \vdash_{\mathrm{w}} x : \psi\tau}$$

$$\textsc{Abs}_I\quad \frac{\begin{array}{c}\alpha, \kappa \text{ fresh} \qquad \Sigma_x \,|\, (C', \psi') \,|\, \Gamma; (x : \alpha) \vdash_{\mathrm{w}} e : \tau \\ \Sigma = \Sigma_x \backslash \{x\} \qquad D = C' \wedge (\Sigma \leq \kappa) \wedge \mathrm{Weak}_{(x:\alpha)}(\Sigma_x) \\ (C, \psi) = \mathrm{normalize}(D, \psi')\end{array}}{\Sigma \,|\, (C, \psi \backslash \{\alpha, \kappa\}) \,|\, \Gamma \vdash_{\mathrm{w}} \lambda x.e : \psi(\alpha) \xrightarrow{\psi(\kappa)} \tau}$$

$$\textsc{Borrow}_I\quad \frac{\kappa \text{ fresh} \qquad \Sigma \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} x : \tau}{(\&^b x : \&^b(\kappa, \tau)) \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} \&^b x : \&^b(\kappa, \tau)}$$

$$\textsc{ReBorrow}_I\quad \frac{\begin{array}{c}\Sigma \,|\, (C', \psi') \,|\, \Gamma \vdash_{\mathrm{w}} x : \tau' \\ (C, \psi) = \mathrm{normalize}(C' \wedge (\tau' \leq \&^b(\kappa, \tau)), \psi')\end{array}}{(\&^b x : \&^b(\kappa, \tau)) \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} \&^b x : \&^b(\kappa, \tau)}$$

$$\textsc{Region}_I\quad \frac{\begin{array}{c}\Sigma' \,|\, (C', \psi') \,|\, \Gamma \vdash_{\mathrm{w}} e : \tau \qquad (C_\tau, \psi_\tau) \,|\, \Gamma \vdash_{\mathrm{w}} \tau : k_\tau \\ C_r \Lleftarrow \Sigma \leadsto_n^x \Sigma' \qquad D = C' \wedge C_\tau \wedge (k_\tau \leq \mathbf{L}_{n-1}) \wedge C_r \\ (C, \psi) = \mathrm{normalize}(D, \psi' \sqcup \psi_\tau)\end{array}}{\Sigma \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} \{\!|e|\!\}^n_{\{x \mapsto b\}} : \tau}$$

$$\textsc{App}_I\quad \frac{\begin{array}{c}\alpha, \kappa \text{ fresh} \qquad \Sigma_1 \,|\, (C_1, \psi_1) \,|\, \Gamma \vdash_{\mathrm{w}} e_1 : \tau_1 \qquad \Sigma_2 \,|\, (C_2, \psi_2) \,|\, \Gamma \vdash_{\mathrm{w}} e_2 : \tau_2 \\ C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes \Sigma_2 \qquad D = C_1 \wedge C_2 \wedge (\tau_1 \leq \tau_2 \xrightarrow{\kappa} \alpha) \wedge C_s \\ \psi' = \psi_1 \sqcup \psi_2 \qquad (C, \psi) = \mathrm{normalize}(D, \psi')\end{array}}{\Sigma \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} (e_1\ e_2) : \psi(\alpha)}$$

$$\textsc{MatchPair}_I\quad \frac{\begin{array}{c}\alpha, \kappa, \alpha', \kappa' \text{ fresh} \qquad \Sigma_1 \,|\, (C_1, \psi_1) \,|\, \Gamma \vdash_{\mathrm{w}} e_1 : \tau_1 \\ \Gamma' = \Gamma; (x : \phi(\alpha)); (\alpha : \kappa); (x' : \phi(\alpha')); (\alpha' : \kappa') \qquad \Sigma_2 \,|\, (C_2, \psi_2) \,|\, \Gamma' \vdash_{\mathrm{w}} e_2 : \tau_2 \qquad \psi' = \psi_1 \sqcup \psi_2 \\ C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes (\Sigma_2 \backslash \{x, x'\}) \qquad D = C_1' \wedge C_2 \wedge (\tau_1 \leq \phi(\alpha \times \alpha')) \wedge C_s \wedge \mathrm{Weak}_{(x:\phi(\alpha)),(x':\phi(\alpha)')}(\Sigma_2) \\ (C, \psi) = \mathrm{normalize}(D, \psi')\end{array}}{\Sigma \,|\, (C, \psi|_{\mathrm{fv}(\Gamma)}) \,|\, \Gamma \vdash_{\mathrm{w}} \mathtt{match}_\phi\ x, x' = e_1\ \mathtt{in}\ e_2 : \psi\tau_2}$$

$$\textsc{Pair}_I\quad \frac{\begin{array}{c}\Sigma_1 \,|\, (C_1, \psi_1) \,|\, \Gamma \vdash_{\mathrm{w}} e_1 : \tau_1 \qquad \Sigma_2 \,|\, (C_2, \psi_2) \,|\, \Gamma \vdash_{\mathrm{w}} e_2 : \tau_2 \qquad \psi' = \psi_1 \sqcup \psi_2 \\ C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes \Sigma_2 \qquad D = C_1 \wedge C_2 \wedge C_s \qquad (C, \psi) = \mathrm{normalize}(D, \psi')\end{array}}{\Sigma \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\textsc{Let}_I\quad \frac{\begin{array}{c}\Sigma_1 \,|\, (C_1, \psi_1) \,|\, \Gamma \vdash_{\mathrm{w}} e_1 : \tau_1 \qquad (C_\sigma, \sigma) = \mathrm{gen}(C_1, \psi_1\Gamma, \tau_1) \\ \Sigma_2 \,|\, (C_2, \psi_2) \,|\, \Gamma; (x : \sigma) \vdash_{\mathrm{w}} e_2 : \tau_2 \qquad C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes \Sigma_2 \backslash \{x\} \qquad \psi' = \psi_1 \sqcup \psi_2 \\ D = C_\sigma \wedge C_2 \wedge C_s \wedge \mathrm{Weak}_{(x:\sigma)}(\Sigma_2) \qquad (C, \psi) = \mathrm{normalize}(D, \psi')\end{array}}{\Sigma \,|\, (C, \psi|_{\mathrm{fv}(\Gamma)}) \,|\, \Gamma \vdash_{\mathrm{w}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \psi\tau_2}$$

$$\mathrm{Weak}_{(x:\sigma)}(\Sigma) = \begin{cases} \mathrm{True} & \text{if } x \in \Sigma \\ (\sigma \leq \mathbf{A}_\infty) & \end{cases}$$

Fig. 26. Type inference rules – $\Sigma \,|\, (C, \psi) \,|\, \Gamma \vdash_{\mathrm{w}} e : \tau$

of local variables inside the constraints. This allows us to encapsulate all the quantified variables in the global constraints. It also reflects the fact that there should exist at least one solution for C for the scheme to be valid. Odersky et al. [22] give a detailed account on the projection operators in HM inference.

E.3 Soundness

Lemma E.1. *Given constraints C and D and substitution ψ, if $D \vdash_e C$ then $\psi D \vdash_e \psi C$.*

Proof. By induction over the entailment judgment. □

Lemma E.2. *Given a typing derivation $C \mid \Gamma \vdash_s e : \tau$ and a constraint $D \in \mathcal{S}$ in solved form such that $D \vdash_e C$, then $D \mid \Gamma \vdash_s e : \tau$*

Proof. By induction over the typing derivation □

Lemma E.3. *Given a type environment Γ, $\Gamma' \subset \Gamma$, a term e and a variable $x \in \Gamma$, if $C \mid \Gamma' \vdash_s e : \tau$ then $C \wedge \mathrm{Weak}_x(\Gamma') \mid \Gamma'; (x : \Gamma(x)) \vdash_s e : \tau$*

Proof. Trivial if $x \in \Sigma$. Otherwise, by induction over the typing derivation. □

We define the flattening $\Downarrow\Gamma$ of an environment Γ, as the environment where all the binders are replaced by normal ones. More formally:

$$\begin{aligned}\Downarrow\Gamma = &\left\{(x : \sigma) \mid (x : \sigma) \in \Gamma \vee (x \div \sigma)_b^k \in \Gamma \vee [x : \sigma]_b^n \in \Gamma\right\}\\ &\cup \{(\alpha : k) \mid (\alpha : k) \in \Gamma\}\end{aligned}$$

Lemma E.4. *Given a type environment Γ and a term e such that $\Sigma \mid (C, \psi) \mid \Gamma \vdash_w e : \tau$ then $\Downarrow\Sigma \subset \Gamma$.*

Proof. By induction over the typing derivation. □

Theorem E.5 (Soundness of inference). *Given a type environment Γ containing only value bindings, $\Gamma|_\tau$ containing only a type binding and a term e, if $\Sigma \mid (C, \psi) \mid \Gamma; \Gamma_\tau \vdash_w e : \tau$ then $C \mid \psi(\Sigma; \Gamma_\tau) \vdash_s e : \tau$, $\psi C = C$ and $\psi\tau = \tau$*

Proof. We proceed by induction over the derivation of $\vdash_w$. Most of the cases follow the proofs from HM(X) closely. For brevity, we showcase three rules: the treatment of binders and weakening, where our inference algorithm differ significantly from the syntax-directed rule, and the *Pair* case which showcase the treatment of environment splitting.

$$\textbf{Case}\quad \frac{\begin{array}{l}\textsc{Var}_I\\ (x : \sigma) \in \Gamma \qquad \sigma = \forall\kappa_i\forall(\alpha_j : k_j).C_x \Rightarrow \tau\\ \overline{\kappa_i'}, \overline{\alpha_j'} \text{ fresh} \qquad \psi' = [\kappa_i \mapsto \kappa_i', \alpha_j \mapsto \alpha_j']\\ \qquad (C, \psi) = \mathrm{normalize}(C_x, \psi')\end{array}}{(x : \sigma) \mid (C, \psi|_{\mathrm{fv}(\Gamma)}) \mid \Gamma \vdash_w x : \psi\tau}$$

We have $\Sigma = \{x \mapsto \sigma\}$. Without loss of generality, we can consider $\psi_x = \psi'|_{\mathrm{fv}(\Gamma)} = \psi'|_{\mathrm{fv}(\sigma)}$. Since $\Sigma\backslash\{x\}$ is empty and by definition of normalize, we have that $C \vdash_e \psi'(C_x) \wedge (\psi_x\Sigma\backslash\{x\} \leq \mathbf{A}_\infty)$, $\psi' \leq \psi$ and $\psi' C = C$. By definition, $\psi_x\psi' = \psi'$. By rule *Var*, we obtain $C \mid \psi_x(\Sigma; \Gamma_\tau) \vdash_s x : \psi_x\psi'\tau$, which concludes.

$$\textbf{Case}\quad \dfrac{\begin{array}{c}\alpha,\kappa\ \text{fresh} \qquad \Sigma_x \mid (C',\psi') \mid \Gamma;(x:\alpha) \vdash_{\mathrm{w}} e : \tau \\ \Sigma = \Sigma_x \backslash \{x\} \qquad D = C' \wedge (\Sigma \leq \kappa) \wedge \mathrm{Weak}_{(x:\alpha)}(\Sigma_x) \\ (C,\psi) = \mathrm{normalize}(D,\psi')\end{array}}{\Sigma \mid (C, \psi\backslash\{\alpha,\kappa\}) \mid \Gamma \vdash_{\mathrm{w}} \lambda x.e : \psi(\alpha) \xrightarrow{\psi(\kappa)} \tau}\ \textsc{Abs}_I$$

By induction, we have $C' \mid \Sigma_x; \Gamma_\tau \vdash_{\mathrm{s}} e : \tau$, $\psi C = C$ and $\psi\tau = \tau$. By definition of normalize and Lemma E.1, we have $C \vdash_{\mathrm{e}} C' \wedge (\psi'\Sigma \leq \psi'\kappa) \wedge \mathrm{Weak}_x(\psi'\Sigma_x)$ and $\psi \leq \psi'$. By Lemma E.1, we have $C \vdash_{\mathrm{e}} (\Sigma \leq \psi\kappa)$.

