An introduction to Mezzo

François Pottier
INRIA
Francois.Pottier@inria.fr

Jonathan Protzenko
INRIA
Jonathan.Protzenko@inria.fr

Abstract

We present the design of *Mezzo*, a programming language in the ML tradition, which places strong emphasis on the control of aliasing and access to mutable memory. A balance between simplicity and expressiveness is achieved by marrying a static discipline of permissions and a dynamic mechanism of adoption and abandon.

1. Introduction

Programming with mutable, heap-allocated objects and data structures is difficult. In many typed imperative programming languages, including Java, C#, and ML, the type system keeps track of the structure of objects, but not of how they are aliased. As a result, a programming mistake can cause undesired sharing, which in turn leads to breaches of abstraction, invariant violations, race conditions, and so on. Furthermore, the fact that sharing is uncontrolled implies that the type of an object must never change. This forbids certain idioms, such as delayed initialization, and prevents the type system from keeping track of the manner in which objects change state through method calls. In order to work around this limitation, programmers typically use C# and Java's null pointer, or ML's option type. This implies that a failure to follow an intended protocol is not detected at compile time, but leads to a runtime error.

In short, there is a price to pay for the simplicity of traditional type systems:

- bugs caused by undesired sharing, or by the failure to follow an object protocol, are not statically detected;
- the dynamic checks that must often be inserted in order to satisfy the type-checker have a runtime cost.

This paper presents the design of a new programming language, code-named *Mezzo*, which attempts to partly address these issues. *Mezzo* is a high-level programming language in the ML tradition. It has algebraic data structures and pattern matching, first-class functions, and implicit memory management. Like ML, it blends functional and imperative programming. Unlike Java, C#, and ML, it places strong emphasis on the control of ownership, aliasing, and access to mutable memory.

Our design is influenced by several strands of existing work. Although it is impossible to list all of our sources of inspiration, let us mention type systems based on regions and affine capabilities [11, 23, 30, 12, 15, 6, 7, 14, 10], affine type systems [27], separation logic [22, 21] and separation-logic-based analyses [2], and typestate systems [13, 3, 4, 1]. We borrow many ideas from these papers, and make the following main contributions:

- The system is based entirely on permissions, as opposed to types. A value `x` does not inherently "have a type". Instead, at each program point, `x` may be referred to by zero, one, or more permissions, which tell us what `x` is (say, a linked list) and what we are allowed to do with it (say, read and modify the list's spine, but not its elements). The permissions that describe `x` may change with time. Permissions have structure: a compound permission can be decomposed into multiple simpler permissions, out of which other compound permissions can be constructed.
- The system is designed with simplicity in mind. There are only two basic *modes*, or kinds of permissions: some permissions are *duplicable*, while others are *exclusive*. A permission that grants read access to an immutable object is a typical example of a duplicable permission: an immutable object has an uncontrolled number of readers. A permission that grants read and write access to a mutable object is a typical example of an exclusive permission: a mutable object is governed by a single read and write permission. We show that, in spite of this relative simplicity, the system is very expressive. In particular, it allows strong (type-varying) updates and can be used to perform typestate checking.
- Modes are viewed as predicates over permissions. This treatment is new and lightweight. For instance, many of the polymorphic functions that operate on containers, such as the `length` and `map` functions for immutable lists, are indifferent to whether the elements are duplicable or exclusive, while others, such as the shallow copy function for mutable lists, require the elements to be duplicable. This requirement is expressed via a *mode constraint*.
- An object can "adopt" a number of other objects, and can "abandon" an object that it has adopted. We keep track *at runtime* of the relationship between adoptees and adopters, and allow abandon to fail at runtime. In return, we obtain a simple and flexible discipline for managing

groups and permissions over groups. This mechanism allows building and working with mutable data structures that are not forest-shaped.

The system takes advantage of algebraic data structures and pattern matching, and "feels" very much like ML. It does not have global type inference: instead, it has intra-procedural permission inference. Function signatures must be provided by the programmer.

The system is meant to be sound: well-typed programs cannot go wrong. Still, a program can stop abruptly if the "abandon" operation, which involves a runtime check, is mis-used. In a shared-memory concurrent setting, the system guarantees mutual exclusion (the absence of race conditions) between threads.

This paper presents the design of *Mezzo* in an informal manner. As much as possible, we rely on examples and write in a tutorial style. At present, the design is stable, but still in a preliminary state. Several details, including the concrete syntax, are not quite set in stone yet. The code-name *Mezzo* itself is temporary. A prototype implementation of the type-checker is in the works, but is not complete: the examples presented in this paper have been manually type-checked. We envision compiling *Mezzo* down to untyped OCaml code, which can then be translated into native code by the OCaml compiler. This has not yet been implemented. A formal definition of the system, as well as a formal proof of its soundness, are work in progress and should be the subject of a forthcoming paper.

