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.

CCS Concepts: • Software and its engineering → Semantics; Functional languages.

Additional Key Words and Phrases: Functional programming, Linear types, Ownership, Type inference

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

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 [Girard 1987] and/or uniqueness types [Barendsen and Smetsers 1995]. 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 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 [Achten and Plasmeijer 1995]. In typestate-oriented programming, they would also modify its type [Aldrich et al. 2009]. Functional session types represent a popular example [Gay and Vasconcelos 2010; Lindley and Morris 2017].

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 [2005] 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 the rights to 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 [Matsakis and Klock II 2014], 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 seamlessly 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 [Bernardy et al. 2018], henceforth LH, Quill [Morris 2016], and ALMS [Tov and Pucella 2011]. 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 \(\sigma^{\text{lin}} \rightarrow \beta\) denotes the type of a single-use function that can be called just once.
1 module File : sig  
2 type t : lin  
3 val fopen : path → t  
4 val write : &!t → string aff −→ unit  
5 val close : t → unit  
6 end

(a) File API

let main () =  
  let h = File.fopen "foo" in  
  File.write &!h "Hello␣";  
  File.write &!h "world!";  
  File.close h

(b) File example

let main () =  
  let h = File.fopen "foo" in  
  { | File.write &!h "Hello␣" |}  
  { | File.write &!h "world!" |};  
  File.close h

(c) File example with regions

Fig. 1. Writing files

once, whereas the multiplicities in LH choose between \( \alpha \to \beta \) and \( \alpha \twoheadrightarrow \beta \) where the latter is a function that promises to use its argument exactly once.

(3) Affe makes use of multiplicity constraints (like Quill) and kind subsumption (unlike Quill). Kind subsumption results in significantly simpler, more readable inferred types.

(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 \( \text{File.t} \). A call like \( \text{File.fopen "foo"} \) returns a linear handle to a newly created file, which \emph{must} be released later on with \( \text{File.close} \) 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 \( \text{File.write} \) 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 \( \text{aff} \) of the second arrow in the type of \( \text{File.write} \): a partial application like \( \text{File.write \&!h} \) 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

\[
\text{let } w = \text{File.write \&!h in } w \text{ "Hello␣";} \; w \text{ "world!" (*type error*)}
\]

The remaining arrows are unrestricted and we write \( \to \) instead of the explicitly annotated \( \text{un} \).

Every borrow is restricted to a \emph{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 \( \text{File.write} \). Affe elaborates regions automatically before type inference. Alternatively, programmers may mark regions explicitly (See Section 2.1).

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 \( \text{File.t} \).

Direct style means that there is a function like \( \text{fopen} \) 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 \( \text{File} \) might provide functions like

\[
\text{withFile : path → (handle ← Unrestricted r) → r, which creates a new file handle and takes a continuation that uses the handle linearly, but returns an unrestricted value}^1
\]

\[
^1\text{For technical reasons, LH requires the programmer to use a type like Unrestricted at this point.}
\]
writeFile : string → handle → handle, which returns the transformed resource handle, and
closeFile : handle → unit, which consumes the handle by closing the file.

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.

```haskell
let app f x = f x
# app : (α ↠ β) → (α ↠ β)
```

The reading of the inferred type is straightforward. If \( f \) is a \( \kappa \)-restricted function, then so is \( \text{app } f \). The multiplicities of \( \alpha \) and \( \beta \) 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 \( \alpha \)
and \( \beta \), but the full prefix \( \forall \kappa \forall (\alpha : \kappa_1) \forall (\beta : \kappa_2) \) of the type of \( \text{app} \) is only revealed as much as
necessary for understanding the type.

For compose, Affe infers this type.

```haskell
let compose f g x = f (g x)
# compose : (\kappa \leq \kappa_1) ⇒ (\beta \leq \gamma) → (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow (\alpha \rightarrow \gamma)
```

Like in \( \text{app} \), the multiplicities of the type variables \( \alpha, \beta, \gamma \) do not matter. However, the multiplicity \( \kappa \)
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 \( \kappa_1 \). 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 \( \kappa, \kappa_1 \) as well as the kinds of
\( \alpha, \beta, \gamma \).

### 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 existing ML-like languages.
- Expressive type soundness theorem with respect to a big-step linearity-aware semantics.
- An extension of the HM(\( X \)) framework [Odersky et al. 1999] 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 sim-
  plification 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 supports the resource-passing style common in functional encodings of session types (e.g.,
[Padovani 2017]; see also Appendix A.1 in the supplement) as well as other functional resource

²Affe’s signature for application and composition are simpler than Quill [Morris 2016] thanks to kind subsumption.
kindly bent to free us 103:5

```
mODULE Array : sig

  type (α : k) 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 → unit) × α t → unit

END
```

Fig. 2. Linear arrays

```
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
```

Fig. 3. An array of Fibonacci numbers

Handling. But it really shines when manipulating mutable resources like buffers or connection pools using a mix of functional and imperative programming styles. To support this usage pattern of linearity, Affe relies on the notion of borrowing [Boyland and Retert 2005]. Our first example of linear arrays demonstrates simple borrowing and imperative programming; the second example demonstrates reborrowing and the interaction between closures and borrowing by implementing a Sudoku solver based on a hybrid copy-on-write data structure; the third example demonstrates advanced uses of regions with iterators on linear values and the low-level primitives needed to implement them. 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 or linear 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. To free an array the elements must be affine. Thanks to subkinding, the type of free is pleasingly simple: any type α whose kind is less than or equal to aff is acceptable. The length function is always applicable. The get function is only applicable if the element type is unrestricted as one element is duplicated. To set an array element displaces the previous content, which must be at least affine.

The map function can transform arrays with arbitrary elements. In particular, it can turn unrestricted elements into linear (affine) ones. It takes a borrow of the input array and returns a newly created output array. As freeing requires affine elements, we provide the iter function which takes a suitable finalizer and an array with arbitrary elements, which is consumed. Indeed such an iteration is the only way to free an array with linear elements. A real-life API would provide a combination of map and iter as a “destructive” map that consumes its input array. Assuming a uniform representation, such a destructive map might be implemented by in-place update.