We now consider two cases:

(1) If $x \in \Sigma_x$, then $\mathrm{Weak}_x(\psi\Sigma_x) = \mathrm{True}$ and by Lemma E.4, $\Sigma_x = \Sigma; (x : \alpha)$. We can deduce $C' \wedge \mathrm{Weak}_{(x:\alpha)}(\psi\Sigma_x) \mid \psi\Sigma; \Gamma_\tau; (x : \psi(\alpha)) \vdash_{\mathrm{s}} e : \tau$.
(2) If $x \notin \Sigma_x$, then $\Sigma = \Sigma_x$ and $\mathrm{Weak}_{(x:\alpha)}(\psi\Sigma_x) = (\psi\alpha \leq \mathbf{A}_\infty)$. By Lemma E.3, we have $C' \wedge \mathrm{Weak}_{(x:\alpha)}(\psi\Sigma_x) \mid \psi\Sigma; \Gamma_\tau; (x : \psi(\alpha)) \vdash_{\mathrm{s}} e : \tau$

By Lemma E.2, we have $C \mid \psi(\Sigma; \Gamma_\tau); (x : \psi(\alpha)) \vdash_{\mathrm{s}} e : \tau$.
By rule *Abs*, we obtain $C \mid \psi(\Sigma; \Gamma_\tau) \vdash_{\mathrm{s}} \lambda x.e : \psi(\alpha) \xrightarrow{\psi(\kappa)} \tau$ which concludes.

$$\textbf{Case}\quad \dfrac{\begin{array}{c}\Sigma_1 \mid (C_1,\psi_1) \mid \Gamma \vdash_{\mathrm{w}} e_1 : \tau_1 \qquad \Sigma_2 \mid (C_2,\psi_2) \mid \Gamma \vdash_{\mathrm{w}} e_2 : \tau_2 \qquad \psi' = \psi_1 \sqcup \psi_2 \\ C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes \Sigma_2 \qquad D = C_1 \wedge C_2 \wedge C_s \qquad (C,\psi) = \mathrm{normalize}(D,\psi')\end{array}}{\Sigma \mid (C,\psi) \mid \Gamma \vdash_{\mathrm{w}} (e_1, e_2) : \tau_1 \times \tau_2}\ \textsc{Pair}_I$$

By induction, we have $C_1 \mid \psi_1(\Sigma_1; \Gamma^1_\tau) \vdash_{\mathrm{s}} e_1 : \tau_1$, $\psi_1 C_1 = C_1$, and $\psi_1\tau_1 = \tau_1$ and $C_2 \mid \psi_2(\Sigma_2; \Gamma^2_\tau) \vdash_{\mathrm{s}} e_2 : \tau_2$, $\psi_2 C_2 = C_2$ and $\psi_2\tau_2 = \tau_2$. Wlog, we can rename the type Γ^1_τ and Γ^2_τ to be disjoint and define $\Gamma_\tau = \Gamma^1_\tau \cup \Gamma^2_\tau$. By normalization, $C \vdash_{\mathrm{e}} D$, $\psi \leq \psi'$ and $\psi C = C$. By Lemma E.2 and by substitution, we have $C \mid \psi\Sigma_1 \vdash_{\mathrm{s}} e_1 : \psi\tau_1$ and $C \mid \psi\Sigma_2 \vdash_{\mathrm{s}} e_2 : \psi\tau_2$. We directly have that $\psi C_s \Lleftarrow \psi\Sigma = \Sigma_1 \ltimes \psi\Sigma_2$ and by Lemma E.2, $\psi C \vdash_{\mathrm{e}} \psi C_s$. By rule *Pair*, we obtain $C \mid \psi(\Sigma; \Gamma_\tau) \vdash_{\mathrm{s}} (e_1\ e_2) : \psi(\alpha_1 \times \alpha_2))$, which concludes.

□

E.4 Completeness

We now state our algorithm is complete: for any given syntax-directed typing derivation, our inference algorithm can find a derivation that gives a type at least as general. For this, we need first to provide a few additional definitions.

Definition E.6 (More general unifier). Given a set of variable U and ψ, ψ' and ϕ substitutions. Then $\psi \leq^\phi_U \psi'$ iff $(\phi \circ \psi)|_U = \psi'|_U$.

Definition E.7 (Instance relation). Given a constraints C and two schemes $\sigma = \forall\vec{\alpha}.D \Rightarrow \tau$ and $\sigma' = \forall\vec{\alpha}'.D' \Rightarrow \tau'$. Then $C \vdash_{\mathrm{e}} \sigma \leq \sigma'$ iff $C \vdash_{\mathrm{e}} D[\alpha \to \tau'']$ and $C \wedge D' \vdash_{\mathrm{e}} (\tau[\alpha \to \tau''] \leq \tau')$

We also extend the instance relation to environments Γ.

We now describe the interactions between splitting and the various other operations.

Lemma E.8. *Given* $C \Lleftarrow \Gamma = \Gamma_1 \ltimes \Gamma_2$, *Then* $\Downarrow\Gamma = \Downarrow\Gamma_1 \cup \Downarrow\Gamma_2$.

Proof. By induction over the splitting derivation. □

Lemma E.9. *Given* $C \vdash_e \Gamma_1 = \Gamma_2 \ltimes \Gamma_3$, $C' \Lleftarrow \Gamma'_1 = \Gamma'_2 \ltimes \Gamma'_3$ *and* ψ *such that* $\Gamma'_i \subset \Gamma''_i$ *and* $\vdash_e \psi\Gamma''_i \leq \Gamma_i$ *for* $i \in \{1; 2; 3\}$.
Then $C \vdash_{\mathrm{e}} \psi C'$.

Proof. By induction over the derivation of $C' \Leftarrow \Gamma'_1 = \Gamma'_2 \ltimes \Gamma'_3$. □

We can arbitrarily extend the initial typing environment in an inference derivation, since it is not used to check linearity.

Lemma E.10. *Given $\Sigma \,|\, (C, \psi) \,|\, \Downarrow\Gamma \vdash_{\mathrm{w}} e : \tau$ and Γ' such that $\Gamma \subseteq \Gamma'$, then $\Sigma \,|\, (C, \psi) \,|\, \Downarrow\Gamma' \vdash_{\mathrm{w}} e : \tau$*

Proof. By induction over the type inference derivation. □

Finally, we present the completeness theorem.

Theorem E.11 (Completeness). *Given $C' \,|\, \Gamma' \vdash_{\mathrm{s}} e : \tau'$ and $\vdash_{\mathrm{e}} \psi'\Gamma \leq \Gamma'$. Then*

$$\Sigma \,|\, (C, \psi) \,|\, \Downarrow\Gamma \vdash_{\mathrm{w}} e : \tau$$

for some environment Σ, substitution ψ, constraint C and type τ such that

$$\psi \leq^{\phi}_{\mathrm{fv}(\Gamma)} \psi' \qquad C' \vdash_{\mathrm{e}} \phi C \qquad \vdash_{\mathrm{e}} \phi\sigma \leq \sigma' \qquad \Sigma \subset \Gamma$$

where $\sigma' = gen(C', \Gamma', \tau')$ and $\sigma = gen(C, \Gamma, \tau)$

Proof. Most of the difficulty of this proof comes from proper handling of instanciation and generalization for type-schemes. This part is already proven by Sulzmann [35] in the context of HM(X). As before, we will only present few cases which highlights the handling of bindings and environments. For clarity, we will only present the part of the proof that only directly relate to the new aspect introduced by Affe.

Case $\dfrac{C' \,|\, \Gamma'_x; (x : \tau'_2) \vdash_{\mathrm{s}} e : \tau'_1 \qquad C \vdash_{\mathrm{e}} (\Gamma'_x \leq k)}{C' \,|\, \Gamma'_x \vdash_{\mathrm{s}} \lambda^k x.e : \tau'_2 \xrightarrow{\mathrm{k}} \tau'_1}\textsc{Abs}$ and $\vdash_{\mathrm{e}} \psi'\Gamma \leq \Gamma'$.

Let us pick α and κ fresh. Wlog, we choose $\psi'(\alpha) = \tau_2$ and $\psi'(\kappa) = k$ so that $\vdash_{\mathrm{e}} \psi'\Gamma_x \leq \Gamma'_x$. By induction:

$$\Sigma_x \,|\, (C, \psi) \,|\, \Downarrow\Gamma_x; (x : \alpha) \vdash_{\mathrm{w}} e : \tau \qquad \psi \leq^{\phi}_{\mathrm{fv}(\Gamma_x)\cup\{\alpha;\kappa\}} \psi'$$

$$C' \vdash_{\mathrm{e}} \phi C \qquad \vdash_{\mathrm{e}} \phi\sigma \leq \sigma' \qquad \Sigma_x \subset \Gamma_x; (x : \alpha)$$

$$\sigma' = \mathrm{gen}(C', \Gamma'_x; (x : \tau'_2), \tau'_1) \qquad \sigma = \mathrm{gen}(C, \Gamma_x; (x : \alpha), \tau_1)$$

Let $C_a = C \wedge (\Sigma \leq \kappa) \wedge \mathrm{Weak}_{(x:\alpha)}(\Sigma_x)$ and By definition, $\psi_D \backslash \{\alpha; \kappa\} \leq^{\phi_D}_{\mathrm{fv}(\Gamma_x)} \psi$ which means we have $\psi_D \backslash \{\alpha; \kappa\} \leq^{\phi\circ\phi_D}_{\mathrm{fv}(\Gamma_x)} \psi'$. We also have that $\Sigma_x \backslash \{x\} \subset \Gamma_x$.

Since $C \vdash_{\mathrm{e}} (\Gamma'_x \leq k)$, we have $C \vdash_{\mathrm{e}} \psi'(\Sigma \leq \kappa)$. If $x \in \Sigma_x$, then $\mathrm{Weak}_{(x:\alpha)}(\Sigma_x) = \mathrm{True}$. Otherwise we can show by induction that $C' \vdash_{\mathrm{e}} \psi' \,\mathrm{Weak}_{(x:\alpha)}(\Sigma_x)$. We also have $\psi C = C$, which gives us $C' \vdash_{\mathrm{e}} \psi'(C_a)$. We can deduce $C' \vdash_{\mathrm{e}} \psi'(C_a)$.
This means (C', ψ') is a normal form of C_a, so a principal normal form exists. Let $(D, \psi_D) = \mathrm{normalize}(C_a, \psi\backslash\{\alpha; \kappa\})$. By the property of principal normal forms, we have $C' \vdash_{\mathrm{e}} \rho D$ and $\psi_D \leq^{\rho}_{\mathrm{fv}(\Gamma_x)} \psi'$.

By application of $\textsc{Abs}_I$, we have $\Sigma_x \backslash \{x\} \,|\, (C, \psi_D \backslash \{\alpha, \kappa\}) \,|\, \Downarrow\Gamma_x \vdash_{\mathrm{w}} \lambda x.e : \psi_D(\alpha) \xrightarrow{\psi_D(\kappa)} \tau_1$.
The rest of the proof proceeds as in the original HM(X) proof.

Case $\dfrac{\overset{\textsc{Pair}}{C' \mid \Gamma'_1 \vdash_s e_1 : \tau'_1 \qquad C' \mid \Gamma'_2 \vdash_s e_2 : \tau'_1 \qquad C \vdash_e \Gamma' = \Gamma'_1 \ltimes \Gamma'_2}}{C' \mid \Gamma' \vdash_s (e_1\ e_2) : \tau'_1 \times \tau'_2}$
and $\vdash_e \psi'\Gamma \leq \Gamma'$

The only new elements compared to HM(X) is the environment splitting. By induction:

$$\Sigma_1 \mid (C_1, \psi_1) \mid \Downarrow \Gamma_1 \vdash_w e : \tau_1 \qquad \psi_1 \leq^{\phi_2}_{\mathrm{fv}(\Gamma)} \psi'_1 \qquad C' \vdash_e \phi C_1 \qquad \vdash_e \phi_2 \sigma_1 \leq \sigma'_1$$

$$\Sigma_1 \subset \Gamma_1 \qquad \sigma'_1 = \mathrm{gen}(C', \Gamma'_1, \tau'_1) \qquad \sigma_1 = \mathrm{gen}(C, \Gamma_1, \tau_1)$$

and

$$\Sigma_2 \mid (C_2, \psi_2) \mid \Downarrow \Gamma_2 \vdash_w e : \tau_2 \qquad \psi_2 \leq^{\phi_2}_{\mathrm{fv}(\Gamma)} \psi'_2 \qquad C' \vdash_e \phi C_2$$

$$\vdash_e \phi_2 \sigma_2 \leq \sigma'_2 \qquad \Sigma_2 \subset \Gamma_2$$

$$\sigma'_2 = \mathrm{gen}(C', \Gamma'_2, \tau'_2) \qquad \sigma_2 = \mathrm{gen}(C, \Gamma_2, \tau_2)$$

By Lemmas E.8 and E.10, we have

$$\Sigma_1 \mid (C_1, \psi_1) \mid \Downarrow \Gamma \vdash_w e : \tau_1 \qquad \Sigma_2 \mid (C_2, \psi_2) \mid \Downarrow \Gamma \vdash_w e : \tau_2$$

Let $C_s \Lleftarrow \Sigma = \Sigma_1 \ltimes \Sigma_2$. We know that $\vdash_e \psi'\Gamma \leq \Gamma'$, $\vdash_e \psi'_i\Gamma_i \leq \Gamma'_i$ and $\Sigma_i \subset \Gamma_i$. By Lemma E.9, we have $C \vdash_e \psi' C_s$. The rest of the proof follows HM(X).

□

Corollary E.12 (Principality). *Let* True $\mid \Gamma \vdash_s e : \sigma$ *a closed typing judgment. Then* $\Sigma \mid (C, \psi) \mid \Downarrow \Gamma \vdash_w e : \tau$ *such that:*

$$(\mathrm{True}, \sigma_o) = \mathit{gen}(C, \psi\Gamma, \tau) \qquad \vdash_e \sigma_o \leq \sigma$$

F SEMANTICS DEFINITIONS

Fig. 27 presents the full big-step interpretation. Fig. 28 contains the cases for resources.