The paper is organized as follows. We present permissions and explain how they describe heap-allocated data structures (§2). Then, we introduce tuple types and function types, which, together, are used to describe functions (§3). In order to illustrate the power of these basic notions, we show how to implement a number of standard operations on immutable and mutable lists (§4). In order to support mutable data structures that involve sharing, we propose groups, adoption, and abandon (§5). We briefly discuss a few features that we would like to include in the future (§7) and conclude.

2. Permissions

2.1 Simple permissions

At the heart of the system is the notion of a *permission*. A permission takes the form "`x @ t`", where `x` is a variable and `t` is a type. Such a permission plays three simultaneous roles:

- it describes the value denoted by the variable `x` as well as a (possibly empty) part of the heap that is accessible through `x`;
- it entitles *us* to perform certain read or write operations on this part of the heap;
- it guarantees that *others* cannot perform certain read or write operations on this part of the heap.

Here, "we" can be informally understood as "the software component that we are implementing", whereas "others" means "other components in the system". In a concurrent setting, "we" can additionally be understood as "the current thread", whereas "others" means "other threads in the system". By controlling who is allowed to do what, permissions ensure that the components of a software system cooperate in a safe manner, without undesired interference.

In keeping with the ML tradition, variables are immutable. Only heap-allocated objects can be modified.

Permissions do not exist at runtime. They are used by the type-checker as a way of keeping track of which operations are permitted at each point in the code. Permissions are typically explicitly mentioned by the programmer as part of function signatures. They are not explicitly mentioned in the code: they flow implicitly.

It is tempting to interpret the permission "`x @ t`" as an assertion that "`x` has type `t`". This interpretation is acceptable, provided one is well aware of a number of differences between permissions and traditional type assertions. In most typed programming languages, if `x` has type `t` when it is introduced, then `x` has type `t` forever: the type of a variable does not vary with time. Permissions, on the other hand, come and go. We may, for instance, possess a permission "`x @ t`" at a certain point in the code; at a later point, lose this permission, so we no longer have any permission for `x`; and, later yet, recover a different permission "`x @ u`". We could also, at a certain point in the code, possess multiple distinct permissions for a single variable `x`. In short, there is no such thing as "the" type of `x`.

Let us now give a few simple examples of permissions. More elaborate forms of permissions are introduced later on (§2.4, §2.5, §2.6).

Integer A very simple permission is "`x @ int`", which tells us that the variable `x` denotes an integer value. This allows using `x` as an operand in an arithmetic operation.

Unknown An even simpler permission is "`x @ unknown`", which tells us nothing. This permission does not allow us to do anything with `x` except copy it around. In fact, this permission is just as good as no permission at all.

Immutable memory block Suppose we wish to manipulate points in two-dimensional space with integer coordinates. We can make the following algebraic data type definition:

```
data point =
  Point { h: int; v: int }
```

This type describes a heap-allocated object with two integer fields named `h` and `v`. Thus, the permission "`x @ point`" guarantees that `x` denotes an address in the heap and that, at this address, there exists an object whose fields contain integer values. This permission allows us to read `x.h` and `x.v`, and to treat the result as an integer value. It does not allow us to modify `x.h` or `x.v`, and it guarantees that no one else can modify them either. In other words, points are *immutable*.

Mutable memory block Suppose now that we wish to define a variant of the type `point` whose fields are modifiable. We can make the following algebraic data type definition:

```
exclusive data xpoint =
  XPoint { h: int; v: int }
```

Like `point`, the type `xpoint` describes a heap-allocated object with two integer fields named `h` and `v`. Thus, like "`x @ point`", the permission "`x @ xpoint`" guarantees that, at address `x`, there exists an object whose fields contain integer values. However, this permission is strictly stronger than "`x @ point`". Indeed, it allows us to read and write `x.h` and `x.v`, and guarantees that no one else can read or write them. In other words, this permission gives us *exclusive* access to the *mutable* memory block found at address `x`. One can think of it as a token of *ownership*: whoever holds this permission "owns" the memory block at address `x`. It is analogous to a "unique" permission in other systems [3] and to a separation logic assertion [22].

Towards composite permissions The permissions that we have considered so far control access to either zero or one memory block. Naturally, it is often desirable to construct permissions that control a data structure (such as a list, a tree, a hash table, etc.) or a combination of data structures. We will soon present several ways of constructing a composite permission out of simpler permissions, and, conversely, of deconstructing a composite permission so as to recover more elementary permissions (§2.4, §2.5).

2.2 Co-existence of permissions

As mentioned earlier, in general, multiple permissions for a single object `x` may co-exist. However, these permissions are always *consistent* with one another in the following sense:

- their descriptions of the value `x`, and of the part of the heap that is accessible through `x`, must not contradict each other;
- if one of them claims that "others" cannot perform a certain operation on the heap, then the other permissions must not allow "us" to perform this operation.

Let us give a few examples of permissions that can and cannot co-exist.

The permission "`x @ int`" can co-exist with (a copy of) itself. It cannot co-exist with "`x @ point`" or "`x @ xpoint`", because it claims that `x` is an integer value, whereas the latter two permissions claim that `x` is an address in the heap.