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 as 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 borrow are affine (aff).
Using the API we can create an array of Fibonacci numbers in an imperative coding style, as shown in Fig. 3. 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 (recall that shared borrows are unrestricted and may thus be duplicated). 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 { | ... | }:

```plaintext
for i = 2 to n - 1 do {
    let x = (| get (&a, i-1) + get (&a, i-2) |) in
    set (&a, i, x)
} 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. Affine guarantees that borrow never escape the smallest enclosing region. It employs a system of indexed kinds like \texttt{aff}_r and \texttt{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 \texttt{aff}_1 whereas the type of &a in Line 4 has kind \texttt{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, borrow cannot escape because they are consumed immediately by get and set.

### 2.2 Solving Sudokus with Hybrid Data-Structures

This section presents an implementation of a backtracking Sudoku solver using a safe API for persistent arrays that supports both mutable updates and immutable versioning. The implementation showcases safe mixing of resource allocation and deallocation in the presence of exclusive (mutable) and immutable borrows. It also demonstrates two new aspects: the interaction between closures and borrows and the notion of reborrowing.

Recently introduced persistent data structures permit transient mutations where non-linear uses lead to degraded performance [Conchon and Filliâtre 2007] or to dynamic and static checks [Puente 2017]. 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.

Our implementation of a backtracking Sudoku solver abstracts this scenario. The solver maintains a two-dimensional array to represent the state of the game and uses backtracking when there are several choices to proceed. As choice points may be revisited several times, it seems advantageous to select a persistent data structure for the array. However, local changes between choice points may be implemented as cheap in-place mutations.

Fig. 4 contains an API \texttt{HYBARRAY} along with an implementation \texttt{CowArray} that enables using mutable and immutable modifications to the board through affine types and borrows. The signature differs slightly from the \texttt{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 \texttt{Array}. The in-place mutation function \texttt{set\_mut} with type \&!(\alpha\ t)\times\texttt{int}\times\alpha\rightarrow\texttt{unit} works on an exclusive borrow whereas the persistent set operation has type \&!(\alpha\ t)\times\texttt{int}\times\alpha\rightarrow\alpha\ t. It takes a shared borrow because it only reads from the argument array, but returns a fresh, modified structure. The module \texttt{CowArray}, also in Fig. 4, contains a very simple implementation of \texttt{HYBARRAY} that represents hybrid arrays as regular arrays and uses copy-on-write for persistent modifications.

The function \texttt{mapi}: (\texttt{int}\times\&\alpha\rightarrow\beta)\times\&!(\alpha\ t)\rightarrow\beta\ t is a simple variation on \texttt{Array.map} where the...
module type HYBARRAY = sig
  include ARRAY
  val set : &\(\alpha\) t \times int \times \alpha \rightarrow \alpha t
  val set_mut : &!\(\alpha\) t \times int \times \alpha \rightarrow 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. 4. Signature and Implementation of hybrid arrays

val propagate : int \rightarrow int \rightarrow &!Matrix.t \rightarrow int \rightarrow unit
val solve : int \rightarrow int \rightarrow Matrix.t \rightarrow unit

Fig. 5. Excerpt of the Sudoku solver

mapping function also takes the position of the element. Recall that Array.map always creates a new array for the result. In the example code, we make use of a two-dimensional version Matrix of the CowArray datastructure. The only difference is that the API functions get, set, and set_mut now take two index parameters of type int instead of one. The internal working is exactly the same.

Our implementation of a Sudoku solver (Fig. 5) represents the board as a 2D-matrix (Line 1). Each cell contains an integer set of type IntSet.t that represents admissible solutions so far. The main functions are solve and propagate with the typings shown on top of Fig. 5. The types of propagate_line etc are the same as for propagate. From the types, we can see that solve takes ownership of the board, whereas propagate only takes a mutable borrow. Hence, the propagate functions can only modify or read the board, whereas solve has full control.

The Sudoku solver solve iterates over the cells and tries each possible solution (Line 17). When a value is picked for the current cell, it creates a choice point in new_board, where the current cell is updated 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_line (Line 3).

The board parameter of the propagate function is an exclusive borrow, it should be handled in an affine manner. To pass it safely to the three helper functions, the body of propagate reborrows (i.e., it takes a borrow of a borrow) the board three times in Lines 11 to 13. The function propagate_line also contains two reborrows of the exclusive borrow board, an immutable one (Line 5) and an exclusive one (Line 7). It demonstrates the facility to take immutable borrows from exclusive ones.

The typing ensures that the mutations do not compromise the state at the choice point, because they operate on a new state new_board 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 `&board`. The price is that `try_solution` cannot escape from `&board`’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 25. As `board` is linear we must free it outside of the region before returning (Line 27).

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) [Puente 2017] or persistent HAMTs [Bagwell 2001; Hickey 2017] 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? Here is a first attempt at the type of a fold function:

```ocaml
val fold : (\alpha:\kappa) => (\alpha \rightarrow \beta k \rightarrow \beta) \rightarrow \alpha Array.t \rightarrow \beta Lin k \rightarrow \beta
```

This type puts no restrictions on the element type of the array, but it requires the `fold` function to consume the array and all its elements. The last function arrow is linear because the array type (from Section 2.1) is linear.

If we want to work on borrows of linear and affine resources, then the typing gets more involved because we must make sure those borrows are not leaked in the result. We obtain the following signature for `bfold`, the borrowing fold operation:

```ocaml
val bfold : (\beta:\kappa), (\kappa \leq Lin r) => (\beta Aff r \rightarrow \beta) \rightarrow (\beta Aff r + 1 \rightarrow \beta) \rightarrow &(!\alpha Array.t) \rightarrow \beta k1 \rightarrow \beta
```

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 `&(!\alpha Array.t)` 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 Lin r`. The important part of this constraint is the `r` index, which ensures that the callback cannot return the borrowed argument from the more deeply nested scope. 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 `all_files : File.t Array.t` to compute the sum of their sizes:

```ocaml
let total_size_of_files = bfold (fun f s \rightarrow File.size f + s) files 0
let total_size = total_size_of all_files
```

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

```ocaml
val iter_mut : (\&(!\alpha Array.t) \rightarrow Unit) \rightarrow (\&(!\alpha Array.t) \rightarrow Unit
```

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).
To close this discussion, let's see which primitives are needed to implement functions like `bfold` and `iter_mut`. The naive sequential implementations of both functions boil down to a loop over the index range of the array:

```ml
let rec bfold_helper f a z i =
  if i < 0 then z
  else bfold_helper f && a (f (get_sb && a i) z) (i-1)

let bfold f a z =
  let l = length && a - 1 in
  bfold_helper f && a z l

let rec iter_helper f a i =
  if i < 0 then ()
  else (f (get_eb &&! a i); iter_helper f &&! a (i-1))

let iter_mut f a =
  let l = length &&! a - 1 in
  iter_helper f &&! a l
```

Observe that we are proposing two different primitives

- `get_sb : &@(k, α Array.t) → int → &k (k, α)`
  to get a shared borrow from a shared borrow of an array and

- `get_eb : &!@!(k, α Array.t) → int → &!k (k, α) (* Unsafe! *)`
  to get an exclusive borrow from an exclusive borrow of an array.

These primitives have the same underlying implementation (the same as get from Section 2.1). Their types arise from the intuition that the borrow of a structure should entitle to borrows of its substructures, roughly, the borrow of an array could be considered as array of borrows. However, only `get_sb` is safe: the shared borrow of the array entitles us to obtain shared borrows for the elements in the same region as the shared borrow for the array freezes the array inside the region. This freeze extends to the elements because lifetime of the array fully overlaps with the lifetime of its elements. Considering `get_eb`, we see that we may obtain two different exclusive borrows of the same array element inside a region. Clearly, the exclusive borrow for the element should live in a nested region where the array is not accessible. Hence, the safe alternative is to use a different function for obtaining borrows of elements