G PROOFS FOR METATHEORY

- For simplicity, we only consider terms in A-normal forms following the grammar:

$$e ::= \ \dots \mid (x\ x') \mid (x, x')^k \mid \mathtt{match}_\phi\ x, y = z\ \mathtt{in}\ e$$

  Typing and semantics rules are unchanged.
- Borrow qualifiers $\beta ::= \mathbf{U}_n \mid \mathbf{A}_n$ where $n \geq 0$ is a region level. A vector of borrow qualifiers $\overline{\beta}$ is wellformed if all $\mathbf{U}$s come before all $\mathbf{A}$s in the vector.
- Borrow compatibility $\overline{\beta} \nwarrow \beta$,

$$\beta_n \overline{\beta} \nwarrow \beta_n$$

- Store typing $\vdash \delta : \Delta$,

$$\frac{(\forall \ell \in \mathrm{dom}(\delta))\ \ \Delta \vdash \delta(\ell) : \Delta(\ell)}{\vdash \delta : \Delta}$$

- Relating storables to type schemes $\Delta \vdash w : \sigma$
  We write $\mathrm{dis}(\Gamma)$ for γ^{L} and γ^{A} and γ^{U} and $\gamma_\#$ are all disjoint.

$$\frac{(\exists \Gamma)\ \Delta \vdash \gamma : \Gamma \qquad \mathrm{dis}(\Gamma) \qquad C \mid \Gamma; (x : \tau_2) \vdash_s e : \tau_1 \qquad \overline{\alpha} = \mathrm{fv}(\tau_1, \tau_2) \setminus \mathrm{fv}(\Gamma)}{\Delta \vdash (\gamma, \lambda[\overline{\kappa} \mid C \Rightarrow k]x.e) : \forall \overline{\kappa} \forall \overline{(\alpha : k)}.(C \Rightarrow \tau_2 \xrightarrow{\mathrm{k}} \tau_1)}$$

- Relating storables to types $\Delta \vdash w : \tau$

$$\frac{(\exists \Gamma, C)\ \Delta \vdash \gamma : \Gamma \qquad \mathrm{dis}(\Gamma) \qquad C \mid \Gamma; (x : \tau_2) \vdash_s e : \tau_1 \qquad C \vdash_e (\Gamma \leq k)}{\Delta \vdash (\gamma, \lambda^k x.e) : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$$

$$\frac{\Delta \vdash r_1 : \tau_1 \qquad \Delta \vdash r_2 : \tau_2 \qquad \cdot \vdash_e (\tau_1 \leq k) \wedge (\tau_1 \leq k)}{\Delta \vdash (r_1, r_2)^k : \tau_1 \times^k \tau_2} \qquad \frac{\Delta \vdash r : \mathrm{IType}(\mathrm{T}, \overline{\tau})}{\Delta \vdash [r] : \mathrm{T}\ \overline{\tau}} \qquad \Delta \vdash \bullet : \tau$$

- Relating results to type schemes $\Delta \vdash r : \sigma$

$$\Delta \vdash c : \mathrm{CType}(c) \qquad \Delta \vdash \ell : \Delta(\ell) \qquad \frac{\overline{\beta} \nwarrow b_n \qquad \Delta \vdash \ell : \tau}{\Delta \vdash \overline{\beta}\ell : \&^b(b_n, \tau)}$$

- We write $\mathrm{aff}_\Delta(\ell)$ to express that ℓ points to a resource that requires at least affine treatment. Borrow types do not appear in store types as the store only knows about the actual resources. Define $\mathrm{aff}_\Delta(\ell)$ if one of the following cases holds:
  - $\Delta(\ell) = \forall \overline{\kappa} \forall \overline{(\alpha : k)}.(C \Rightarrow \tau_2 \xrightarrow{\mathrm{k}} \tau_1)$ and $C \wedge (k \leq \mathbf{U}_\infty)$ is contradictory;
  - $\Delta(\ell) = \tau_2 \xrightarrow{\mathrm{k}} \tau_1$ and $(\mathbf{A} \leq k)$;
  - $\Delta(\ell) = \tau_1 \times^k \tau_2$ and $(\mathbf{A} \leq k)$;
  - $\Delta(\ell) = \mathrm{T}\ \overline{\tau}$.
- We write $\mathrm{lin}_\Delta(\ell)$ to express that ℓ points to a linear resource.
  Define $\mathrm{lin}_\Delta(\ell)$ if one of the following cases holds:
  - $\Delta(\ell) = \forall \overline{\kappa} \forall \overline{(\alpha : k)}.(C \Rightarrow \tau_2 \xrightarrow{\mathrm{k}} \tau_1)$ and $C \wedge (k \leq \mathbf{A}_\infty)$ is contradictory;
  - $\Delta(\ell) = \tau_2 \xrightarrow{\mathrm{k}} \tau_1$ and $(\mathbf{L} \leq k)$;
  - $\Delta(\ell) = \tau_1 \times^k \tau_2$ and $(\mathbf{L} \leq k)$;

```
let rec eval
  (δ:store) (π:perm) (γ:venv) i e
  : (store × perm × result) sem =
  if i=0 then TimeOut else
  let i' = i − 1 in
  match e with
  | Const (c) ⟶
    Ok (δ, π, c)

  | Var (x) ⟶
    let* r = γ(x) in
    Ok (δ, π, r)

  | Varinst (x, k̄) ⟶
    let* rx = γ(x) in
    let* ℓ = getloc rx in
    let*? () = ℓ ∈ π in
    let* w = δ(ℓ) in
    let* (γ', κ̄', C', k', x', e') = getstpoly w in
    let π' =
      if C'{k̄\>κ̄'} =e [(k' ≤ U)]{k̄\>κ̄'}
      then π
      else π − ℓ
    in
    let w = STCLOS (γ', k'{k̄\>κ̄'}, x', e'{k̄\>κ̄'}) in
    let (ℓ', δ') = salloc δ w in
    Ok (δ', π' + ℓ', ℓ')

  | Lam (k, x, e) ⟶
    let w = STCLOS (γ, k, x, e) in
    let (ℓ', δ') = salloc δ w in
    let π' = π + ℓ' in
    Ok (δ', π', ℓ')

  | App (e1, e2, sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* ℓ1 = getloc r1 in
    let*? () = ℓ1 ∈ π1 in
    let* w = δ1(ℓ1) in
    let* (γ', k', x', e') = getstclos w in
    let π1' = if k' ≤ U then π1 else π1 − ℓ1 in
    let* δ1' = δ1(ℓ1) ⟵ (if k' ≤ U then w else •) in
    let* (δ2, π2, r2) = eval δ1' π1' γ2 i' e2 in
    let* (δ3, π3, r3) = eval δ2 π2 γ'(x'↦ r2) i' e' in
    Ok (δ3, π3, r3)

  | Let (x, e1, e2, sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* (δ2, π2, r2) = eval δ1 π1 γ2(x ↦ r1) i' e2 in
    Ok (δ2, π2, r2)

  | LetFun (f, σ, k, x, e, e', sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let ∀(κ̄, _, C, τ) = σ in
    let w = STPOLY (γ1, κ̄, C, k, x, e') in
    let (ℓ', δ') = salloc δ w in
    let π' = π + ℓ' in
    let* (δ1, π1, r1) = eval δ' π' γ2(f ↦ ℓ') i' e' in
    Ok (δ1, π1, r1)

  | Pair (k, e1, e2, sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* (δ2, π2, r2) = eval δ1 π1 γ2 i' e2 in
    let w = STPAIR (k, r1, r2) in
    let (ℓ', δ') = salloc δ2 w in
    Ok (δ', π2 + ℓ', ℓ')

  | Match (x, x', e1, e2, sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* ℓ = getloc r1 in
    let* w = δ1(ℓ) in
    let* (k', r1', r2') = getstpair w in
    let π1' = (if k' ≤ U then π1 else π1 − ℓ) in
    let γ2' = γ2(x ↦ r1)(x' ↦ r1') in
    let* (δ2, π2, r2) = eval δ1 π1' γ2' i' e2 in
    Ok (δ2, π2, r2)

  | Matchborrow (x, x', e1, e2, sp) ⟶
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* ρ = getaddress r1 in
    let* (b, _, ℓ) = getborrowed_loc r1 in
    let* w = δ1(ℓ) in
    let* (k', r1', r2') = getstpair w in
    let* ρ1 = getaddress r1' in
    let* ρ2 = getaddress r2' in
    let* ρ1' = b.ρ1 in
    let* ρ2' = b.ρ2 in
    let π1'' = (if k' ≤ U then π1 else π1 − ρ) in
    let γ2'' = γ2(x ↦ ρ1')(x' ↦ ρ2') in
    let* (δ2, π2, r2) = eval δ1 π1'' γ2'' i' e2 in
    Ok (δ2, π2, r2)
```

Fig. 27. Big-step interpretation

```
| Region (e, n, x, τx, b) →
    let+ ρ = γ(x) in
    let* ρ' = b.ρ in
    let* π' = reach ρ τx δ in
    let* π'' = b.π' in
    let γ' = γ(x ↦ρ') in
    let π = (π ∪ π'') \ π' in
    let* (δ1, π1, r1) = eval δ π γ' i' e in
    let π1 = (π1 \ π'') ∪ π' in
    Ok (δ1, π1, r1)
| Borrow (b, x) →
    let+ ρ = γ(x) in
    let*? () = ρ ? b && ρ ∈ π in
    Ok (δ, π, ρ)
| Destroy (e1) →
   let* (δ1, π1, r1) = eval δ π γ i' e1 in
   let* ρ = getaddress r1 in
   let* ℓ = getloc r1 in
   let* w = δ1(ℓ) in
   let* r = getstrsrc w in
   let*? () = ρ ∈ π1 in
   let* δ1' = δ1(ℓ) ← • in
   let π1' = π1 − ℓ in
   Ok (δ1', π1', ())
```

```
| Create →
    let w = STRSRC (0) in
    let (ℓ1, δ1) = salloc δ w in
    let π1 = π + ℓ1 in
    Ok (δ1, π1, ℓ1)
| Observe (e1) →
    let* (δ1, π1, r1) = eval δ π γ i' e1 in
    let* ρ = getaddress r1 in
    let*? () = ρ ∈ π1 in
    let* (b, _, ℓ) = getborrowed_loc r1 in
    let*? () = (b = U) in
    let* w = δ1(ℓ) in
    let* r = getstrsrc w in
    Ok (δ1, π1, r)
| Update (e1, e2, sp) →
    let (γ1, γ2) = vsplit γ sp in
    let* (δ1, π1, r1) = eval δ π γ1 i' e1 in
    let* ρ = getaddress r1 in
    let* (b, _, ℓ) = getborrowed_loc r1 in
    let*? () = (b = A) in
    let* (δ2, π2, r2) = eval δ1 π1 γ2 i' e2 in
    let* w = δ2(ℓ) in
    let* r = getstrsrc w in
    let*? () = ρ ∈ π2 in
    let* δ2' = δ2(ℓ) ← STRSRC (r2) in
    let π2' = π2 − ρ in
    Ok (δ2', π2', ())
```

Fig. 28. Big-step interpretation (resources)

  - $\Delta(\ell) = \mathrm{T}\,\overline{\tau}$.
- It remains to characterize unrestricted resources. Define $\mathrm{unr}_\Delta(\ell)$ if neither $\mathrm{aff}_\Delta(\ell)$ nor $\mathrm{lin}_\Delta(\ell)$ holds.
- Relating environments to contexts
  $\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma^{\mathbf{A}}_{\#}, \gamma^{\mathbf{U}}_{\#} : \Gamma$.
  Here we consider an environment $\gamma = (\gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#})$ as a tuple consisting of the active entries in $\gamma^{\mathbf{L}}$ and the entries for exclusive borrows in $\gamma^{\mathbf{A}}$ and for shared borrows in $\gamma^{\mathbf{U}}$, and suspended entries in $\gamma_{\#} = \gamma^{\mathbf{A}}_{\#}, \gamma^{\mathbf{U}}_{\#}$ for affine and unrestricted entries. The suspended entries cannot be used directly, but they can be activated by appropriate borrowing on entry to a region.

$$\Delta \vdash \cdot, \cdot, \cdot, \cdot : \cdot$$

$$\frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma \qquad \Delta \vdash r : \sigma \qquad \mathrm{lin}_{\Delta}(r)}{\Delta \vdash \gamma^{\mathbf{L}}[x \mapsto r], \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma; (x : \sigma)} \qquad \frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma \qquad \Delta \vdash r : \sigma \qquad \mathrm{aff}_{\Delta}(r)}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}[x \mapsto r], \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma; (x : \sigma)}$$

$$\frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma \qquad \Delta \vdash r : \sigma \qquad \mathrm{unr}_{\Delta}(r)}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}[x \mapsto r], \gamma_{\#} : \Gamma; (x : \sigma)} \qquad \frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#}^{\mathbf{A}}, \gamma_{\#}^{\mathbf{U}} : \Gamma \qquad \Delta \vdash r : \sigma}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#}^{\mathbf{A}}, \gamma_{\#}^{\mathbf{U}}[x \mapsto r] : \Gamma; [x : \sigma]_{\mathbf{U}}^{n}}$$

$$\frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#}^{\mathbf{A}}, \gamma_{\#}^{\mathbf{U}} : \Gamma \qquad \Delta \vdash r : \sigma}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#}^{\mathbf{A}}[x \mapsto r], \gamma_{\#}^{\mathbf{U}} : \Gamma; [x : \sigma]_{\mathbf{A}}^{n}} \qquad \frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma \qquad \Delta \vdash \mathbf{U}\rho : \sigma}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}[x \mapsto \mathbf{U}\rho], \gamma_{\#} : \Gamma; (x \div \sigma)_{\mathbf{U}}^{k}}$$

$$\frac{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma \qquad \Delta \vdash \mathbf{A}\rho : \sigma}{\Delta \vdash \gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}[x \mapsto \mathbf{A}\rho], \gamma^{\mathbf{U}}, \gamma_{\#} : \Gamma; (x \div \sigma)_{\mathbf{A}}^{k}}$$