The permission "`x @ unknown`" is consistent with every permission. Indeed, it contains no information at all, so it cannot contradict another permission.

The permission "`x @ point`" can co-exist with (a copy of) itself. Indeed, this permission claims that "others" cannot modify `x.h` and `x.v`, and at the same time prevents "us" from modifying these fields.

The permission "`x @ xpoint`" cannot co-exist with itself or with "`x @ point`". Indeed, this permission claims that "no one else can read `x.h` or `x.v`", so it cannot co-exist with a permission that allows reading these fields.

By the last remark, if the permissions "`x @ xpoint`" and "`y @ xpoint`" co-exist, then `x` and `y` must be distinct heap addresses. If "`x @ xpoint`" and "`y @ point`" co-exist, the same conclusion can be drawn. On the other hand, if one holds "`x @ point`" and "`y @ point`", then one cannot tell at compile time whether the heap addresses `x` and `y` are distinct. Thus, permissions sometimes (but not always) carry *separation* information, in the sense of separation logic [22].

2.3 Modes

Duplicable versus exclusive permissions The permissions that we have presented so far can be classified in two disjoint categories, or *modes*.

In the first category, we find the permissions "`x @ int`", "`x @ unknown`", and "`x @ point`". Each of these permissions can co-exist with itself. Furthermore, each of them has the property that two copies of itself are just as good as one copy. That is, holding two copies of such a permission does not provide more information or grant more privileges than possessing just one copy. This implies that, if one holds one copy of such a permission, it is sound to turn it into two identical copies. (This can be useful, for example, if one needs to pass one copy to a callee and keep one copy.) We say that these permissions are *duplicable*. Of course, this property has nothing to do with the variable `x`: it is a property of the types `int`, `unknown`, and `point`. Thus, we say that these types are duplicable.

In the second category, we find "`x @ xpoint`". This permission cannot co-exist with itself, so it is not duplicable. It grants exclusive access to the object found at address `x`. We say that this permission is *exclusive*. We also say that the type `xpoint` is exclusive.

Permissions for immutable objects are always duplicable. (We do not offer fractional permissions [6].) Full read/write permissions for mutable objects are always exclusive. In order to allow mutable objects to be shared in flexible ways, we introduce later on a duplicable permission to perform a dynamic ownership test over a mutable object (§5).

Affine permissions Although the modes "duplicable" and "exclusive" are the two fundamental modes, there are two reasons why a third mode is needed.

First, not all permissions are known to be duplicable or known to be exclusive. For instance, the permission "`x @ a`", where `a` is a type variable (in other words, an abstract type), could turn out to be duplicable if `a` is instantiated with `point` and could turn out to be exclusive if `a` is instantiated with `xpoint`. As long as the type `a` remains abstract, one must treat this permission in a conservative manner. One assumes

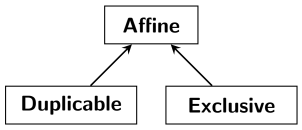


Figure 1. The hierarchy of *modes*

neither that it is duplicable nor that it is exclusive. Thus, this permission cannot be copied, and does not allow adoption and abandon (§5).

Second, as will be explained shortly (§2.4, §2.5), the system allows building composite permissions, some of which are neither duplicable nor exclusive. The permission "`x @ list xpoint`", which describes `x` as (a pointer to) an immutable list of distinct mutable points, is one example. It is not duplicable because it grants exclusive access to the list elements, which are `xpoint` objects. It is not exclusive because it does not grant exclusive access to the object found at address `x` (or, more generally, to the list spine).

A permission that must not be duplicated is said to be *affine*. This is the third mode in the system.

A hierarchy of modes Modes form a hierarchy, which is shown in Figure 1. The hierarchy is simple: "affine" is a strict superset of "duplicable" and "exclusive".

A duplicable permission, such as "`x @ int`", is also affine. An exclusive permission, such as "`x @ xpoint`", is also affine. Some permissions, such as "`x @ a`" (where `a` is a type variable) and "`x @ list xpoint`", are neither duplicable nor exclusive, but are affine.

2.4 Conjunction of permissions

A simple way of building a composite permission consists in putting several permissions side by side. If `p` and `q` are permissions, then "`p * q`", the *conjunction* of `p` and `q`, is also a permission.

Naturally, the conjunction operation `*` is commutative and associative. It is not idempotent: the permission "`p * p`" is equivalent to `p` if and only if `p` is a duplicable permission.

It is convenient to equip conjunction with a neutral element, which we write `empty`. The permission `empty` is trivial: it does not allow anything.

2.5 Hierarchical permissions

A more powerful way of building a composite permission consists in organizing several objects into a hierarchy that is governed by a single permission.

Permissions for pairs The type `point` (§2.1) describes an immutable pair of integer values. Let us now make this notion more abstract and more general. The type "`pair a b`" describes an immutable pair whose left-hand and right-hand components respectively have types `a` and `b`:

```
data pair a b =
  Pair { left: a; right: b }
```

This is a parameterized algebraic data type definition. The type variables `a` and `b` are parameters and can be instantiated with arbitrary types. For instance, the type "`pair int int`", where `a` and `b` are both instantiated with `int`, describes an immutable pair of integers.