```ml
with_eborrow : (β : k), (k ≤ linr) ⇒ &!@!(k1, α Array.t) → int → (affr+1, a) → β
```

Hence, the helper function for `iter_mut` should read like this

```ml
let rec iter_helper f a i =
  if i < 0 then ()
  else (with_eborrow &! a i f; iter_helper f &! a (i-1))
```

which also explains the occurrence of `affr+1` in the type of `iter_mut`.

We conclude that borrows of datastructures create the need for differently typed access functions that are tailored for different use cases. The argumentation whether such an access function is safe is sometimes subtle and gives rise to nonobvious types.

### 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).
Expressions
\[
e ::= c \mid x \mid (e,e') \mid \lambda x.e \mid \text{let } x = e \text{ in } e' \\
| (e,e') \mid \text{match}_\phi x, y = e \text{ in } e' \quad \text{(Pairs)} \\
| \{e\}_{x \mapsto b} \quad \text{(Region)} \\
| &^b x \mid &&^b x \quad \text{(Borrows)} \\
| \text{create} \mid \text{observe} \quad \text{(Resources)} \\
| \text{update} \mid \text{destroy} \\
\]

Types
\[
\tau ::= a \mid a \times a' \mid T \overline{\tau} \quad \text{(ML types)} \\
\mid \tau \to a' \quad \text{(Function types)} \\
\mid \&^b(k, \tau) \quad \text{(Borrowed Type)}
\]

Kinds
\[
k ::= k \mid Q_n \quad \forall n \in \mathbb{N} \cup \{\infty\} \quad \text{(Kinds)} \\
Q ::= U \mid A \mid L \quad \text{(Quality)}
\]

Constrained type and kind schemes
\[
C ::= (k \leq k') \quad \text{(Constraints)} \\
\sigma ::= \forall \kappa \forall (\alpha : k). (C \Rightarrow \tau) \quad \text{(Type scheme)} \\
\theta ::= \forall \kappa. (C \Rightarrow \overline{k} \to k) \quad \text{(Kind scheme)}
\]

Fig. 6. Syntax

3.1 Syntax

Figure 6 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\}_{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 \phi that indicates whether the match operates on borrows (\phi = \&^b) or not (\phi = \text{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 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 \kappa or a constant (linear L, affine A, or unrestricted U) indexed by a nesting level n in \mathbb{N} \cup \{\infty\}.

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

Type schemes \sigma add quantification over kind variables \kappa, 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 \theta 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 \leadsto 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\}_{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 \(\&^A x\) 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:

\[
\lambda a. \text{let } x = (f \&^A a) \text{ in } g (\&^A x); f (\&^U x) (\&^U x) \quad \Rightarrow \quad \lambda a. \text{let } x = (f \&^A a) \text{ in } \left\{ \begin{array}{l} \{g (\&^A x) \}_{\{x \mapsto A\}}; \\
\{f (\&^U x) (\&^U x) \}_{\{x \mapsto U\}} \end{array} \right\} \]

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. In particular, it is essential to place the second borrow around both occurrence of \((\&^U x)\): we want the function to receive both borrows, but its result must not contain any borrow (otherwise it would escape). 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 \(r\):

\[
\text{let } r = \text{ref 0 in } \lambda a. \text{set } r \left\{ \begin{array}{l} \{g (\&^U a)\}; \\
\{f (\&^U a)\}_{\{a \mapsto U\}} \end{array} \right\} \]

The rules allow merging the two regions around the borrows \(\&^U a\). However the explicit annotation indicates that the region should stay around the closure passed as argument of \(\text{set}\). 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 \(\{ \ldots \}_b\) is nested inside \(\{ \ldots \}_p\), 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:

\[
C \vdash \Gamma e : \tau \quad \text{Expression } e \text{ has type } \tau \text{ in environment } \Gamma \text{ under constraints } C.
\]

\[
C \vdash \Gamma \tau : k \quad \text{Type } \tau \text{ has kind } k \text{ in environment } \Gamma \text{ under constraints } C.
\]

\[
D \vdash e C \quad \text{Constraint } D \text{ entails constraint } C.
\]

\[
D \equiv e C \quad \text{Constraints } C \text{ and } D \text{ are equivalent.}
\]
A kind \( k \) is either a kind variable, \( \kappa \), or a constant \( Q_a \). 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 \( A_0 \). Level \( \infty \) refers to an empty region that is infinitely nested. For instance, the constraint \( (\kappa \leq U_\infty) \) indicates that \( \kappa \) must be unrestricted, but can be local to a region. Kinds form a lattice described in Fig. 7. Unrestricted values may be used where affine ones are expected and affine ones are less restricted than linear ones as reflected in \( \text{LAT-UAL} \). Values usable at level \( n \) may be used at any more deeply nested level \( n' \) as defined in the \( \text{LAT-LEVEL} \) 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.
Environments and bindings. Affe controls the use of variables by supporting new modes of binding in type environments \( \Gamma \), as defined in Fig. 8. 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, \([x : \sigma]_b^n\), indicates that \(x\) is earmarked for a borrowing use in a nested region marked with \(b\) but cannot be used directly. A borrow binding, \((x : \sigma)_k\), 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 \textsc{Region} and \textsc{BorrowBinding} in Fig. 10 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 \(\Gamma\), which in turn means that \(k' \leq k\) where \(k'\) is the kind of the variable’s type scheme \(\sigma\). Borrow bindings follow the same rules, but suspended bindings are forbidden in an environment \(\Gamma\) constrained like that. This intuitive explanation is sufficient to understand the \textsc{Var} and \textsc{Ans} rules shown in Fig. 10.

Rule \textsc{Var} looks up the type scheme of the variable \(x\) in the environment \(\Gamma\) and instantiates it with \(\text{Inst}(\Gamma, \sigma)\). Instantiation follows the HM(\(X\)) formulation and takes as input a scheme \(\sigma\) and an environment \(\Gamma\) and returns a constraint \(C\) and a type \(\tau\). The rule also checks that the other bindings in \(\Gamma\) can be safely discarded by imposing the constraint \((\Gamma \setminus \{x\} \leq A_n)\). It enforces that all remaining bindings (except \(x\)) are affine or unrestricted and can therefore be discarded.

Rule \textsc{Ans} ensures the kind annotation on the arrow type \((\tau_2 \rightarrow \tau_1)\) reflects the restrictions on captured variables via the constraint \((\Gamma \leq k)\). If, for instance, any binding in \(\Gamma\) is affine, it gives rise to the constraint \((A_n \leq k)\) 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 \textsc{App} typing rule in Fig. 10 demonstrates how Affe deals with duplication and dropping of values. The splitting \(C \tau_\circ \Gamma = \Gamma_1 \times \Gamma_2\) in the rule decomposes the type environment \(\Gamma\) in two parts, \(\Gamma_1\) and \(\Gamma_2\), which are used to typecheck the components of the application.

Fig. 9 shows the action of splitting rules on single bindings. If \(x\)’s type is unrestricted, rule \textsc{Both} indicates that we can duplicate it. Similarly, unrestricted borrows can be duplicated with rule \textsc{Borrow}. \textsc{Left} and \textsc{Right} rules are always applicable and move a binding either to the left or right environment. The rules \textsc{Susp}, \textsc{Suspb} and \textsc{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 \textsc{Susp} rule works with a full resource, rule \textsc{Suspb} with a borrow, and rule \textsc{Susps} with a suspended binding.

Splitting applies whenever an expression has multiple subexpressions: function applications, let bindings and pairs. In the expression \(\text{let } a = \text{create } 8 \text{ x in } f \{\text{length \& } U a\}_{\{a=U\}} a\), the rule \textsc{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 \textit{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 \([e]_n^{\{x=b\}}\) indicates a region at nesting level \(n\) in which a \(b\)-borrow can be taken of \(x\).

The typing for a region (rule \textsc{Region} in Fig. 10) replaces suspended bindings by borrow bindings (rule \textsc{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 \( (\tau \leq \text{L}_{n-1}) \) ensures that the return type of the region must live at some enclosing, lower level.

As an example, consider the expression \( \text{letfun} (x : \sigma) =_s p \lambda^k y.e \text{ in } e' \) (Poly let) where the suspended binding \( \lambda^k y.e \) is in the environment \( \Gamma' \). When entering the region, rule \text{Region} imposes \( C \vdash_e \Gamma \leadsto \Gamma' \), which defines \( \Gamma' \) corresponding to \( \Gamma \) where the suspended binding is replaced by the borrow binding \( _s p \lambda^k \). To constrain the borrow to this region we impose the constraint \( C \vdash_e (U_\alpha \leq k) \land (k \leq U_{\alpha}) \), which affirms that the borrow is unrestricted, but can only be used in nesting levels \( n \) and higher. Rule \text{Region} also imposes the constraint \( (\tau \leq \text{L}_{n-1}) \), which prevents the borrow, having kind \( k \) of level \( \geq n \), from escaping the region’s body of type \( \tau \).

Pattern matching. Elimination of pairs is done using a matching construct \( \text{match}_\phi x, x' = e_1 \in e_2 \). 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 \( \phi \) indicates if it applies to a pair (\( \phi = \text{id} \)) or a borrow (\( \phi = \&^b \)). If \( \phi = \text{id} \), 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 \( R \tau \) whose content of type \( \tau \) must be unrestricted (\( U_0 \)) and which is equipped with the four operations introduced in Section 3.1:

- create: \( \forall \kappa_\alpha (\alpha : \kappa_\alpha). (\kappa_\alpha \leq U_0) \Rightarrow \alpha \rightarrow R \alpha \)
- observe: \( \forall \kappa \forall \kappa_\alpha (\alpha : \kappa_\alpha). (\kappa_\alpha \leq U_0) \Rightarrow \&^U (\kappa, R \alpha) \rightarrow \alpha \)
- update: \( \forall \kappa \forall \kappa_\alpha (\alpha : \kappa_\alpha). (\kappa_\alpha \leq U_0) \Rightarrow \&^A (\kappa, R \alpha) \rightarrow A \rightarrow \text{Unit} \)
- destroy: \( \forall \kappa \forall \kappa_\alpha (\alpha : \kappa_\alpha). (\kappa_\alpha \leq U_0) \Rightarrow R \alpha \rightarrow \text{Unit} \)

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 [Amin and Rompf 2017; Owens et al. 2016; Siek 2013]. A function eval manipulates the semantic objects defined in Fig. 11. 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 \( \sigma \) and a kind \( k \) indicating their usage restriction (linear, affine, etc) relative to the variables and constraints of \( \sigma \). 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 \( \rho \) are composed of a raw location \( \ell \), which is just a pointer into a store, and a stack of modifiers that indicates the borrows and reborrows that have been taken from \( \ell \). 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 \( \pi \) 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 \( \gamma \) 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 \( \delta \) is a partial map from raw locations to storables. The function \( \text{salloc} : \text{store} \to \text{storable} \) is such that \( \text{salloc} \) \( \delta \) 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 \( \alpha \) sem which combines error reporting and timeout:

```ocaml
1 type \( \alpha \) sem = Error of string | TimeOut | Ok of \( \alpha \)
2 val eval: store \to perm \to venv \to int \to exp \to (store \ast perm \ast 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. 12 and leave the full definition for Appendix F. The definition uses OCaml syntax with extensive pretty printing. The pervasive \texttt{let *} operator acts as monadic bind for the sem monad. The operator \texttt{let *?} converts a boolean argument into success or failure in the monad:

```ocaml
1 let (let*?) : bool \to (unit \to \beta \) sem \to \beta \ sem =
2 fun b f \to if b then f () else Error ("test_failed")
```

The function header of eval checks whether time is up and otherwise proceeds processing the expression.
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
| App (e₁, e₂, sp) →
  let (y₁, y₂) = vsplit y sp in
  let (δ₁, π₁, r₁) = eval δ π y₁ i' e₁ in
  let- t₁ = getloc r₁ in
  let- w = δ₁(t₁) in
  let- (y', k', x', e') = getstcls 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' y₂ i' e₂ in
  Ok (δ₂, π₂, r₂)
| Varinst (x, \overline{k}) →
  let- rx = y(x) in
  let- ℓ = getloc rx in
  let-? () = ℓ ∈ π in
  let- w = δ(ℓ) in
  let (y', k', C', k', x', e') = getstpoly w in
  let π' =
    if C'(\overline{k}\rightarrow\overline{r}) =_e [(k' ≤ U)(\overline{k}\rightarrow\overline{r})]
     then π else π − ℓ
  in
  let w = STCLOS (y', k' \overline{k}\rightarrow\overline{r}, x', e'(\overline{k}\rightarrow\overline{r})) in
  let (ℓ', δ') = alloc δ w in
  Ok (δ', π' + ℓ', ℓ')
| Region (e, n, x, rₓ, b) →
  let- ρ = y(x) in
  let- ρ' = b.ρ in
  let π' = reach ρ rₓ δ in
  let π'' = b.π' in
  let y' = γ(x → ρ') in
  let π = (π ∪ π'') \ π' in
  let (δ₁, π₁, r₁) = eval δ π y' i' e in
  let π₁' = (π₁ \ π'') ∪ π' in
  Ok (δ₁, π₁, r₁)