Extending environments and stores.

$$\Delta \leq \Delta \qquad \frac{\Delta \leq \Delta' \qquad \ell \notin \mathrm{dom}(\delta)}{\Delta \leq \Delta'(\ell : \sigma)}$$

$$\delta \leq \delta \qquad \frac{\delta \leq \delta' \qquad \ell \notin \mathrm{dom}(\delta)}{\delta \leq \delta'[\ell \mapsto w]}$$

Lemma G.1 (Store Weakening). *$\Delta \vdash \gamma : \Gamma$ and $\Delta \leq \Delta'$ implies $\Delta' \vdash \gamma : \Gamma$.*

Lemma G.2 (Store Extension).
- *If $\Delta_1 \leq \Delta_2$ and $\Delta_2 \leq \Delta_3$, then $\Delta_1 \leq \Delta_3$.*
- *If $\delta_1 \leq \delta_2$ and $\delta_2 \leq \delta_3$, then $\delta_1 \leq \delta_3$.*

We write $\mathrm{getloc}(\cdot)$ for the function that extracts a multiset of *raw locations* from a result or from the range of the variable environment.

$$\begin{aligned} \mathrm{getloc}(\overline{\beta}\ell) &= \{\ell\} \\ \mathrm{getloc}(c) &= \{\} \\ \mathrm{getloc}(\cdot) &= \{\} \\ \mathrm{getloc}(\gamma(x \mapsto r)) &= \mathrm{getloc}(\gamma) \cup \mathrm{getloc}(r) \end{aligned}$$

We write $\mathrm{reach}_{\delta}(\gamma)$ for the multiset of all *addresses* reachable from $\mathrm{getloc}(\gamma)$ assuming that $\mathrm{getloc}(\gamma) \subseteq \mathrm{dom}(\delta)$[4]. The function $\mathrm{reach}_{\delta}(\cdot)$ is defined in two steps. First a helper function for results, storables, and environments.

[4]In mixed comparisons between a multiset and a set, we tacitly convert a multiset M to its supporting set $\{x \mid M(x) \neq 0\}$.

$$\begin{aligned}
\mathrm{reach}_0(\cdot) &= \cdot \\
\mathrm{reach}_0(\gamma(x \mapsto r)) &= \mathrm{reach}_0(\gamma) \cup \mathrm{reach}_0(r) \\
\mathrm{reach}_0(\rho) &= \{\rho\} \\
\mathrm{reach}_0(c) &= \{\} \\
\mathrm{reach}_0(\mathrm{STPOLY}(\gamma, \overline{\kappa}, C, k, x, e)) &= \mathrm{reach}_0(\gamma) \\
\mathrm{reach}_0(\mathrm{STCLOS}(\gamma, k, x, e)) &= \mathrm{reach}_0(\gamma) \\
\mathrm{reach}_0(\mathrm{STPAIR}(k, r_1, r_2)) &= \mathrm{reach}_0(r_1) \cup \mathrm{reach}_0(r_2) \\
\mathrm{reach}_0(\mathrm{STRSRC}(r)) &= \mathrm{reach}_0(r) \\
\mathrm{reach}_0(\bullet) &= \{\}
\end{aligned}$$

This multiset is closed transitively by store lookup. We define $\mathrm{reach}_\delta(\gamma)$ as the smallest multiset Θ that fulfills the following inequations. We assume a nonstandard model of multisets such that an element ℓ may occur infinitely often as in $\Theta(\ell) = \infty$.

$$\begin{aligned}
&\Theta \supseteq \mathrm{reach}_0(\gamma) \\
&\Theta \supseteq \mathrm{reach}_0(w) && \text{if } \overline{\beta}\ell \in \Theta \wedge w = \delta(\ell)
\end{aligned}$$

Definition G.3 (Wellformed permission). A permission π is *wellformed* if it contains at most one address for each raw location.

Definition G.4 (Permission closure). The closure of a permission $\downarrow\pi$ is the set of addresses reachable from π by stripping an arbitrary number of borrows from it. It is the homomorphic extension of the closure $\downarrow\rho$ for a single address.

$$\downarrow\ell = \{\ell\} \qquad\qquad \downarrow(\beta\rho) = \{\beta\rho\} \cup \downarrow\rho$$

Lemma G.5 (Containment). *Suppose that* $\vdash \delta : \Delta$, $\Delta \vdash r : \tau$, $C \vdash_e (\tau \leq k) \wedge (k \leq \mathbf{L}_{m-1})$. *Then* $\mathrm{reach}_\delta(r)$ *cannot contain addresses* ρ *such that* $\rho = b_n\rho'$ *with* $n \geq m$.

Proof. By inversion of result typing there are three cases.

Case $\Delta \vdash c : \mathrm{CType}(c)$. Immediate: reachable set is empty.

Case $\dfrac{\overline{\beta} \nwarrow b_n \qquad \Delta \vdash \ell : \tau}{\Delta \vdash \overline{\beta}\ell : \&^b(b_n, \tau)}$. The typing constraint enforces that $n < m$.

Case $\Delta \vdash \ell : \Delta(\ell)$. We need to continue by dereferencing ℓ and inverting store typing.

Case $\Delta \vdash \bullet : \tau$. Trivial.

Case $\dfrac{\Delta \vdash r : \mathrm{IType}(\mathrm{T}, \overline{\tau})}{\Delta \vdash [r] : \mathrm{T}\,\overline{\tau}}$. We assume the implementation type of a result to be unrestricted.

Case $\dfrac{\Delta \vdash r_1 : \tau_1 \qquad \Delta \vdash r_2 : \tau_2 \qquad \cdot \vdash_e (\tau_1 \leq k) \wedge (\tau_1 \leq k)}{\Delta \vdash (r_1, r_2)^k : \tau_1 \times^k \tau_2}$.

The typing constraint yields that $k \leq \mathbf{L}_{m-1}$. By induction and transitivity of $\leq$, we find that $\mathrm{reach}_\delta(r_1)$ and $\mathrm{reach}_\delta(r_2)$ cannot contain offending addresses.

Case $\dfrac{(\exists \Gamma, C)\ \Delta \vdash \gamma : \Gamma \qquad \mathrm{dis}(\Gamma) \qquad C \mid \Gamma; (x : \tau_2) \vdash_s e : \tau_1 \qquad C \vdash_e (\Gamma \leq k)}{\Delta \vdash (\gamma, \lambda^k x.e) : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$.

The typing constraint yields that $k \leq \mathbf{L}_{m-1}$. By transitivity of $\leq$ and $\Delta \vdash \gamma : \Gamma$, we find that the types of all addresses in γ have types bounded by $\mathbf{L}_{m-1}$ and, by induction, they cannot contain offending addresses. □

Theorem G.6 (Type Soundness). *Suppose that*

(A1) $C \mid \Gamma \vdash_{s} e : \tau$
(A2) $\Delta \vdash \gamma : \Gamma$
(A3) $\vdash \delta : \Delta$
(A4) π *is wellformed and* $\mathrm{getloc}(\pi) \subseteq \mathrm{dom}(\delta) \setminus \delta^{-1}(\bullet)$
(A5) $\mathrm{reach}_0(\gamma) \subseteq \pi$, $\mathrm{reach}_\delta(\gamma) \subseteq \downarrow\pi$.
(A6) $\mathrm{getloc}(\gamma^{\mathbf{L}})$, $\mathrm{getloc}(\gamma^{\mathbf{A}})$, $\mathrm{getloc}(\gamma^{\mathbf{U}})$, *and* $\mathrm{getloc}(\gamma_{\#})$ *are all disjoint*
(A7) Incoming Resources:
  (a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_\delta(\gamma))$, $\delta(\ell) \neq \bullet$.
  (b) $\forall \ell \in \Theta = \mathrm{getloc}(\mathrm{reach}_\delta(\gamma^{\mathbf{L}}, \gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_{\#}))$, $\Theta(\ell) = 1$.

For all $i \in \mathbb{N}$, *if* $R' = \mathrm{eval}\ \delta\ \pi\ \gamma\ i\ e$ *and* $R' \neq \mathrm{TimeOut}$, *then* $\exists\, \delta', \pi', r', \Delta'$ *such that*

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$
(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
(R3) $\Delta' \vdash r' : \tau$
(R4) π' *is wellformed and* $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_{\#}) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
(R6) Frame:
  For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ *it must be that*
  - $\delta'(\ell) = \delta(\ell)$ *and*
  - *for any* ρ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

(R7) Unrestricted values, resources, and borrows:
  For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_{\#})$ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, *it must be that* $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ *and* $\rho \in \pi'$.
(R8) Affine borrows and resources:
  For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_{\#})$ *with* $\mathrm{getloc}(\rho) = \{\ell\}$, *it must be that* $\ell \in \mathrm{dom}(\delta)$. *If* $\rho \neq \ell$, *then* $\delta'(\ell) \neq \bullet$. *If* $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}_{\#})$, *then* $\rho \in \pi'$.
(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. *Let* $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
  For all $\ell \in \Theta$ *it must be that* $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, *and if* $\delta(\ell)$ *is a resource, then* $\delta'(\ell) = \bullet$.
(R10) No thin air permission:
  $\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.

Some explanations are in order for the resource-related assumptions and statements.

Incoming resources are always active (i.e., not freed). Linear and affine resources as well as suspended affine borrows have exactly one pointer in the environment.

The Frame condition states that only store locations reachable from the current environment can change and that all permissions outside the reachable locations remain the same.

Unrestricted values, resources, and borrows do not change their underlying resource and do not spend their permission.

Affine borrows and resources may or may not spend their permission. Borrows are not freed, but resources may be freed. The permissions for suspended entries remain intact.

A linear resource is always freed.

Outgoing permissions are either inherited from the caller or they refer to newly created values.

Proof. By induction on the evaluation of eval δ π γ i e.

The base case is trivial as eval δ π γ 0 e = TimeOut.

For $i > 0$ consider the different cases for expressions. For lack of spacetime, we only give details on some important cases.

Case e of

| Let (x, e_1, e_2, sp) $\rightarrow$
  let (γ_1, γ_2) = vsplit γ sp **in**
  let* (δ_1, π_1, r_1) = eval δ π γ_1 i' e_1 **in**
  let* (δ_2, π_2, r_2) = eval δ_1 π_1 γ_2(x $\mapsto$ r_1) i' e_2 **in**
  Ok (δ_2, π_2, r_2)

We need to invert rule Let for monomorphic let:

$$\frac{sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_s e_1 : \tau_1 \qquad C \mid \Gamma; (x : \tau_1) \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s \texttt{let}\ x =_{sp} e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{Let}$$

As sp is the evidence for the splitting judgment and vsplit distributes values according to sp, we obtain

$$\Delta \vdash \gamma_1 : \Gamma_1 \tag{1}$$
$$\Delta \vdash \gamma_2 : \Gamma_2 \tag{2}$$

Moreover (using $\uplus$ for disjoint union),

- $\gamma^{\mathbf{L}} = \gamma_1{}^{\mathbf{L}} \uplus \gamma_2{}^{\mathbf{L}}$,
- $\gamma^{\mathbf{A}} = \gamma_1{}^{\mathbf{A}} \uplus \gamma_2{}^{\mathbf{A}}$,
- $\gamma^{\mathbf{U}} = \gamma_1{}^{\mathbf{U}} = \gamma_2{}^{\mathbf{U}}$,
- $\gamma_\# = \gamma_{1\#} \uplus \gamma_{2\#}$ (this splitting does not distinguish potentially unrestricted or affine bindings)

We establish the assumptions for the call eval δ π γ_1 i' e_1.

(A1-1) From inversion: $C \wedge D \mid \Gamma_1 \vdash_s e_1 : \tau_1$
(A1-2) From (1): $\Delta \vdash \gamma_1 : \Gamma_1$
(A1-3) From assumption
(A1-4) From assumption
(A1-5) From assumption because γ_1 is projected from γ.
(A1-6) From assumption because γ_1 is projected from γ.
(A1-7) From assumption because γ_1 is projected from γ.