One can instantiate `a` and `b` with more complex types. This yields new types of pairs, such as "`pair point point`" and "`pair xpoint xpoint`", which can be used to construct permissions, such as "`x @ pair point point`" and "`x @ pair xpoint xpoint`".

What is the meaning of these permissions? What description of the heap do they offer? Which operations do they allow or deny? In order to answer these questions, one must understand how these permissions can be decomposed into conjunctions of more elementary permissions.

Structural permissions Apparently, the permission "`x @ pair a b`" indicates that `x` is the address of an immutable object whose first and second fields respectively "have type" `a` and `b`. However, we have stressed that it does not really make sense to say that a value "has type" `a` or `b`. Instead, one must think in terms of permissions. In order to explain what a pair permission really means, we posit that the permission:

```
x @ pair a b
```

is equivalent to the following conjunction:

```
x @ Pair { left = l; right = r } *
l @ a *
r @ b
```

provided the variables `l` and `r` are fresh. Thus, "`x @ pair a b`" is really a composite permission. It can be decomposed into (that is, replaced with) the above conjunction of three permissions. Conversely, this conjunction can be replaced with "`x @ pair a b`". Decomposition requires picking fresh variables `l` and `r`, whereas reconstruction does not require that `l` and `r` be fresh.

In the following, variables that are implicitly introduced by the type-checker when unfolding a permission, such as `l` and `r` above, are referred to as *auxiliary variables*, whereas the variables that explicitly appear in the program text are referred to as *program variables*.

The permission "`x @ Pair { left = l; right = r }`" is a *structural permission*, something which we have not encountered yet. This permission describes only one object in the heap. It guarantees that there is an immutable memory block at address `x` and that the values held in the fields `x.left` and `x.right` are exactly `l` and `r`. The permissions "`l @ a`" and "`r @ b`" describe what we are allowed to do with the values `l` and `r`. Their exact meaning depends on the types `a` and `b`. A few examples follow.

Pair of exclusive points By instantiating both `a` and `b` with `xpoint`, we find that the permission "`x @ pair xpoint xpoint`" is equivalent to:

```
x @ Pair { left = l; right = r } *
```

```
l @ xpoint *
r @ xpoint
```

where `l` and `r` are fresh. Because these three permissions co-exist, and because the latter two are exclusive permissions, the addresses `x`, `l`, and `r` must be distinct. Thus, this conjunction of three permissions describes exactly three objects in the heap, and grants exclusive access to two of them. It can be read like thus: "`x` is an immutable pair of distinct mutable points, which we own".

Pair of immutable points An analogous analysis can be carried out about the permission "`x @ pair point point`". This permission is equivalent to:

```
x @ Pair { left = l; right = r } *
l @ point *
r @ point
```

Because the permissions "`l @ point`" are "`r @ point`" are not exclusive, the addresses `l` and `r` are not necessarily distinct. This permission can be read as follows: "`x` is an immutable pair of immutable points".

Exclusive pair One can also define an exclusive version of the `pair` algebraic data type:

```
exclusive data xpair a b =
  XPair { left: a; right: b }
```

The permission "`x @ xpair a b`" is then equivalent to the following conjunction:

```
x @ XPair { left = l; right = r } *
l @ a *
r @ b
```

Everything works in the same manner as in the case of immutable pairs. The only difference is that the first of the three permissions above is exclusive. This is determined by looking up the definition of the data constructor `XPair`.

2.6 Equations and singleton types

Equations The structural permission "`x @ Pair { left = l; right = r }`" refers to three variables, namely `x`, `l`, and `r`. It guarantees that `x` is the address of a memory block, that this block has two fields named `left` and `right`, and that the values stored in these fields are respectively `l` and `r`. Thus, *this permission encodes the equations* `x.left = l` *and* `x.right = r`. Permissions can carry not only structural information and ownership information, but also *must-alias information*. This mechanism is inspired by alias types [23] and by separation logic [22].

Structural permissions (again) and singleton types The permission "`x @ Pair { left = l; right = r }`" plays a double role. On the one hand, it guarantees that there is a pair at address `x`. On the other hand, it encodes the equations `x.left = l` and `x.right = r`. This remark encourages us to view this permission as a combination of two more primitive constructs, namely:

1. *structural* permissions of the following form:

   ```
   x @ Pair { left: t; right: u }
   ```

   where `t` and `u` are arbitrary types.
2. *singleton types* of the form "`=x`", where `x` is a variable.

The one and only value that has type "`=x`" is the value `x` itself. Thus, a permission of the form "`x @ =y`" is logically equivalent to the equation "`x = y`". For this reason, we write "`x = y`" for "`x @ =y`". Similarly, we can now re-interpret the structural permission "`x @ Pair { left = l; right = r }`" as sugar for "`x @ Pair { left: =l; right: =r }`".