Fig. 12. Big-step interpretation

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 e₁ 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 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 [Odersky et al. 1999], 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 $C_L$. We adapt and extend HM($X$)'s rules to support kind inference, track linearity, and handle borrows and regions. We formulate constraint solving and simplification algorithms for $C_L$. Finally, we prove that the inference algorithm computes principal types.

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 green**. 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 $\tau$, which is of kind $k$, we add the constraint $(k \leq U)$. To infer such constraints, our inference judgment not only takes an environment as parameter but also returns a *usage environment*, denoted $\Sigma$, 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.

- $C \triangleleft \Sigma_1 = \Sigma_2 \uplus \Sigma_3$. Given two usage environments $\Sigma_1$ and $\Sigma_2$, we return $\Sigma$, the merged environment, and $C$, a set of constraints that must be respected.
- $C \triangleleft \Sigma \rightsquigarrow \Sigma'$. Given a usage environment $\Sigma'$, a nesting level $n$, and a variable name $x$, we return $\Sigma$ where the borrow binding of $x$ in $\Sigma'$, 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.

**Constraint Normalization.** The HM($X$) framework assumes the existence of a function “normalize” which takes a constraint $C$ and a substitution $\varphi$ and returns a simplified constraint $C'$ and an updated substitution $\varphi'$. Normalization returns a normal form such that $\varphi'$ 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 $\Sigma \vdash (C, \psi) \mid \Gamma \vdash_w e : \tau$ when $e$ has type $\tau$ in $\Gamma$ under the constraints $C$ and unifier $\psi$ with a usage environment $\Sigma$. $\Gamma$ and $e$ are the input parameters of our inference algorithm. Unlike in the syntax-directed version, $\Gamma$ contains only regular and type bindings. Suspended and borrow bindings can only be present in $\Sigma$. 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. 13. The complete type inference rules are shown in Appendix E.

**Environments and Bindings.** In the syntax-directed system, the $\text{VAR}$ 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 $\text{VAR}_I$ and $\text{ABS}_I$ 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 $\sigma$ 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 $\text{ABS}_I$
rule. First, we typecheck the body of the lambda and obtain a usage environment \( \Sigma_x \). As in the syntax-directed type system, we introduce the constraint \( \Sigma \{x \leq \kappa\} \) which properly accounts for captures in the body of the lambda expression. We then introduce the constraint \( \text{Weak}_{(x:a)}(\Sigma) \), which fails if we try to abandon a linear variable. The Weak constraint is introduced at each binding 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 \( \Sigma \). As such, we use the “inference” version of the relations on the environment, \( C \equiv \Sigma = \Sigma_1 \bowtie \Sigma_2 \) and \( C \equiv \Sigma \bowtie \Sigma_n \), which returns the necessary constraints. We then collect all constraints and normalize them.

### 4.3 Constraints

To properly define our type system, we need to define \( C_L \), a constraint system equipped with an entailment relation noted \( \vdash_e \) and a normalizing function. For concision, we first demonstrate the constraint solving algorithm with an example. We then state the various properties that make it suitable for use in the HM(X) framework. The complete constraint system is defined in Appendix C.

#### 4.3.1 Constraints Normalization by Example

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

\[
\Gamma = (\alpha_f : \kappa_f)(\alpha_x : \kappa_x) \ldots
\]

\[
C = (\alpha_f \leq \gamma \frac{\kappa_i}{\kappa_j} \beta) \land (\gamma \leq \alpha_x) \land (\beta \times \alpha_x \leq \alpha_r) \land (\kappa_x \leq U)
\]

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

\[
(\gamma \frac{\kappa_3}{\kappa_2}) \xrightarrow{\kappa_i} \gamma \xrightarrow{\kappa_j} \beta \times \gamma
\]

In addition, we obtain the following kind constraints:

\[
(\kappa_x \leq U) \land (\kappa_y \leq \kappa_x) \land (\kappa_x \leq \kappa_r) \land (\kappa_f \leq \kappa_y) \land (\kappa_3 \leq \kappa_f) \land (\kappa_f \leq \kappa_1)
\]
We translate these constraints into a relation whose graph is shown in Fig. 14. The algorithm then proceeds as follow:

- From the constraints above, we deduce the graph shown with plain arrows on the left of Fig. 14.
- 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 $\kappa_\gamma = \kappa_x = U$.
- We take the transitive closure, which adds the arrow in blue from $\kappa_3$ to $\kappa_1$.
- 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

$\kappa_\gamma = \kappa_x = U \land \kappa_3 \leq \kappa_1$

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

$$\forall \kappa_\beta \kappa_1 \kappa_2 \kappa_3 (\gamma : U) (\beta : \kappa_\beta). (\kappa_3 \leq \kappa_1) \Rightarrow (y \to \beta) \to \gamma \to \beta \times \gamma$$

We can further simplify this type by exploiting variance. As $\kappa_1$ and $\kappa_2$ are only used in covariant position, they can be replaced by their lower bounds, $\kappa_3$ and $U$. By removing the unused quantifiers, we obtain a much simplified equivalent type:

$$\forall \kappa (\gamma : U). (\gamma \to \beta) \to \gamma \to \beta \times \gamma$$

4.3.2 Properties of the Constraint System. To apply HM($X$) to $C_L$, normalize must compute principal normal forms and $C_L$ must be regular.

**Property 1** (Principal normal form). Normalization computes principal normal forms for $C_L$, i.e., given a constraint $D \in C_L$, a substitution $\phi$ and $(C, \psi) = \text{normalize}(D, \phi)$, then $\phi \leq \psi, C = \psi D$ and $\psi C = C$.

**Property 2** (Regular constraint system). $C_L$ is regular, i.e., for $x, x'$ two types or kinds, $\vdash_c (x = x')$ implies $\text{fv}(x) = \text{fv}(x')$.