Hence, we can apply the induction hypothesis and obtain

(R1-1) $R_1 = \mathrm{Ok}(\delta_1, \pi_1, r_1)$
(R1-2) $\Delta \leq \Delta_1$, $\delta \leq \delta_1$, $\vdash \delta_1 : \Delta_1$
(R1-3) $\Delta_1 \vdash r_1 : \tau_1$
(R1-4) π_1 is wellformed and $\mathrm{getloc}(\pi_1) \subseteq \mathrm{dom}(\delta_1) \setminus \delta_1{}^{-1}(\bullet)$.
(R1-5) $\mathrm{reach}_0(r_1) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(r_1) \subseteq \downarrow\pi_1 \cap (\mathrm{reach}_{\delta_1}(\gamma) \setminus \mathrm{reach}_{\delta_1}(\gamma_\#) \cup \mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
(R1-6) Frame:
  For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma_1))$ it must be that
  - $\delta_1(\ell) = \delta(\ell)$ and
  - for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.

(R1-7) Unrestricted values, resources, and borrows:
  For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_1(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_1$.

(R1-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_1(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{A}}_{\#})$, then $\rho \in \pi_1$.

(R1-9) Resources: Let $\Theta = \mathrm{reach}_{\delta}(\gamma^{\mathbf{L}})$. Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_1(\ell) = 1$, $\ell \notin \pi_1$, and if $\delta(\ell)$ is a resource, then $\delta_1(\ell) = \bullet$.

(R1-10) No thin air permission:
$\pi_1 \subseteq \pi \cup (\mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.

To establish the assumptions for the call
eval δ_1 π_1 γ_2(x↦r$_1$) i' e$_2$, we write $\gamma_2' = \gamma(x \mapsto r_1)$.

(A2-1) From inversion: $C \mid \Gamma; (x : \tau_1) \vdash_{\mathrm{s}} e_2 : \tau_2$

(A2-2) From (2) we have $\Delta \vdash \gamma_2 : \Gamma_2$. By store weakening (Lemma G.1) and using (R1-2), we have $\Delta_1 \vdash \gamma_2 : \Gamma_2$. With (R1-3), we obtain $\Delta_1 \vdash \gamma_2(x \mapsto r_1) : \Gamma_2; (x : \tau_1)$.

(A2-3) Immediate from (R1-2).

(A2-4) Immediate from (R1-4).

(A2-5) Show $\mathrm{reach}_0(\gamma_2') \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(\gamma_2') \subseteq \downarrow\pi_1$.
From (A1-5), we have $\mathrm{reach}_0(\gamma_2) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(\gamma_2) \subseteq \downarrow\pi_1$. The extra binding $(x \mapsto r_1)$ goes into one of the compartments according to its type. We conclude by (R1-5).

(A2-6) Disjointness holds by assumption for γ_2 and it remains to discuss r_1. But r_1 is either a fresh resource, a linear/affine resource from γ_1 (which is disjoint), or unrestricted. In each case, there is no overlap with another compartment of the environment.

(A2-7) We need to show Incoming Resources:
(a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma_2'))$, $\delta_1(\ell) \neq \bullet$.
(b) $\forall \ell \in \Theta_1 = \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma_2'^{\mathbf{L}}, \gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}}))$, $\Theta_1(\ell) = 1$.
The first item holds by assumption, splitting, and (for r_1) by (R1-4) and (R1-5).
The second and third items hold by assumption (A1-7), splitting, and framing (R1-6).

Hence, we can apply the induction hypothesis and obtain

(R2-1) $R_2 = \mathrm{Ok}(\delta_2, \pi_2, r_2)$

(R2-2) $\Delta_1 \leq \Delta_2$, $\delta_1 \leq \delta_2$, $\vdash \delta_2 : \Delta_2$

(R2-3) $\Delta_2 \vdash r_2 : \tau_2$

(R2-4) π_2 is wellformed and $\mathrm{getloc}(\pi_2) \subseteq \mathrm{dom}(\delta_2) \setminus {\delta_2}^{-1}(\bullet)$.

(R2-5) $\mathrm{reach}_0(r_2) \subseteq \pi_2$, $\mathrm{reach}_{\delta_2}(r_2) \subseteq \downarrow\pi_2 \cap (\mathrm{reach}_{\delta_2}(\gamma_1) \setminus \mathrm{reach}_{\delta_2}(\gamma_{1\#}) \cup \mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta_1))$.

(R2-6) Frame:
For all $\ell \in \mathrm{dom}(\delta_1) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma_2'))$ it must be that
- $\delta_2(\ell) = \delta_1(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi_1 \Leftrightarrow \rho \in \pi_2$.

(R2-7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma_2'^{\mathbf{U}}, \gamma_{2\#}'^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta_1)$, $\delta_2(\ell) = \delta_1(\ell) \neq \bullet$ and $\rho \in \pi_2$.

(R2-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta_1)$. If $\rho \neq \ell$, then $\delta_2(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_2}(\gamma_{2\#}'^{\mathbf{A}})$, then $\rho \in \pi_2$.

(R2-9) Resources: Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma_2'^{\mathbf{L}})$. Let $\Theta_2 = \mathrm{reach}_{\delta_2}(\gamma_2'^{\mathbf{L}})$.
For all $\ell \in \Theta_1$ it must be that $\Theta_1(\ell) = \Theta_2(\ell) = 1$, $\ell \notin \pi_2$, and if $\delta_1(\ell)$ is a resource, then $\delta_2(\ell) = \bullet$.

(R2-10) No thin air permission:
$\pi_2 \subseteq \pi_1 \cup (\mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta_1))$.

It remains to establish the assertions for the let expression.

(R-1) $R_2 = \mathrm{Ok}(\delta_2, \pi_2, r_2)$
Immediate from (R2-1).

(R-2) $\Delta \leq \Delta_2$, $\delta \leq \delta_2$, $\vdash \delta_2 : \Delta_2$
Transitivity of store extension (Lemma G.2), (R2-2), and (R1-2).

(R-3) $\Delta_2 \vdash r_2 : \tau_2$
Immediate from (R2-3).

(R-4) π_2 is wellformed and $\mathrm{getloc}(\pi_2) \subseteq \mathrm{dom}(\delta_2) \setminus {\delta_2}^{-1}(\bullet)$.
Immediate from (R2-4).

(R-5) $\mathrm{reach}_0(r_2) \subseteq \pi_2$, $\mathrm{reach}_{\delta_2}(r_2) \subseteq \downarrow\pi_2 \cap (\mathrm{reach}_{\delta_2}(\gamma) \setminus \mathrm{reach}_{\delta_2}(\gamma_\#) \cup \mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta))$.
Immediate from (R2-5) because $\mathrm{reach}_{\delta_2}(\gamma_1) \subseteq \mathrm{reach}_{\delta_2}(\gamma)$ and $\mathrm{reach}_{\delta_2}(\gamma_{1\#}) \subseteq \mathrm{reach}_{\delta_2}(\gamma_\#)$.
Moreover, $\mathrm{dom}(\delta) \subseteq \mathrm{dom}(\delta_1)$.

(R-6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma))$ it must be that
- $\delta_2(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_2$.

Suppose that $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma))$.
Then $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma_1))$.
By (R1-6), $\delta_1(\ell) = \delta(\ell)$ and for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$: $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.
But also $\ell \in \mathrm{dom}(\delta_1) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma_2'))$.
By (R2-6), $\delta_2(\ell) = \delta_1(\ell)$ for applicable ρ, $\rho \in \pi_1 \Leftrightarrow \rho \in \pi_2$.
Taken together, we obtain the claim.

(R-7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_2(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_2$.
Follows from (R2-7) or (R1-7) because $\gamma^{\mathbf{U}} = {\gamma_1}^{\mathbf{U}} = {\gamma_2}^{\mathbf{U}}$.

(R-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_2(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_2}(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi_2$.
Follows from (R2-8), (R1-8), and framing.

(R-9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta_2 = \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_2(\ell) = 1$, $\ell \notin \pi_2$, and if $\delta(\ell)$ is a resource, then $\delta_2(\ell) = \bullet$.
Follows from disjoint splitting of $\gamma^{\mathbf{L}}$, (R2-9), (R1-9), and framing.

(R-10) No thin air permission:
$\pi_2 \subseteq \pi \cup (\mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta))$.
Immediate from (R2-10).

Case e **of**

```
| VApp (x₁, x₂)⟶
  let* r₁ = γ(x₁) in
  let* ℓ₁ = getloc r₁ in
  let*? () =  ℓ₁ ∈ π in
  let* w = δ(ℓ₁) in
  let* (γ', k', x', e') = getstclos w in
  let π' = (if k' ≤ U then π else π −  ℓ₁) in
  let* δ' = δ(ℓ₁) ⟵ (if k' ≤ U then w else •) in
  let* r₂ = γ(x₂) in
  let* (δ₃, π₃, r₃) = eval δ' π' γ'(x'↦ r₂) i' e' in
  Ok (δ₃, π₃, r₃)
```

We need to invert rule App:

$$\textsc{App}\quad\frac{(x_1 : \tau_2 \xrightarrow{k} \tau_1) \in \Gamma \quad (x_2 : \tau_2') \in \Gamma \quad C \vdash_e (\tau_2' \le \tau_2) \quad C \vdash_e (\Gamma \backslash \{x_1, x_2\} \le \mathbf{A}_\infty)}{C \mid \Gamma \vdash_s (x_1\ x_2) : \tau_1}$$

We need to establish the assumptions for the recursive call eval δ' π' γ'(x'↦r$_2$) i' e'. We write $\gamma_2' = \gamma'(x' \mapsto r_2)$.

(A1-1) $C' \mid \Gamma'; (x' : \tau_2) \vdash_s e' : \tau_1$, for some C' and Γ'

Applying the first premise of App to $r_1 = \gamma(x_1)$, $\Delta \vdash \gamma : \Gamma$, and inversion of result typing yields that $r_1 = \ell_1$ with $\Delta(\ell_1) = \tau_2 \xrightarrow{k} \tau_1$. By inversion of store typing and storable typing, we find that there exist Γ' and C' such that

(a) $\delta(\ell_1) = (\gamma', \lambda^k x'.e')$
(b) $\mathrm{dis}(\Gamma')$
(c) $\Delta \vdash \gamma', \Gamma'$
(d) $C' \mid \Gamma'; (x' : \tau_2) \vdash_s e' : \tau_1$
(e) $C' \vdash_e (\Gamma' \le k)$

(A1-2) $\Delta \vdash \gamma'(x' \mapsto r_2) : \Gamma'; (x' : \tau_2)$

By (A1-1)c, assumption on γ, and the subtyping premise.

(A1-3) $\vdash \delta' : \Delta'$

by assumption and the released rule of store typing (where we write $\Delta' = \Delta$ henceforth)

(A1-4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$

the possible removal of a permission does not violate wellformedness; the permission is taken away exactly when the closure is destroyed

(A1-5) $\mathrm{reach}_0(\gamma_2') \subseteq \pi$, $\mathrm{reach}_\delta(\gamma_2') \subseteq \downarrow\pi$.

as the reach set is a subset of the incoming environment's reach

(A1-6) $\mathrm{getloc}(\gamma_2'^{\mathbf{L}})$, $\mathrm{getloc}(\gamma_2'^{\mathbf{A}})$, $\mathrm{getloc}(\gamma_2'^{\mathbf{U}})$, and $\mathrm{getloc}(\gamma_{2\#}')$ are all disjoint follows from $\mathrm{dis}(\Gamma')$ and since $r_2 = \Gamma(x_2)$ which is an entry disjoint from the closure $\Gamma(x_1)$.

(A1-7) Incoming Resources:

(a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma_2'))$, $\delta'(\ell) \neq \bullet$.
(b) $\forall \ell \in \Theta' = \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma_2'^{\mathbf{L}}, \gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}}))$, $\Theta'(\ell) = 1$.

The first item holds because of assumption (A1-7).
The second item holds because $\Theta' \subseteq \Theta$ from assumption (A1-7).

The inductive hypothesis yields that $\exists\ \delta_3, \pi_3, r_3, \Delta_3$ such that

(R1-1) $R_3 = \mathrm{Ok}(\delta_3, \pi_3, r_3)$

(R1-2) $\Delta' \leq \Delta_3, \delta' \leq \delta_3, \vdash \delta_3 : \Delta_3$

(R1-3) $\Delta_3 \vdash r_3 : \tau_1$

(R1-4) π_3 is wellformed and $\mathrm{getloc}(\pi_3) \subseteq \mathrm{dom}(\delta_3) \setminus {\delta_3}^{-1}(\bullet)$.

(R1-5) $\mathrm{reach}_0(r_3) \subseteq \pi_3$, $\mathrm{reach}_{\delta_3}(r_3) \subseteq {\downarrow}\pi_3 \cap (\mathrm{reach}_{\delta_3}(\gamma) \setminus \mathrm{reach}_{\delta_3}(\gamma_\#) \cup \mathrm{dom}(\delta_3) \setminus \mathrm{dom}(\delta))$.

(R1-6) Frame:
For all $\ell \in \mathrm{dom}(\delta') \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_3}(\gamma'_2))$ it must be that
- $\delta_3(\ell) = \delta'(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi' \Leftrightarrow \rho \in \pi_3$.