We now see that, when we explained how to decompose a permission for a pair, we could have proceeded in two steps. We could have stated that the permission:

```
x @ pair t u
```

is equivalent to:

```
x @ Pair { left: t; right: u }
```

which in turn is equivalent to:

```
x @ Pair { left = l; right = r } *
l @ t *
r @ u
```

where `l` and `r` are fresh. The first step expands the definition of the type `pair` and replaces the *nominal permission* "`x @ pair t u`" with a structural permission. The second step introduces the names `l` and `r` and uses singleton types to indicate that these names stand for the components of the pair.

Properties of equality A singleton type is duplicable. As a result, a permission of the form "`x = y`" is duplicable as well. An equation, once true, is true forever.

Equality is reflexive. Provided `x` is in scope, a permission of the form "`x = x`" can be created out of thin air at any time.

Equals can be substituted for equals. In the presence of an equation "`x = y`", the singleton types `=x` and `=y` are considered equivalent. Furthermore, in the presence of this equation, any permission for `x` can be re-interpreted as a permission for `y`. More precisely, the conjunction "`x @ t * x = y`" is equivalent to "`y @ t * x = y`". These rules imply that equality is symmetric and transitive.

The type-checker is aware of these rules and exploits them transparently. Thus, when an equation "`x = y`" is known to the type-checker, the variables `x` and `y` can be used interchangeably by the programmer. We believe that this mechanism is simpler and more powerful than "borrowing" [20], a mechanism by which permissions are implicitly transferred from one alias to another, without requiring an explicit alias analysis. We do rely on must-alias information, but this information is expressed in terms of permissions, so it can be exposed to and controlled by the programmer.

2.7 Reading and writing

We have stated many times already that certain permissions "allow reading", while others "allow reading and writing", but we have not yet explained how reading and writing are type-checked. Equipped with structural permissions and equations, we can now do this.

Reading Suppose we hold the permission "`x @ pair t u`", where `x` is a program variable, and suppose we wish to gain access to the left component of the pair `x`. We choose to use a field access expression and to bind its value to a program variable `y`. Thus, we write "`let y = x.left in ...`". Is this code legal? What permissions are available at the beginning of the continuation "..."?

In order to answer these questions, the type-checker transparently expands the permission "`x @ pair t u`" into:

```
x @ Pair { left = l; right = r } *
l @ t *
r @ u
```

where `l` and `r` are fresh auxiliary variables. At this point, the structural permission "`x @ Pair { left = l; right = r }`" plays a double role. First, it guarantees that there is indeed a pair at address `x`, so the memory access expression `x.left` is safe. Second, it indicates that the value produced by this expression has already received a name, to wit, `l`. Thus, the expression `x.left` "has type" `=l`, or, in other words, the program variable `y` becomes bound to the value denoted by the auxiliary variable `l`. In order to keep track of this fact, the type-checker adds the equation "`y = l`" to the currently available permissions. Thus, the available permissions at the beginning of the continuation "..." are:

```
x @ Pair { left = l; right = r } *
l @ t *
r @ u *
y = l
```

It is important to remark that the permission "`l @ t`" was not duplicated. Indeed, it is not necessarily duplicable: it could be exclusive or affine. Thus, after the memory access instruction, we still have just one permission for an object that is now known under the names `l` and `y`.

As part of the code in the "...", the programmer can use the variable `y` at type `t`, because the permission "`y @ t`" can be constructed out of "`l @ t`" and "`y = l`". If desired, the programmer can also read `x.left` again and use its value at type `t`. The type-checker considers that this new occurrence of `x.left` produces a value `y'` together with the equation "`y' = l`". Thus, the permission "`y @ t`" can be transformed into "`y' @ t`". In short, even though *permissions* can in general not be copied, *values* can be copied without restriction.

Writing Now, suppose we hold "`x @ xpair t u`", and we wish to update the first component of this pair with a new value `y`, where `x` and `y` are program variables. We write "`x.left <- y`". Is this instruction legal? What permissions are available after it?

In order to answer these questions, the type-checker transparently expands the permission "`x @ xpair t u`" into:

```
x @ XPair { left = l; right = r } *
l @ t *
r @ u
```

At this point, the structural permission "`x @ XPair { left = l; right = r }`" guarantees not only that there exists a pair at address `x`, but also that no one else in the system knows about this pair. This means that we can write into a new value into `x.left` without running the risk of invalidating some other permission that we do not know about. After the update instruction "`x.left <- y`", the state of the heap is described by the following conjunction of permissions:

```
x @ XPair { left = y; right = r } *
l @ t *
r @ u
```

The structural permission that describes the object `x` has been updated: this is a *strong update*.

It is worth noting that the permissions "`l @ t`" and "`r @ u`" do not play any role: they are neither required nor affected by the update.