These properties are sufficient to state that $HM(C_L)$ provides principal type inference. The next section shows that these properties carry over to the inference algorithm for our extension of HM($X$) with kind inference, affine types, and borrows. This algorithm includes sound and complete constraint simplification. In addition, we may add “best-effort” simplification rules which help reduce the size of inferred signatures [Simonet 2003b].
4.4 Soundness and Principality

The extended inference algorithm is sound and complete with respect to our extension of HM(X). The first theorem states that inference is sound with respect to the syntax-directed type system.

**Theorem 4.1 (Soundness of inference).** Given a type environment \( \Gamma \) containing only value bindings, \( \Gamma|_r \) containing only type bindings, and a term \( e \):

\[
\text{if } \Sigma, (C, \psi) \mid \Gamma, \Gamma_r \vdash_{tr} e : \tau \\
\text{then } C \mid \psi(\Sigma; \Gamma_r) \vdash_{s} e : \tau, \psi C = C \text{ and } \psi \tau = \tau
\]

The syntax-directed derivation holds with the usage environment \( \Sigma \) instead of the originally provided environment \( \Gamma \). Indeed, \( \Gamma \) does not contain suspended and borrow bindings. Those are discovered on the fly and recorded in \( \Sigma \). Type bindings are taken directly from the syntax-directed derivation.

The second theorem states that inference 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 \( \sigma = \forall \alpha D \Rightarrow \tau \) and \( \sigma' = \forall \alpha D' \Rightarrow \tau' \). Then \( C \vdash_{s} \sigma \leq \sigma' \) iff \( C \vdash_{s} D[\alpha \rightarrow \tau''] \) and \( C \land D' \vdash_{s} (\alpha \rightarrow \tau'') \leq \tau' \).

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

\[
\downarrow \Gamma = \{ (x : \tau) \in \Gamma \mid \forall b : \& b(k, \tau) \in \Gamma \lor [x : \tau]_{b} \in \Gamma \} \cup \{ (\alpha : k) \mid (\alpha : k) \in \Gamma \}
\]

**Theorem 4.4 (Principality).** Let \( \text{True} \mid \Gamma \vdash_{s} e : \sigma \) a closed typing judgment. Then \( \Sigma, (C, \psi) \mid \downarrow \Gamma, \Gamma_r \vdash_{tr} e : \tau \) such that:

\[
(\text{True}, \sigma_0) = \text{gen}(C, \psi, \tau), \quad \tau \vdash_{s} \sigma_0 \leq \sigma
\]

5 METATHEORY

There are several connections between the type system and the operational semantics, which we state as one type soundness theorem. The theorem relies on several standard notions like store typing \( \vdash : \Delta \) 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 getloc(\( \pi \)) we extract the underlying raw locations from the permissions as in getloc(\( \overline{U \ A } \ell \)) = \( \ell \) and with reach\( \gamma \) we transversely trace the addresses reachable from \( y \) in store \( \delta \). 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 \( \gamma \) that correspond to the different kinds of bindings in the type environment: \( \gamma^A \) for active entries of direct references to linear resources, closures, etc; \( \gamma^U \) for affine borrows or resources; \( \gamma^U \) for unrestricted values including unrestricted borrows; and \( \gamma^U \) for suspended entries. The judgment \( \Delta \vdash \gamma : \Gamma \) is defined in terms of this structure. We treat reach\( \gamma \) 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

\[
(A1) C \mid \Gamma \vdash_{s} e : \tau \\
(A2) \Delta \vdash \gamma : \Gamma \\
(A3) \vdash_{s} \delta : \Delta \\
(A4) \pi \text{ is wellformed and } \text{getloc}(\pi) \subseteq \text{dom}(\delta) \setminus \delta^{-1}(\bullet) \\
(A5) \text{reach}_0(\gamma) \subseteq \pi, \text{reach}_\gamma(\gamma) \subseteq \downarrow \pi.
\]

Proc. ACM Program. Lang., Vol. 4, No. ICFP, Article 103. Publication date: August 2020.
(A6) getloc(\(y^L\)), getloc(\(y^A\)), getloc(\(y^U\)), and getloc(\(y_s\)) are all disjoint

(A7) Incoming Resources:
(a) \(\forall \ell \in \text{getloc}(\text{reach}_\delta(\gamma)), \delta(\ell) \neq \bullet\).
(b) \(\forall \ell \in \Theta = \text{getloc}(\text{reach}_\delta(\gamma^L, y^A, y^A)), \Theta(\ell) = 1\).

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

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

(R2) \(\Delta \leq \Delta', \delta \leq \delta', \lor \delta' : \Delta'\)

(R3) \(\Delta' + r' : \tau\)

(R4) \(\pi'\) is wellformed and getloc(\(\pi'\)) \(\subseteq \text{dom}(\delta') \setminus \delta'^{-1}(\bullet)\).

(R5) reach_0(r') \(\subseteq \pi', \text{reach}_\delta(r') \subseteq \downarrow \pi' \cap (\text{reach}_\delta(\gamma) \setminus \text{reach}_\delta(\gamma_s) \cup \text{dom}(\delta') \setminus \text{dom}(\delta))\).

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

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

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

(R9) Resources: Let \(\Theta = \text{reach}_\delta(\gamma^L)\). Let \(\Theta' = \text{reach}_\delta(\gamma^L)\).
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:
\(\pi' \not\subseteq \pi \cup (\text{dom}(\delta') \setminus \text{dom}(\delta))\).

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 LIMITATIONS AND EXTENSIONS

6.1 Flow Sensitivity

The type system defined so far does not support any form of flow sensitivity. Therefore, code patterns that rely on subtle flow-sensitive usage of permissions and linearity will most likely not typecheck in Affe. For example, the following merge function on linear lists cannot be expressed directly, because matching against \(\underline{11}\) and \(\underline{12}\) consumes both lists.
Patterns like this require a richer logic, such as provided by Mezzo [Protzenko 2014]. However, Weiss et al. [2019] formalize Rust’s notion of non-lexical lifetimes which partially supports such code patterns. We believe this notion can be adapted to Affe’s notion of regions.

**Non-Lexical Regions.** The notion of non-lexical lifetimes is a recent addition to Rust. With this feature code is acceptable even if borrowing does not respect lexical scoping as in this example:

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

This code pattern is dynamically safe because \( a \) is not used after the function call \( f \ a \). Here, this can be made explicit by transforming the code to \( (\text{let } a = &x \text{ in } f \ a); g &!x \). However, this is not possible in programs with branches who uses different dynamic patterns. Non-lexical lifetimes (NLL) handle such a pattern by removing expressions that do not mention \( a \) from its region; in this example, NLL removes the last expression. In Affe, regions are lexical and marked by the expression \( \{ | e | \}^\alpha_\beta \). During inference, kind constraints 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 expressions would be annotated to be “in the region associated with \( &x \)”, but not the last expression. Thanks to these annotations, type checking the sequence would check that the borrow does not escape the left-hand side (i.e., the second expression \( f \ a \)).