(R1-7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_3}(\gamma_2'^{\mathbf{U}}, \gamma_{2\#}'^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta')$, $\delta_3(\ell) = \delta'(\ell) \neq \bullet$ and $\rho \in \pi_3$.

(R1-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_3}(\gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta')$. If $\rho \neq \ell$, then $\delta_3(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_3}(\gamma_{2\#}'^{\mathbf{A}})$, then $\rho \in \pi_3$.

(R1-9) Resources: Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma_2'^{\mathbf{L}})$. Let $\Theta_3 = \mathrm{reach}_{\delta_3}(\gamma_2'^{\mathbf{L}})$.
For all $\ell \in \Theta'$ it must be that $\Theta'(\ell) = \Theta_3(\ell) = 1$, $\ell \notin \pi_3$, and if $\delta'(\ell)$ is a resource, then $\delta_3(\ell) = \bullet$.

(R1-10) No thin air permission:
$\pi_3 \subseteq \pi' \cup (\mathrm{dom}(\delta_3) \setminus \mathrm{dom}(\delta'))$.

The desired results are immediate because $\mathrm{dom}(\delta) = \mathrm{dom}(\delta')$.

Case e **of**
| Region (e, n, x, τ_x, b) $\rightarrow$
let+ ρ = γ(x) **in**
let* ρ' = b.ρ **in**
let* π' = reach ρ τ_x δ **in**
let* π'' = b.π' **in**
let γ' = γ(x $\mapsto \rho$') **in**
let π = ($\pi \cup \pi$'') \ π' **in**
let* (δ_1, π_1, r$_1$) = eval δ π γ' i' e **in**
let π_1 = (π_1 \ π'') $\cup$ π' **in**
Ok (δ_1, π_1, r$_1$)

We need to invert rule Region

$$\textsc{Region}\ \frac{[x : \tau_x]^n_b \in \Gamma \qquad C \vdash_e \Gamma \rightsquigarrow^x_n \Gamma' \qquad C \mid \Gamma' \vdash_s e : \tau \qquad C \vdash_e (\tau \leq \mathbf{L}_{n-1})}{C \mid \Gamma \vdash_s \{\!|e|\!\}^n_{\{x \mapsto b\}} : \tau}$$

We need to establish the assumptions for the recursive call eval δ π' γ' i' e' where $\gamma' = \gamma(x \mapsto \rho')$.

(A1-1) $C \mid \Gamma' \vdash_s e : \tau$
immediate from the inverted premise

(A1-2) $\Delta \vdash \gamma' : \Gamma'$
the only change of the environments is at x; adding the borrow modifier b succeeds due to the second premise; the address ρ' stored into x is compatible with its type by store typing

(A1-3) $\vdash \delta : \Delta$
Immediate by outer assumption

(A1-4) π is wellformed and $\mathrm{getloc}(\pi) \subseteq \mathrm{dom}(\delta) \setminus \delta^{-1}(\bullet)$
Immediate by outer assumption; adding the modifier does not change the underlying raw location

(A1-5) $\mathrm{reach}_0(\gamma') \subseteq \pi$, $\mathrm{reach}_\delta(\gamma') \subseteq \downarrow\pi$.
locations were swapped simultaneously

(A1-6) $\mathrm{getloc}(\gamma'^{\mathbf{L}})$, $\mathrm{getloc}(\gamma'^{\mathbf{A}})$, $\mathrm{getloc}(\gamma'^{\mathbf{U}})$, and $\mathrm{getloc}(\gamma'_\#)$ are all disjoint
Immediate by assumption

(A1-7) Incoming Resources:
(a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_\delta(\gamma')), \delta(\ell) \neq \bullet$.
(b) $\forall \ell \in \Theta = \mathrm{getloc}(\mathrm{reach}_\delta(\gamma'^{\mathbf{L}}, \gamma'^{\mathbf{A}}, \gamma'^{\mathbf{A}}_\#)), \Theta(\ell) = 1$.
Immediate by assumption.

The induction hypothesis yields the following statements. $\exists\ \delta_1, \pi_1, r_1, \Delta_1$ such that

(R1-1) $R_1 = \mathrm{Ok}(\delta_1, \pi_1, r_1)$
(R1-2) $\Delta \leq \Delta_1$, $\delta \leq \delta_1$, $\vdash \delta_1 : \Delta_1$
(R1-3) $\Delta_1 \vdash r_1 : \tau$
(R1-4) π_1 is wellformed and $\mathrm{getloc}(\pi_1) \subseteq \mathrm{dom}(\delta_1) \setminus {\delta_1}^{-1}(\bullet)$.
(R1-5) $\mathrm{reach}_0(r_1) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(r_1) \subseteq \downarrow\pi_1 \cap (\mathrm{reach}_{\delta_1}(\gamma) \setminus \mathrm{reach}_{\delta_1}(\gamma_\#) \cup \mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
(R1-6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma'))$ it must be that
- $\delta_1(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.

(R1-7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{U}}, \gamma'^{\mathbf{U}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_1(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_1$.

(R1-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{A}}, \gamma'^{\mathbf{A}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_1(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{A}}_{\#})$, then $\rho \in \pi_1$.

(R1-9) Resources: Let $\Theta = \mathrm{reach}_{\delta}(\gamma'^{\mathbf{L}})$. Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_1(\ell) = 1$, $\ell \notin \pi_1$, and if $\delta(\ell)$ is a resource, then $\delta_1(\ell) = \bullet$.

(R1-10) No thin air permission:
$\pi_1 \subseteq \pi \cup (\mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.

It remains to derive the induction hypothesis in the last line. The only additional action is the exchange of permissions which withdraws the borrow.

(R1) $R_1 = \mathrm{Ok}(\delta_1, \pi_1, r_1)$
Immediate

(R2) $\Delta \leq \Delta_1$, $\delta \leq \delta_1$, $\vdash \delta_1 : \Delta_1$
Immediate

(R3) $\Delta_1 \vdash r_1 : \tau$
Immediate

(R4) π_1 is wellformed and $\mathrm{getloc}(\pi_1) \subseteq \mathrm{dom}(\delta_1) \setminus {\delta_1}^{-1}(\bullet)$.
The addresses ρ and ρ' (as well as the elements of π' and π'') have the same raw location, so exchanging them does not affect wellformedness. The underlying set of locations does not change.

(R5) $\mathrm{reach}_0(r_1) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(r_1) \subseteq \downarrow\pi_1 \cap (\mathrm{reach}_{\delta_1}(\gamma) \setminus \mathrm{reach}_{\delta_1}(\gamma_{\#}) \cup \mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
This case is critical for region encapsulation. Here we need to argue that ρ' (and hence π'') is not reachable from r_1 because its type τ is bounded by $\mathbf{L}_{n-1}$ according to the fourth premise. We conclude with Lemma G.5.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma'))$ it must be that
- $\delta_1(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.

Immediate

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{U}}, \gamma'^{\mathbf{U}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_1(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_1$.
Immediate

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{A}}, \gamma'^{\mathbf{A}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_1(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{A}}_{\#})$, then $\rho \in \pi_1$.
Immediate

(R9) Resources: Let $\Theta = \mathrm{reach}_{\delta}(\gamma'^{\mathbf{L}})$. Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma'^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_1(\ell) = 1$, $\ell \notin \pi_1$, and if $\delta(\ell)$ is a resource, then $\delta_1(\ell) = \bullet$.
Immediate

(R10) No thin air permission:
$\pi_1 \subseteq \pi \cup (\mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| Create (x) $\rightarrow$
  let* r = γ(x) **in**
  let w = STRSRC (r) **in**
  let (ℓ_1, δ_1) = salloc δ w **in**
  let π_1 = π + ℓ_1 **in**
  Ok (δ_1, π_1, ℓ_1)

We need to invert the corresponding rule

$$\textsc{Create}\quad \frac{C \mid \Gamma \vdash_s \tau : k \qquad C \vdash_e (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_s \texttt{create} : \tau \rightarrow \mathrm{R}\,\tau}$$

It is sufficient to show that there is some $\Delta_1 = \Delta(\ell_1 : \mathrm{R}\,\tau)$ such that δ_1, π_1, and $r_1 = \ell_1$ fulfill the following requirements.

(R1) $R_1 = \mathrm{Ok}(\delta_1, \pi_1, r_1)$

(R2) $\Delta \leq \Delta_1, \delta \leq \delta_1, \vdash \delta_1 : \Delta_1$
For the last item, we need to show that $\Delta(\ell_1) : \mathrm{R}\,\tau$, but this follows from the setting of w to a resource storable in the semantics.

(R3) $\Delta_1 \vdash r_1 : \mathrm{R}\,\tau$
Immediate from the discussion of the preceding case

(R4) π_1 is wellformed and $\mathrm{getloc}(\pi_1) \subseteq \mathrm{dom}(\delta_1) \setminus {\delta_1}^{-1}(\bullet)$.
Follows from the assumption on π and for ℓ_1 from the allocation of the resource.

(R5) $\mathrm{reach}_0(r_1) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(r_1) \subseteq \downarrow\pi_1 \cap (\mathrm{reach}_{\delta_1}(\gamma) \setminus \mathrm{reach}_{\delta_1}(\gamma_\#) \cup \mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
Immediate from the assignment to π_1.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma))$ it must be that
- $\delta_1(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.

Obvious as no existing location is changed.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_1(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_1$.
Obvious as no existing location has changed and no permission is withdrawn.

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_1(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_1}(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi_1$.
Obvious as no existing location has changed and no permission is withdrawn.

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_1(\ell) = 1$, $\ell \notin \pi_1$, and if $\delta(\ell)$ is a resource, then $\delta_1(\ell) = \bullet$.
By the constraint on Γ in the Create rule, $\gamma^{\mathbf{L}} = \emptyset$.

(R10) No thin air permission:
$\pi_1 \subseteq \pi \cup (\mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| VDestroy (x) $\longrightarrow$
let$*$ r = γ(x) **in**
let$*$ ρ = getaddress r **in**
let$*$ ℓ = getloc r **in**
let$*$ w = $\delta(\ell)$ **in**
let$*$ r = getstrsrc w **in**
let$*$**?** () = $\rho \in \pi$ **in**
let$*$ $\delta_1 = \delta(\ell) \leftarrow \bullet$ **in**
let $\pi_1 = \pi - \ell$ **in**
Ok (δ_1, π_1, ())

We need to invert rule Destroy.

$$\textsc{Destroy}\quad \frac{C \mid \Gamma \vdash_{\mathsf{s}} \tau : k \qquad C \vdash_{\mathsf{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathsf{s}} \mathtt{destroy} : \mathrm{R}\ \tau \longrightarrow \mathtt{Unit}}$$

It is sufficient to show that $\Delta_1 = \Delta$, δ_1, π_1, and $r_1 = ()$ fulfill the following requirements.

(R1) $R_1 = \mathrm{Ok}(\delta_1, \pi_1, r_1)$

(R2) $\Delta \leq \Delta_1$, $\delta \leq \delta_1$, $\vdash \delta_1 : \Delta_1$
Immediate: ℓ was updated to void, which has any type.

(R3) $\Delta_1 \vdash () : \mathtt{Unit}$

(R4) π_1 is wellformed and $\mathrm{getloc}(\pi_1) \subseteq \mathrm{dom}(\delta_1) \setminus {\delta_1}^{-1}(\bullet)$.
By assumption on π and because ℓ was removed.

(R5) $\mathrm{reach}_0(r_1) \subseteq \pi_1$, $\mathrm{reach}_{\delta_1}(r_1) \subseteq {\downarrow}\pi_1 \cap (\mathrm{reach}_{\delta_1}(\gamma) \setminus \mathrm{reach}_{\delta_1}(\gamma_\#) \cup \mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
Immediate because the reach set is empty

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_1}(\gamma))$ it must be that
- $\delta_1(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_1$.

Only $\delta(\ell)$ was changed, which is not reachable from the frame.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_1(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_1$.
Immediate because we updated (destroyed) a resource (in $\gamma^{\mathbf{L}}$).

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_1(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{A}}_\#)$, then $\rho \in \pi_1$.
Immediate because we updated (destroyed) a resource (in $\gamma^{\mathbf{L}}$).

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta_1 = \mathrm{reach}_{\delta_1}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_1(\ell) = 1$, $\ell \notin \pi_1$, and if $\delta(\ell)$ is a resource, then $\delta_1(\ell) = \bullet$.
By the constraint on Γ, ℓ was the only resource passed to this invocation of eval. The claimed condition holds as ℓ was removed from π_1 and the location's contents cleared.

(R10) No thin air permission:
$\pi_1 \subseteq \pi \cup (\mathrm{dom}(\delta_1) \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| Var (x) ⟶
let* r = γ(x) **in**
Ok (δ, π, r)

We need to invert rule Var.

$$\textsc{Var}\ \frac{(x:\tau)\in\Gamma \qquad C\vdash_{\mathrm{e}}(\Gamma\backslash\{x\}\leq\mathbf{A}_\infty)}{C\mid\Gamma\vdash_{\mathrm{s}}x:\tau}$$

We establish that the claims hold for $\delta' = \delta$, $\pi' = \pi$, $r = \gamma(x)$, and $\Delta' = \Delta$.