It is worth noting that no permission for `y` is required. If we happen to have such a permission at hand, say "`y @ t'`", then, after the update, the permissions that describe `x`, `y`, and `r` can be combined to produce "`x @ xpair t' u`". This yields the high-level rule that one might expect: if initially `x` has type "`xpair t u`" and if `y` has type `t'`, then, after the update instruction, `x` has type "`xpair t' u`".

The permission "`l @ t`" remains available after the update. Even though the value `l` is no longer found in the field `x.left`, the programmer may have saved this value, say, by writing "`let z = x.left in ...`" prior to the update. In that case, an equation "`z = l`" is available, and the permission "`l @ t`" allows continuing to use `z` at type `t`.

Tag updates It is possible to update not only a field, but also the tag of a mutable object. The *tag update* instruction "`x <- B`" changes the tag of the object `x` to `B`. This instruction requires an exclusive structural permission of the form "`x @ A { ... }`" and replaces it with a structural permission of the form "`x @ B { ... }`". We provide more details in §4.3 and §5.6, where this feature is exploited.

The data constructors `A` and `B` must have the same number of fields. The fields of `A` need not have the same names as the fields of `B`. The order of the fields in the algebraic data type definitions is used in order to determine how the structural permission must be transformed.

There is no requirement that the data constructors `A` and `B` be associated with the same algebraic data type. If they are associated with different types, say `t` and `u`, the memory block found at address `x` *migrates* from one type to another. This allows a memory block to be "recycled" [24].

Furthermore, there is no requirement that the destination type `u` be exclusive. If `u` is immutable, the memory block at address x *becomes* immutable. This feature enables the delayed initialization of immutable objects. It is illustrated in §4.3.

3. Tuples and functions

We have explained how permissions control access to data. We will soon show that, by building upon these ideas, it is easy to describe and work with algebraic data structures, such as lists (§4) and trees. Before we do this, however, we must introduce functions. *Mezzo*'s function types are somewhat unusual, because functions accept and produce not only values, but also permissions. Because multiple arguments or results are often grouped as a tuple, we begin with a presentation of tuples and tuple types.

3.1 Tuples

Tuples of values A tuple may have zero, one, or more components. A tuple expression has the form "(v1, ..., vn)", where v1, ..., vn are expressions and n is other than one. A tuple type has the form "(t1, ..., tn)", where t1, ..., tn are types and n is other than one. For instance, the pair "(2, 3)" has type "(int, int)". The use of parentheses is in fact optional.

The empty tuple type () is also known as the unit type. It has exactly one inhabitant, namely the empty tuple ().

Tuples are immutable. As a consequence, a tuple type is duplicable if all of its components are duplicable, and is otherwise affine.

A tuple permission can be expanded by the same mechanism that was presented earlier (§2.6). The permission:

```
x @ (t1, ..., tn)
```

is equivalent to:

```
x @ (=x1, ..., =xn) *
x1 @ t1 * ... * xn @ tn
```

where the auxiliary variables x1, ..., xn are fresh. Thus, the tuple permission "x @ (t1, ..., tn)" guarantees that x is an n-tuple and contains a permission for each component.

Packages of a value and a permission It is sometimes necessary to package together a value and a permission. This helps express the types of certain functions that require (or produce) both values and permissions.

For this purpose, we introduce the type "t | p". A value of type "t | p" can be thought of as a package of a value of type t and of the permission p. For instance, the type "(t | y @ u)" describes a value of type t, packaged together with the permission "y @ u". Here, the name y occurs free. Thus, this type makes sense only in a context where some variable y exists. As another example, the type "(t1, t2 | y @ u)" describes a tuple whose two components respectively have types t1 and t2, packaged together with the permission "y @ u".

Permissions do not exist at runtime: they are erased. Thus, when we say that x is a package of a value of type t and of the permission p, this really means that x is a value of type t that has been conceptually packaged together with the permission p. In other words, the permission:

```
x @ (t | p)
```

is equivalent to the following conjunction:

```
(x @ t) * p
```

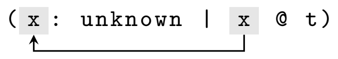
The type-checker automatically applies this equivalence rule, in one direction or the other, when this is appropriate. In particular, if the declared argument type of a function f is "(int, int | p)", then, at the beginning of the body of f, the type-checker considers that the formal argument has type "(int, int)" and that the permission p is available. Symmetrically, if the declared result type of f is "(int, int | p)", then, at the point where f returns, the type-checker verifies that the result value has type "(int, int)" and that p is currently available.

The construction and deconstruction of packages of a value and a permission is always implicit. It is guided by the type information that the programmer provides, including algebraic data type definitions and function signatures.

Dependencies between components The tuple and package types that we have presented up to this point are non-dependent: there is no way for one component to refer to another.

In the type "(int, int | p)", for instance, there is no way for the permission p to refer to the values held in the first and second components of the tuple. Any name that occurs free in p is interpreted as a reference to a variable that exists in the context.