**6.2 Capabilities and Identity**

In Affe the tracking of linearity does not rely on any notion of “identity”: the type system cannot specify that two objects are the same, simply that they share the same usage pattern with regards to linearity. A language like Alms [Tov and Pucella 2011], on the other hand, often relies on a notion of identity to express capabilities. For instance, the Alms typing \( \text{Array.create : } \text{int} \to \alpha \to \exists \beta. (\alpha, \beta) \text{ array} \) uses \( \beta \) as a unique identification of the array. Functions such as \( \text{Array.acquire : } (\alpha, \beta) \text{ array} \to \beta \text{ cap} \) are used to obtain capabilities to operate on the array.

While such uses are partially covered by borrows and regions, a notion of identity associated to regions would enable us to express regions directly in type signatures. For instance, the get_eb function shown in Section 2.3 could be made safe by creating a restricted inner region on function application, with the signature: \( &! (k, \alpha \text{ Array.t}) \to \text{int} \to \exists (k' < k) &! (k', \alpha) \).

This approach relies on existential types to model identities. At present, Affe does not support existentials as it would forgo principal type inference. However, existentials are compatible with the HM(X) framework [Simonet 2003a] and would make a very desirable addition to Affe. Work on GADTs in OCaml and Haskell demonstrates that existential types can be put to use without compromising inference in the rest of the language, by integrating unpacking and pattern matching.

**6.3 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 [Odersky et al. 1995].

### 6.4 A Richer Region System

Affe requires that each region is identified by an index drawn from a partial order that is compatible with the nesting of regions. This order can be implemented in many ways, including region variables as often used in algebraic effects systems, existentials, etc.

For simplicity, the formalization uses the concrete implementation with natural numbers for indices. The proofs only rely on the existence of a partial order and could be adapted to one of the more abstract approaches. In particular, Affe could reuse regions variables provided by the ongoing work on effect systems for OCaml [Dolan et al. 2017].

### 6.5 Standard Features

**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. Our prototype implements non-recursive algebraic datatypes with pattern matching.

**Branching Constructs.** Our formalization of Affe does not cover conditionals. In the typing rules, a conditional is supported as usual by checking all branches with the same constraint and typing environment and requiring the return types to match. It materializes in the inference algorithm as a straightforward (symmetric) join relation which is used for all elimination rules on sum types. This extension is implemented in our prototype.

### 6.6 Concurrency

While our present semantic model does not consider concurrency, some design decisions were taken with a possible extension to concurrency in mind. The main impact of this foresight is the distinction between exclusive borrows and shared borrows, which materializes in the metatheory. The intended contract of the shared borrow is that it can be duplicated and that the program consistently observes the same state of the underlying resource inside its region even in the presence of concurrency.

The exclusive borrow, on the other hand, is propagated according to the evaluation order with the intention that any suspended binding split of from an exclusive borrow has finished its action on the resource before the borrow gets exercised. In the presence of concurrency, this intended semantics of the exclusive borrow should guarantee freedom of data races.

That is, if a thread closes over a borrow, that thread should have terminated before the parent thread leaves the borrow’s region. Rust addresses this lifetime issue with the move qualification for a thread which transfers ownership of the free variables to the thread. However, moving (in Rust) only applies to the resource itself, but not to borrows. A more discerning kind system would be needed for Affe to enable safe sharing of synchronizable resources or borrows analogously to Rust’s Sync trait.
### 7 RELATED WORK

The comparison matrix in Fig. 15 gives an overview over the systems discussed in this section. Each column indicates whether a feature is present (✔), absent (✘), or partially supported (~), i.e., if the feature is limited or can only be obtained through a non-trivial encoding. Features are selected according to their relevance for type-based resource management and programmer convenience.

The column UAL specifies the substructural features supported (Unrestricted, Affine, Linear). The columns “State” and “Borrows” indicate support for the respective feature. In an ideal world, the presence of linearity and state indicates that the system is able to support safe manual memory management as linearity enforces manual deallocation. True affinity and state only works with garbage collection, which eventually automatically finalizes an object no longer referenced. In practice, this distinction is often watered down. For example, Rust automatically destructs objects at the end of their lifetime, creating the illusion of affinity while the low-level code is strictly linear. However, there are ways to consume an object at the source level without invoking its destructor (using `mem::forget`) where the high-level code exhibits linearity, but the low-level code is affine.

“Multiplicity Subsumption” indicates that unrestricted elements can be promoted to affine and then linear. This promotion applies to objects, resources, borrows, and closures. "Multiplicity Polymorphism" refers to polymorphism over substructural features: a function can be parameterized over the multiplicity restriction of an object. For instance, the type of function composition should express that applies to functions with linear, affine, and unrestricted multiplicity and returns a function with the same multiplicity. “Identity” indicates that the language supports a notion of identity, usually through existential types, as described in Section 6.2. “Concurrency” indicates whether the language supports concurrency. For example, the implementation of Linear Haskell supports state and concurrency, but its theory covers neither. “Escape hatch” indicates whether a programmer can (locally) opt out of resource management through language-integrated means such as Rust’s `unsafe`. Partial support (~), in the case of Affe for instance, indicates that this feature is available, but not formalized. Type “inference” can be local, principal, or non-principal (if the

---

**Fig. 15. Comparison matrix**

| Language | UAL | State | Borrows | Multiplicity Subsumption | Multiplicity Polymorphism | Identity | Concurrency | Escape hatch | Inference | Formalisation | Basis |
|----------|-----|-------|---------|--------------------------|---------------------------|----------|-------------|-------------|-----------|---------------|-------|
| System F° | UL  | ✘     | ✘       | ✘                       | ✘                         | ✘        | ✘           | ✘           | ✘         | Coq           | System F |
| Alms     | UA  | ✔     | ~       | ✔                       | ✔                         | ✔        | ✔           | ✔           | ✔         | Local         | ML    |
| Quill    | UL  | ✘     | ✘       | ✘                       | ✘                         | ✘        | ✘           | ✘           | ✘         | Principal     | Manual Qual. types |
| Lin. Haskell | UL | ~     | ✘       | ✔                       | ✘                         | ✔        | ~           | ~           | ✔         | Non-pr. Manual | Haskell |
| Mezzo    | UA  | ✔     | ~       | ✔                       | ✔                         | ✔        | ✔           | ✔           | ✔         | Local         | Coq   |
| Rust     | UA  | ✔     | ✘       | ✘                       | ✘                         | ✔        | ✘           | ✘           | ✘         | Local         | Coq   |
| Plaid    | UA  | ✔     | ✘       | ✘                       | ✔                         | ✔        | ✘           | ✘           | ✘         | Manual        | Java   |
| Affe     | UAL | ✔     | ✔       | ✔                       | ✘                         | ✘        | ✘           | ✘           | ✘         | Principal     | Manual ML/HM(X)  

---

3See [https://doc.rust-lang.org/nomicon/leaking.html](https://doc.rust-lang.org/nomicon/leaking.html) which contains further examples and discussion. Thanks to Derek Dreyer and Ralf Jung for pointing this out.
inferred type is not necessarily the most general one). “Formalization” refers to the existence of a formal semantics and type soundness proof. “Basis” indicates the heritage or inspiration of the language.