(R1) $R = \mathrm{Ok}(\delta, \pi, r)$

(R2) $\Delta \leq \Delta$, $\delta \leq \delta$, $\vdash \delta : \Delta$
Immediate by reflexivity and assumption.

(R3) $\Delta \vdash r : \tau$
Immediate by assumption (A1-1)c.

(R4) π is wellformed and $\mathrm{getloc}(\pi) \subseteq \mathrm{dom}(\delta) \setminus \delta^{-1}(\bullet)$.
Immediate by assumption (A1-4)

(R5) $\mathrm{reach}_0(r) \subseteq \pi$, $\mathrm{reach}_\delta(r) \subseteq \downarrow\pi \cap(\mathrm{reach}_\delta(\gamma) \setminus \mathrm{reach}_\delta(\gamma_\#) \cup \mathrm{dom}(\delta) \setminus \mathrm{dom}(\delta))$.
Immediate

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_\delta(\gamma))$ it must be that
- $\delta(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi$.

Immediate as permissions and store stay the same.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_\delta(\gamma^{\mathrm{U}}, \gamma^{\mathrm{U}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi$.
Immediate

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_\delta(\gamma^{\mathrm{A}}, \gamma^{\mathrm{A}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_\delta(\gamma^{\mathrm{A}}_\#)$, then $\rho \in \pi$.
Immediate

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathrm{L}})$. Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathrm{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta(\ell) = 1$, $\ell \notin \pi$, and if $\delta(\ell)$ is a resource, then $\delta(\ell) = \bullet$.
As π remains the same, a linear resource in x is returned untouched.

(R10) No thin air permission:
$\pi \subseteq \pi \cup (\mathrm{dom}(\delta) \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| Const (c) →
Ok (δ, π, c)
We need to invert rule Const.

$$\frac{\overset{\textsc{Const}}{C \vdash_{\mathrm{e}} (\Gamma \leq \mathbf{A}_\infty)}}{C \mid \Gamma \vdash_{\mathrm{s}} c : \mathrm{CType}(c)}$$

We need to establish the claims for $\delta' = \delta$, $\pi' = \pi$, $r' = c$, and $\Delta' = \Delta$:

(R1) $R = \mathrm{Ok}(\delta, \pi, r)$

(R2) $\Delta \leq \Delta, \delta \leq \delta, \vdash \delta : \Delta$
By assumption (A3).

(R3) $\Delta \vdash c : \mathrm{CType}(c)$
by result typing.

(R4) π is wellformed and $\mathrm{getloc}(\pi) \subseteq \mathrm{dom}(\delta) \setminus \delta^{-1}(\bullet)$.
By assumption (A1-4).

(R5) $\mathrm{reach}_0(r) \subseteq \pi$, $\mathrm{reach}_\delta(r) \subseteq \downarrow\pi \cap (\mathrm{reach}_\delta(\gamma) \setminus \mathrm{reach}_\delta(\gamma_\#) \cup \mathrm{dom}(\delta) \setminus \mathrm{dom}(\delta))$.
As $\mathrm{reach}_\delta(c) = \emptyset$.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_\delta(\gamma))$ it must be that
- $\delta(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi$.

Immediate

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_\delta(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi$.
Immediate

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_\delta(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_\delta(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi$.

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta(\ell) = 1$, $\ell \notin \pi$, and if $\delta(\ell)$ is a resource, then $\delta(\ell) = \bullet$.
Immediate as $\gamma^{\mathbf{L}} = \emptyset$.

(R10) No thin air permission:
$\pi \subseteq \pi \cup (\mathrm{dom}(\delta) \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| VPair (k, x_1, x_2) $\rightarrow$
  let* $r_1 = \gamma(x_1)$ **in**
  let* $r_2 = \gamma(x_2)$ **in**
  let w = STPAIR (k, r_1, r_2) **in**
  let (ℓ', δ') = salloc δ w **in**
  let $\pi' = \pi + \ell'$ **in**
  Ok (δ', π', ℓ')

We need to invert rule Pair.

$$\textsc{Pair}\quad \frac{(x_1 : \tau_1) \in \Gamma \qquad (x_2 : \tau_2) \in \Gamma \qquad C \vdash_{\mathrm{e}} (\Gamma \backslash \{x_1, x_2\} \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathrm{s}} (x_1, x_2)^k : \tau_1 \times \tau_2}$$

Show that $\delta', \pi', r' = \ell', \Delta' = \Delta(\ell' : \tau_1 \times^k \tau_2)$ such that

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$

(R2) $\Delta \leq \Delta', \delta \leq \delta', \vdash \delta' : \Delta'$

(R3) $\Delta' \vdash \ell' : \tau_1 \times \tau_2$

(R4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
By assumption (A1-4) and because ℓ' is properly initialized.

(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
By assumption (A1-5), $\mathrm{reach}_{\delta'}(\ell') = \mathrm{reach}_\delta(r_1, r_2) \cup \{\ell'\} \subseteq \mathrm{reach}_\delta(\gamma) \cup \{\ell'\}$ and $\{\ell'\} = \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta)$.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ it must be that
- $\delta'(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

Immediate

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
Immediate

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}_\#)$, then $\rho \in \pi'$.
Immediate

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.
Every such ℓ must be reachable either from r_1 or r_2. So they become reachable from ℓ', as required.

(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| Lam (k, x, e) →
  let w = STCLOS (γ, k, x, e) **in**
  let (ℓ', δ') = salloc δ w **in**
  let π' = π + ℓ' **in**
  Ok (δ', π', ℓ')

We need to invert rule Abs

$$\text{Abs}\ \frac{C \mid \Gamma; (x : \tau_2) \vdash_{\mathrm{s}} e : \tau_1 \qquad C \vdash_{\mathrm{e}} (\Gamma \leq k)}{C \mid \Gamma \vdash_{\mathrm{s}} \lambda x.e : \tau_2 \xrightarrow{\mathrm{k}} \tau_1}$$

Show that δ', π', $r' = \ell'$, and $\Delta' = \Delta(\ell' : \tau_2 \xrightarrow{\mathrm{k}} \tau_1)$ fulfill

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$

(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
Immediate by definition and store typing

(R3) $\Delta' \vdash r' : \tau_2 \xrightarrow{\mathrm{k}} \tau_1$
Immediate by store typing

(R4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
Wellformedness holds by assumption on π and because ℓ' is a new location. The domain constraint is assumed for π and ℓ' is initialized to a closure.

(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq {\downarrow}\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.

$$\begin{aligned} \mathrm{reach}_{\delta'}(r') &= \{\ell'\} \cup \mathrm{reach}_{\delta'}(\gamma) \\ &= \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta) \cup \mathrm{reach}_{\delta'}(\gamma) \end{aligned}$$

Moreover, the constraint $(\Gamma \leq k)$ implies that $\gamma_\# = \emptyset$.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ it must be that
- $\delta'(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

Immediate

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathrm{U}}, \gamma_\#^{\mathrm{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
Immediate

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathrm{A}}, \gamma_\#^{\mathrm{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma_\#^{\mathrm{A}})$, then $\rho \in \pi'$.
Immediate

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathrm{L}})$. Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathrm{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.
The second case is immediately applicable.

(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| Borrow (b, x) ⟶
let+ $\rho = \gamma(\mathrm{x})$ **in**
let*? () = ρ ? b **&&** $\rho \in \pi$ **in**
Ok (δ, π, ρ)

We have to invert rule Borrow

$$\textsc{Borrow}\quad \frac{(x \div \sigma)_b^k \in \Gamma \qquad C_x, \tau_x = \mathrm{Inst}(\Gamma, \sigma) \qquad C \vdash_e C_x \wedge (\Gamma \backslash \{x\} \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_s \&^b x : \&^b(k, \tau)}$$

Show that $\delta' = \delta$, $\pi' = \pi$, $r' = \rho$, $\Delta' = \Delta$ such that

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$

(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
Immediate, no changes.

(R3) $\Delta' \vdash r' : \&^b(k, \tau)$
Immediate by result typing and because the interpreter checks that the permissions of the borrow are very restricted.

(R4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
Immediate (no change).

(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
By typing, ρ is not in $\gamma_\#$. Hence, the condition is immediate.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ it must be that
- $\delta'(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

Immediate as no change.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
Immediate as no change

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi'$.
Immediate

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.
Immediate because Θ, Θ' must be empty

(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate.

Case e **of**
| VObserve (x) ⟶
let* r = γ(x) **in**
let* ρ = getaddress r **in**
let*? () = $\rho \in \pi$ **in**
let* (b, _, ℓ) = getborrowed_loc r **in**
let*? () = (b = U) **in**
let* w = $\delta(\ell)$ **in**
let* r' = getstrsrc w **in**
Ok (δ, π, r')

We have to invert the rule Observe

$$\textsc{Observe}\quad \frac{C \mid \Gamma \vdash_{\mathsf{s}} \tau : k \qquad C \vdash_{\mathsf{e}} (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_{\mathsf{s}} \mathtt{observe} : \&^{\mathbf{U}}(k', \mathrm{R}\ \tau) \rightarrow \tau}$$

Show that $\delta' = \delta$, $\pi' = \pi$, r', and $\Delta' = \Delta$ fulfill

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$

(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
By reflexivity and assumption.

(R3) $\Delta' \vdash r' : \tau$
Immediate by store typing

(R4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
Immediate: no changes.

(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ it must be that
- $\delta'(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

Immediate: no changes.

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
Immediate: no changes to immutables.

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi'$.
Immediate: one particular ρ is overwritten, but not freed.

(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.
Immediate because $\Theta = \emptyset$

(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate

Case e **of**
| VUpdate (x_1, x_2) ⟶
  let* $r_1 = \gamma(x_1)$ **in**
  let* ρ = getaddress r_1 **in**
  let* (b, _, ℓ) = getborrowed_loc r_1 **in**
  let*? () = (b = **A**) **in**
  let* $r_2 = \gamma(x_2)$ **in**
  let* w = $\delta(\ell)$ **in**
  let* r = getstrsrc w **in**
  let*? () = $\rho \in \pi$ **in**
  let* $\delta' = \delta(\ell) \leftarrow$ STRSRC (r_2) **in**
  let $\pi' = \pi - \rho$ **in**
  Ok (δ', π', ())

We need to invert rule Update

$$\frac{C \mid \Gamma \vdash_s \tau : k \qquad C \vdash_e (k \leq \mathbf{U}_0) \wedge (\Gamma \leq \mathbf{A}_\infty)}{C \mid \Gamma \vdash_s \texttt{update} : \&^{\mathbf{A}}(k', \mathrm{R}\ \tau) \longrightarrow \tau \xrightarrow{\mathbf{A}} \texttt{Unit}}\ \textsc{Update}$$

We need to show that δ', π', $r' = ()$, $\Delta' = \Delta$ fulfill

(R1) $R' = \mathrm{Ok}(\delta', \pi', r')$
(R2) $\Delta \leq \Delta'$, $\delta \leq \delta'$, $\vdash \delta' : \Delta'$
Immediate by store typing for ℓ
(R3) $\Delta' \vdash r' : \texttt{Unit}$
Immediate
(R4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$.
Immediate, as we remove a permission from π
(R5) $\mathrm{reach}_0(r') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(r') \subseteq \downarrow\pi' \cap (\mathrm{reach}_{\delta'}(\gamma) \setminus \mathrm{reach}_{\delta'}(\gamma_\#) \cup \mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.
Immediate, as we only update a reachable ℓ
(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma))$ it must be that
- $\delta'(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi'$.

Immediate
(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
Immediate
(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_\#)$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}_\#)$, then $\rho \in \pi'$.
Immediate; for ℓ, we observe that it is overwritten, but not freed.
(R9) Resources: Let $\Theta = \mathrm{reach}_\delta(\gamma^{\mathbf{L}})$. Let $\Theta' = \mathrm{reach}_{\delta'}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.
Immediate because $\gamma^{\mathbf{L}} = \emptyset$ and hence $\Theta = \emptyset$.
(R10) No thin air permission:
$\pi' \subseteq \pi \cup (\mathrm{dom}(\delta') \setminus \mathrm{dom}(\delta))$.

Case e **of**
| VMatch (x, x', z, e$_2$, sp) ⟶
  let (γ_1, γ_2) = vsplit γ sp **in**
  let* r = γ_1(z) **in**
  let* ℓ = getloc r **in**
  let* w = $\delta(\ell)$ **in**
  let* (k, r$_1$, r$_1$') = getstpair w **in**
  let π' = **if** k ≤ U **then** π **else** $\pi - \ell$ **in**
  let* δ' = $\delta(\ell) \leftarrow$ (**if** k ≤ U **then** w **else** •) **in**
  let γ_2' = γ_2(x ↦ r$_1$)(x' ↦ r$_1$') **in**
  let* (δ_2, π_2, r$_2$) = eval δ' π' γ_2' i' e$_2$ **in**
  Ok (δ_2, π_2, r$_2$)

We need to invert rule MatchPair

$$\textsc{MatchPair}\quad \frac{sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad \Gamma_1 = (z : \phi(\tau_1 \times \tau_1')) \qquad C \mid \Gamma_2; (x : \phi(\tau_1)); (x' : \phi(\tau_1')) \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s \mathtt{match}_\phi\ x, x' = z\ \mathtt{in}\ e_2 : \tau_2}$$

The case VMatch corresponds to the match specification $\phi = \mathrm{id}$.