We remove this limitation by allowing a value component of a tuple or package to bind a variable that can be referred to elsewhere. For instance, it is possible to write "(x: int, y: int | p)", where the variables x and y are considered bound in p. As another example, one may write:



Here, the leftmost occurrence of x *binds* this variable, whereas the rightmost occurrence of x is a *free* occurrence.

This type happens to be equivalent to (x: t), which itself is equivalent to t, so it is not very likely that one will ever write this. A more realistic example of the use of dependencies between components might be:

```
(x: unknown, y: unknown | listseg x y)
```

For an appropriate definition of the parameterized permission listseg, this type might represent a tuple of two values x and y, together with the requirement that there exist in the heap a list segment [22] that leads from x to y.

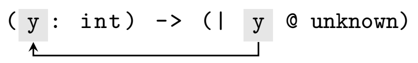
3.2 Functions

Mezzo does not have pervasive type inference in the style of ML. Therefore, we require that every function be explicitly annotated with its type.

Function types are of the form "`t -> u`", where `t` is the type of the argument and `u` is the type of the result. Functions that have multiple arguments or multiple results are encoded as functions that receive or return a tuple. Functions that have no argument or no result are encoded as functions that receive or return an empty tuple.

Function types are duplicable. This means that functions can be shared without restriction and can be invoked as many times as desired.

Dependencies between arguments and results It is often the case that a function returns a permission that refers to one of the function's arguments. In order to allow this, we adopt the following convention. In a function type "`t -> u`", if the type `t` is a tuple type, then the scope of the named components of `t` encompasses not only `t`, but also `u`. For instance, in the following (artificial) example:



```
( y : int) -> (| y @ unknown)
```

the free occurrence of `y` in the result type is interpreted as a reference to the component named `y` in the argument type.

In and out permissions A function that needs access to a data structure usually requires a permission for this data structure to be passed as an argument. (Locks, discussed in §7.2, offer a mechanism for evading this requirement.) There are three typical ways in which the function can deal with this permission:

1. It can return the permission to the caller after it is done. Thus, the caller retains ownership of the data structure. The list length function (§4.1.2) illustrates this.
2. It can choose not to return this permission to the caller. Thus, the caller loses ownership of the data structure. The bag insertion function (§3.5) illustrates this behavior.
3. It can return a different permission to the caller. The caller retains ownership of the data structure, but is aware that a "typestate" change took place. The "pair swap" function (§3.4) is a simple illustration of this behavior. The "tree size" function (§6.1) is a more elaborate one.

Because the first case above is the most common one, we need a concise syntax for it. Drawing inspiration from Sing# [14], we adopt the following convention: *by default, every permission that is passed to a function is returned by this function*. Thus, a function of type "`(x: t) -> (y: u)`" requires the permission "`x @ t`" *and returns this permission* in addition to the permission "`y @ u`". For instance, a function `incr` that increments an integer reference would have type "`ref int -> ()`", which means that the call "`incr r`" requires the permission "`r @ ref int`" and returns it.

Similarly, a function whose type is "`(x: t | p) -> ...`" requires two permissions, namely "`x @ t`" and `p`, and returns them in addition to those described in its result type.

In order to address the cases where the default behavior is not appropriate, we introduce a new keyword, `consumes`. In a function type "`t -> u`", any (value or permission) component of `t` can be preceded with this keyword. In that case, the permission associated with this component is not considered to be returned by this function. For instance, the following type:

```
(consumes x: xpair () int,
 y: xpair int int) ->
(| x @ xpair int int)
```

describes a function that requires two permissions, namely "`x @ xpair () int`" and "`y @ xpair int int`", and also returns two, namely "`x @ xpair int int`" and "`y @ xpair int int`". This function "changes the type" of `x`, possibly by performing the update "`x.left <- y.left`". It requires access to the pair `y`, but does not "change its type".

3.3 Kinds

In order to ensure that types are well-formed, we distinguish different *kinds* of types. There are three kinds: `TYPE`, `PERM` and `TERM`. We have already seen examples of inhabitants of each of them:

- Ordinary types, including "`int`", algebraic data types, singleton types, tuple types, and function types, have kind `TYPE`. They describe values that exist at runtime.
- Permissions, such as "`x @ t`", "`p * q`", and "`empty`", have kind `PERM`. They describe entities that do not exist at runtime.
- Program variables `x`, which occur in singleton types "`=x`" and in permissions "`x @ t`", have kind `TERM`.

The well-kindedness rules include: "a permission "`x @ t`" has kind `PERM` if `x` has kind `TERM` and `t` has kind `TYPE`", "the type "`t | p`" has kind `TYPE` if `t` has kind `TYPE` and `p` has kind `PERM`", and so on. These rules are set up so that every type has at most one kind.

Kinds remain mostly invisible to the programmer: they explicitly appear only when polymorphism (§3.4) is used at a kind other than `TYPE` (which is the default).