### 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°** [Mazurak et al. 2010] 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** [Morris 2016] 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** [Tov and Pucella 2011] 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, consider the signature `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 `ΠαΠβ. ⟨α⟩ ⊔ ⟨β⟩` (where α and β are types and Π is the dependent function) in Alms compared to `κ → κ → κ` 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 [Bernardy et al. 2018] (LH) is to retrofit linear types to Haskell. Unlike the previously discussed approaches, LH relies on “linear arrows”, written `→` 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\(^4\). 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 [Matsuda 2019] formalizes the inference for rank 1 qualified-types. However, there is an implementation of the inference as part of GHC.

---

\(^4\)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](https://github.com/ghc-proposals/ghc-proposals/pull/111#issuecomment-403349707).
• LH promotes a continuation-passing style with functions such as \( \text{withFile : path } \rightarrow (\text{file } \rightarrow \text{Unrestricted file} \rightarrow \text{r}) \) 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 [Balabonski et al. 2016; Protzenko 2014] is an ML-like language with a rich capability system which is able to encode numerous properties akin to separation logic [Reynolds 2002]. 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 [2018] 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.

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 [Boyland and Retert 2005; Grossman et al. 2002].

Rust [Matsakis and Klock II 2014] is the first mainstream language that builds on the concepts of borrowing and ownership to enable 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. Everything is affine in Rust and marker traits like \( \text{Copy} \), \( \text{Send} \), and \( \text{Sync} \) are used to modulate the characteristics of types. Affe relies on kinds to express substructural properties of types and marker traits may be considered as implementing a fine-grained kind structure on Rust types. Rust’s lifetime system is more explicit and more expressive than Affe’s regions. While Rust provides partial lifetime inference, it does not support full type inference. Moreover, Rust programmers have full control over memory allocation and memory layout of objects; they can pass arguments by value or by reference. These features are crucial for the efficiency goals of Rust. In contrast, Affe is garbage collected, assumes a uniform object representation, and all arguments are passed by reference. This choice 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.

In Rust, programmers can implement their low-level abstractions by using unsafe code fragments. Unsafe code is not typechecked with the full force of the Rust type system, but with a watered down version that ignores ownership and lifetimes. This loophole is needed to implement datastructures like doubly-linked lists or advanced concurrency abstractions. When unsafe code occurs as part of a function body, the Rust typechecker leaves the adherence of the unsafe code to the function’s type signature as a proof obligation to the programmer. The RustBelt project [Jung et al. 2018] provides a formal foundation for creating such proofs by exhibiting a framework for semantic soundness of the Rust type system in terms of a low-level core language that incorporates aspects of conccurrency (i.e., data-race freedom). Similar proof obligations would be needed in Affe to check that an implementation of the module types or the type of fold shown in Section 2 matches the semantics of the typings. We aim to develop a suitable framework for this task for Affe. At present, the metatheory of Affe does not cover concurrency.
Weiss et al. [2019] formalize Rust’s ownership discipline from a source-level perspective. Their approach is purely syntactic and is therefore not able to reason about unsafe fragments of Rust code. However, their flow-sensitive type discipline enables soundness proofs for non-lexical lifetimes, which have been adopted in Rust, but cannot be expressed in Affe at present.

Vault [DeLine and Fähndrich 2001] and Plaid [Aldrich et al. 2009; Garcia et al. 2014] 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(\( \mathcal{X} \)) [Odersky et al. 1999] allows us to use constrained types in an ML-like language with complete type inference. HM(\( \mathcal{X} \)) has been shown to be compatible with subtyping, bounded quantification and existentials [Simonet 2003a], GADTs [Simonet and Pottier 2007], and there exists a syntactic soundness proof [Skalka and Pottier 2002]. 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 [Odersky et al. 1999; Trifonov and Smith 1996]. It also admits classical simplification rules [Pottier and Simonet 2002; Simonet 2003b] 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.

ACKNOWLEDGMENTS

This material is based upon work supported by the German Research Council, DFG, project reference number TH 665/11-1. We are indebted to the anonymous reviewers for their thoughtful and constructive comments.
REFERENCES

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/S0956796800000128

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

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

Phil Bagwell. 2001. Ideal Hash Trees.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. ACM Trans. Program. Lang. Syst. 38, 4 (2016), 14:1–14:94. http://dl.acm.org/citation.cfm?id=2837022

Erik Barends 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, Vol. 982), Emanuel V. Hermenegildo and S. Doaitse Swierstra (Eds.). Springer, 189–206. https://doi.org/10.1007/BFb0026821

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

John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adaptation. 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 Martin Abadi (Eds.). ACM, 283–295. https://doi.org/10.1145/1040305.1040329

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

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

Stephen Dolan, Spiro Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10788), Meng Wang and Scott Owens (Eds.). Springer, 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. ACM Trans. Program. Lang. Syst. 36, 4 (2014), 12:1–12:44. https://doi.org/10.1145/2629609

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

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

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

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. Proc. ACM Program. Lang. 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In Behavioral Types: From Theory to Tools, Simon Gay and Antônio Ravara (Eds.). River Publishers. 2017. Concurrent System Programming with Effect Handlers. In Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10788), Meng Wang and Scott Owens (Eds.). Springer, 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

Nicholas D. Matsakis and Felix S. Klock II. 2014. 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
Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. CoRR abs/1803.02796 (2018). arXiv:1803.02796 http://arxiv.org/abs/1803.02796

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

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, FP'95, La Jolla, California, USA, June 25–28, 1995, John Williams (Ed.), ACM, 135–146. https://doi.org/10.1145/224164.224195

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.

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

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

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

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

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

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

Vincent Simonet. 2003a. 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

Vincent Simonet. 2003b. 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, Vol. 2895), Atsushi Ohori (Ed.). Springer, 283–302. https://doi.org/10.1007/978-3-540-40018-9_19

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

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

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

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, Vol. 1145), Radhia Cousot and David A. Schmidt (Eds.). Springer, 349–365. https://doi.org/10.1007/3-540-61739-6_52

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