Establish the assumptions for the recursive call with $\gamma_2' = \gamma_2(x \mapsto r_1)(x' \mapsto r_1')$ and $\Delta' = \Delta$:

(A1-1) $C \mid \Gamma_2(x : \tau_1)(x' : \tau_1') \vdash_s e_2 : \tau_2$ by inversion
(A1-2) $\Delta \vdash \gamma_2 : \Gamma_2$ by assumption; moreover, $\Delta \vdash r_1 : \tau_1$ and $\Delta \vdash r_1' : \tau_1'$ by inversion of the store typing for ℓ. As $\Delta' = \Delta$, we have $\Delta' \vdash \gamma_2' : \Gamma_2(x : \tau_1)(x' : \tau_1')$.
(A1-3) $\vdash \delta' : \Delta'$: the only change from assumption is in ℓ which potentially maps to •.
(A1-4) π' is wellformed and $\mathrm{getloc}(\pi') \subseteq \mathrm{dom}(\delta') \setminus \delta'^{-1}(\bullet)$: permission to ℓ is removed iff ℓ is mapped to •.
(A1-5) $\mathrm{reach}_0(\gamma_2') \subseteq \pi'$, $\mathrm{reach}_{\delta'}(\gamma_2') \subseteq \downarrow\pi'$.
by assumption
(A1-6) $\mathrm{getloc}(\gamma_2'^{\mathbf{L}})$, $\mathrm{getloc}(\gamma_2'^{\mathbf{A}})$, $\mathrm{getloc}(\gamma_2'^{\mathbf{U}})$, and $\mathrm{getloc}(\gamma_{2\#}')$ are all disjoint: by assumption and splitting
(A1-7) Incoming Resources:
  (a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma_2'))$, $\delta'(\ell) \neq \bullet$.
  (b) $\forall \ell \in \Theta' = \mathrm{getloc}(\mathrm{reach}_{\delta'}(\gamma_2'^{\mathbf{L}}, \gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}}))$, $\Theta'(\ell) = 1$.

Hence the call to eval yields $\exists\ \delta_2, \pi_2, r_2, \Delta_2$ such that

(R1-1) $R_2 = \mathrm{Ok}(\delta_2, \pi_2, r_2)$
(R1-2) $\Delta' \leq \Delta_2$, $\delta' \leq \delta_2$, $\vdash \delta_2 : \Delta_2$
(R1-3) $\Delta_2 \vdash r_2 : \tau_2$
(R1-4) π_2 is wellformed and $\mathrm{getloc}(\pi_2) \subseteq \mathrm{dom}(\delta_2) \setminus \delta_2^{-1}(\bullet)$.
(R1-5) $\mathrm{reach}_0(r_2) \subseteq \pi_2$, $\mathrm{reach}_{\delta_2}(r_2) \subseteq \downarrow\pi_2 \cap (\mathrm{reach}_{\delta_2}(\gamma) \setminus \mathrm{reach}_{\delta_2}(\gamma_\#) \cup \mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta))$.
(R1-6) Frame:
  For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma))$ it must be that
  - $\delta_2(\ell) = \delta(\ell)$ and
  - for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_2$.

(R1-7) Unrestricted values, resources, and borrows:
  For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{U}}, \gamma_\#^{\mathbf{U}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta'(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi'$.
(R1-8) Affine borrows and resources:
  For all $\rho \in \mathrm{reach}_{\delta'}(\gamma^{\mathbf{A}}, \gamma_\#^{\mathbf{A}})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta'(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta'}(\gamma_\#^{\mathbf{A}})$, then $\rho \in \pi'$.

(R1-9) Resources: Let $\Theta = \text{reach}_{\delta}(\gamma^{\mathrm{L}})$. Let $\Theta' = \text{reach}_{\delta'}(\gamma^{\mathrm{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta'(\ell) = 1$, $\ell \notin \pi'$, and if $\delta(\ell)$ is a resource, then $\delta'(\ell) = \bullet$.

(R1-10) No thin air permission:
$\pi' \subseteq \pi \cup (\text{dom}(\delta') \setminus \text{dom}(\delta))$.

As R_2 is also returned from the match, these results carry over.

Case e **of**

```
| VMatchborrow (x, x', z, e₂, sp) ⟶
  let (γ₁, γ₂) = vsplit γ sp in
  let* r₁ = γ₁(z) in
  let* (b, _, ℓ) = getborrowed_loc r₁ in
  let* w = δ(ℓ) in
  let* (k', r₁', r₂') = getstpair w in
  let* ρ = getaddress r₁ in
  let π'' = (if k' ≤ U then π else π − ρ) in
  let δ'' = δ in
  let* ρ₁ = getaddress r₁' in
  let* ρ₂ = getaddress r₂' in
  let* ρ₁' = b.ρ₁ in
  let* ρ₂' = b.ρ₂ in
  let r₁'' = ρ₁' in
  let r₂'' = ρ₂' in
  let γ₂'' = γ₂(x ↦ r₁'')(x' ↦ r₂'') in
  let* (δ₂, π₂, r₂) = eval δ'' π'' γ₂'' i' e₂ in
  Ok (δ₂, π₂, r₂)
```

We need to invert rule MatchPair

$$\textsc{MatchPair}\quad \frac{sp : C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad \Gamma_1 = (z : \phi(\tau_1 \times \tau_1')) \qquad C \mid \Gamma_2; (x : \phi(\tau_1)); (x' : \phi(\tau_1')) \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s \mathtt{match}_\phi\ x, x' = z\ \mathtt{in}\ e_2 : \tau_2}$$

The case VMatchborrow corresponds to the match specification $\phi = \&^b$. In contrast to the non-borrowing match, the borrowed pair is never deallocated.

Establish the assumptions for the recursive call with $\Delta'' = \Delta$:

(A1-1) $C \mid \Gamma_2(x : \&^b\tau_1)(x' : \&^b\tau_1') \vdash_s e_2 : \tau_2$ by inversion

(A1-2) $\Delta \vdash \gamma_2 : \Gamma_2$ by assumption; moreover, $\Delta \vdash r_1'' : \&^b\tau_1$ and $\Delta \vdash r_2'' : \&^b\tau_1'$ by inversion of the store typing for ρ. As $\Delta'' = \Delta$, we have $\Delta'' \vdash \gamma_2'' : \Gamma_2(x : \&^b\tau_1)(x' : \&^b\tau_1')$.

(A1-3) $\vdash \delta' : \Delta''$: the only change from assumption is in ℓ which potentially maps to $\bullet$.

(A1-4) π'' is wellformed and $\mathrm{getloc}(\pi'') \subseteq \mathrm{dom}(\delta'') \setminus \delta''^{-1}(\bullet)$: permission to ℓ is removed iff ℓ is mapped to $\bullet$.

(A1-5) $\mathrm{reach}_0(\gamma_2'') \subseteq \pi''$, $\mathrm{reach}_{\delta''}(\gamma_2'') \subseteq \downarrow\pi''$.
by assumption

(A1-6) $\mathrm{getloc}(\gamma_2''^{\mathbf{L}})$, $\mathrm{getloc}(\gamma_2''^{\mathbf{A}})$, $\mathrm{getloc}(\gamma_2''^{\mathbf{U}})$, and $\mathrm{getloc}(\gamma''_{2\,\#})$ are all disjoint: by assumption and splitting

(A1-7) Incoming Resources:
  (a) $\forall \ell \in \mathrm{getloc}(\mathrm{reach}_{\delta''}(\gamma_2')), \delta''(\ell) \neq \bullet$.
  (b) $\forall \ell \in \Theta'' = \mathrm{getloc}(\mathrm{reach}_{\delta''}(\gamma_2'^{\mathbf{L}}, \gamma_2'^{\mathbf{A}}, \gamma_{2\#}'^{\mathbf{A}})), \Theta''(\ell) = 1$.

Hence the call to eval yields $\delta_2, \pi_2, r_2, \Delta_2$ such that

(R1-1) $R_2 = \mathrm{Ok}(\delta_2, \pi_2, r_2)$

(R1-2) $\Delta'' \leq \Delta_2$, $\delta'' \leq \delta_2$, $\vdash \delta_2 : \Delta_2$

(R1-3) $\Delta_2 \vdash r_2 : \tau_2$

(R1-4) π_2 is wellformed and $\mathrm{getloc}(\pi_2) \subseteq \mathrm{dom}(\delta_2) \setminus {\delta_2}^{-1}(\bullet)$.

(R1-5) $\mathrm{reach}_0(r_2) \subseteq \pi_2$, $\mathrm{reach}_{\delta_2}(r_2) \subseteq \downarrow\pi_2 \cap (\mathrm{reach}_{\delta_2}(\gamma_2'') \setminus \mathrm{reach}_{\delta_2}(\gamma''_{2\,\#}) \cup \mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta_2''))$.

(R1-6) Frame:
  For all $\ell \in \mathrm{dom}(\delta'') \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma_2''))$ it must be that
  - $\delta_2(\ell) = \delta''(\ell)$ and

- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi'' \Leftrightarrow \rho \in \pi_2$.

(R1-7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma_2''^{\mathbf{U}}, \gamma_2''^{\mathbf{U}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta'')$, $\delta_2(\ell) = \delta''(\ell) \neq \bullet$ and $\rho \in \pi_2$.

(R1-8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma_2''^{\mathbf{A}}, \gamma_2''^{\mathbf{A}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta'')$. If $\rho \neq \ell$, then $\delta_2(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_2}(\gamma_2''^{\mathbf{A}}_{\#})$, then $\rho \in \pi_2$.

(R1-9) Resources: Let $\Theta'' = \mathrm{reach}_{\delta''}(\gamma_2''^{\mathbf{L}})$. Let $\Theta_2 = \mathrm{reach}_{\delta_2}(\gamma_2''^{\mathbf{L}})$.
For all $\ell \in \Theta''$ it must be that $\Theta''(\ell) = \Theta_2(\ell) = 1$, $\ell \notin \pi_2$, and if $\delta''(\ell)$ is a resource, then $\delta_2(\ell) = \bullet$.

(R1-10) No thin air permission:
$\pi_2 \subseteq \pi'' \cup (\mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta''))$.

It remains to relate to result with the original call to eval.

(R1) $R_2 = \mathrm{Ok}(\delta_2, \pi_2, r_2)$

(R2) $\Delta \leq \Delta_2, \delta \leq \delta_2, \vdash \delta_2 : \Delta_2$ because $\Delta'' = \Delta$ and (R1-2).

(R3) $\Delta_2 \vdash r_2 : \tau_2$ by (R1-3)

(R4) π_2 is wellformed and $\mathrm{getloc}(\pi_2) \subseteq \mathrm{dom}(\delta_2) \setminus {\delta_2}^{-1}(\bullet)$. Immediate from (R1-4).

(R5) $\mathrm{reach}_0(r_2) \subseteq \pi_2$, $\mathrm{reach}_{\delta_2}(r_2) \subseteq \downarrow\pi_2 \cap (\mathrm{reach}_{\delta_2}(\gamma) \setminus \mathrm{reach}_{\delta_2}(\gamma_{\#}) \cup \mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta))$. By (R1-5) and because $\delta = \delta''$.

(R6) Frame:
For all $\ell \in \mathrm{dom}(\delta) \setminus \mathrm{getloc}(\mathrm{reach}_{\delta_2}(\gamma))$ it must be that
- $\delta_2(\ell) = \delta(\ell)$ and
- for any ρ with $\mathrm{getloc}(\rho) = \{\ell\}$, $\rho \in \pi \Leftrightarrow \rho \in \pi_2$.

Immediate from (R1-6) because $\delta = \delta''$

(R7) Unrestricted values, resources, and borrows:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{U}}, \gamma^{\mathbf{U}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$, $\delta_2(\ell) = \delta(\ell) \neq \bullet$ and $\rho \in \pi_2$.

(R8) Affine borrows and resources:
For all $\rho \in \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{A}}, \gamma^{\mathbf{A}}_{\#})$ with $\mathrm{getloc}(\rho) = \{\ell\}$, it must be that $\ell \in \mathrm{dom}(\delta)$. If $\rho \neq \ell$, then $\delta_2(\ell) \neq \bullet$. If $\rho \in \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{A}}_{\#})$, then $\rho \in \pi_2$.

(R9) Resources: Let $\Theta = \mathrm{reach}_{\delta}(\gamma^{\mathbf{L}})$. Let $\Theta_2 = \mathrm{reach}_{\delta_2}(\gamma^{\mathbf{L}})$.
For all $\ell \in \Theta$ it must be that $\Theta(\ell) = \Theta_2(\ell) = 1$, $\ell \notin \pi_2$, and if $\delta(\ell)$ is a resource, then $\delta_2(\ell) = \bullet$.
Immediate by (R1-9) because the borrowing match does not deallocate.

(R10) No thin air permission:
$\pi_2 \subseteq \pi \cup (\mathrm{dom}(\delta_2) \setminus \mathrm{dom}(\delta))$.

□