3.4 Polymorphism

Like ML, Java, and C#, *Mezzo* has polymorphism. As in Java and C#, polymorphism is explicit: type variables must be explicitly bound. We use square brackets to indicate universal quantification. For instance, is the type of a polymorphic function that swaps the components of a mutable pair:

```
val swap: [a, b]
  (consumes x : xpair a b) ->
  (| x @ xpair b a)
```

The system offers not only type polymorphism, but also permission polymorphism. The need for this feature arises as soon as one wishes to combine higher-order functions and side effects. The list map function (§4.1.4) illustrates this.

In the type of `swap` above, nothing is assumed about the type variables `a` and `b`. They can later be instantiated with

```
type bag :: TYPE -> TYPE
fact [a] exclusive (bag a)
val create: [a] () -> bag a
val insert: [a] (consumes a, bag a) -> ()
val retrieve: [a] bag a -> option a
```

Figure 2. A signature for bags

arbitrary types. For this reason, `a` and `b` must be regarded as affine while type-checking `swap`. Sometimes, however, such a conservative treatment is unsatisfactory. In order to address this problem, we introduce *mode constraints*. A mode constraint takes the form "`mode t`", where `t` is a type and `mode` is one of `duplicable`, `exclusive`, and `affine`. When quantifying over a type variable, one can specify a mode constraint. For instance, the function that takes an argument `x` and returns the tuple "`(x, x)`" has type "`[a] duplicable a => a -> (a, a)`". In the absence of this constraint, this function would be ill-typed, because it receives one copy of the permission "`x @ a`" and returns two copies of it.

3.5 Example: a signature for bags

In order to illustrate the syntax and the expressive power of our function types, we present the signature (that is, the interface) of a module that implements a bag. In short, a bag is a mutable container, which supports only two operations: inserting a new element and retrieving an arbitrary element. The signature appears in Figure 2. An implementation of this signature is presented later on (§5).

The first line declares the existence of an abstract type `bag`. The kind "`TYPE -> TYPE`" means that the type of a bag is parameterized with the type of its elements: if the elements have type `a`, then the bag has type "`bag a`".

The second line is a mode declaration. It indicates that, for every type `a`, the type "`bag a`" is exclusive. This implies, in particular, that "`bag a`" is not duplicable: "every bag has a unique owner".

The last three lines publish the names and types of the functions that allow creating and working with bags. All three are polymorphic in the element type.

The function `create` takes no argument and returns a new bag whose element type is chosen by the caller. Thus, if the user writes "`let x : bag a = create() in ...`", then, after the call, the permission "`x @ bag a`" appears. In short, one can say that "`create` returns a new bag, owned by the caller".

The function `insert` takes two arguments, namely an element `x` and a bag `b`, and returns nothing. The idea is that `x` is added to the bag `b`, which is modified in place. The function requires two permissions, namely "`x @ a`" and "`b @ bag a`". The former is consumed, as indicated by the `consumes` keyword, while the latter is returned to the caller. Thus, the ownership of `x` is transferred from the caller to the bag: "the bag owns its elements". The ownership of the bag is retained by the caller, but is nevertheless required in order for a call to `insert` to be permitted: "only the owner of the bag can insert (and retrieve) elements". In a shared-memory concurrent setting (§7.2), this guarantees that two threads cannot simultaneously attempt to access a bag.

The function `retrieve` takes one argument, namely a bag `b`, and produces one result, say `o`. Just like `insert`, it requires the permission "`b @ bag a`", and returns it. It also produces the permission "`o @ option a`". (The definition of the `option` type, which is omitted, is the same as in ML.) A `match` construct can then be used to determine whether `o` carries the tag `None` or `Some`. More specifically, if `o` matches the pattern `None`, then one finds out that the bag is empty. If `o` matches the pattern "`Some { value = x }`", then the type-checker replaces the permission "`o @ option a`" with "`o @ Some { value = x } * x @ a`". (This permission refinement step is explained in greater depth in §4.1.2.) Thus, we obtain a permission "`x @ a`" for the element `x` that was retrieved: "the ownership of the retrieved element is transferred from the bag to the caller".

In short, although the signature of Figure 2 is almost as concise as its ML or Java analogue, it effectively contains detailed information about the ownership of the bag and of its elements.

We do not yet show how the bag signature can be implemented; we do so later on (§5). However, let us point out right away that many implementations of this signature will pose a type-checking challenge. Here is why. Because a successful call to `retrieve` yields a permission for the retrieved element, and because this permission could be exclusive, the system guarantees that *a sequence of successful* `retrieve` *operations produces a sequence of pairwise distinct elements.* How can the system enforce such a nontrivial property?

For a simple bag implementation, say a mutable reference to a list of elements, the permission discipline is able to ensure that there is no duplication of permissions and therefore that the retrieved element is no longer in the list.

However, for an implementation that involves more complex patterns of sharing, say a doubly-linked list, it is quite non-obvious that this property holds. In fact, we believe that, in order to establish this property, either the programmer must provide a mathematical argument (including a definition of what it means for a set of memory blocks to form a "doubly-linked list", and a proof that the code respects this invariant), or one must rely on a dynamic check that somehow enforces this property.

Because we do not wish type-checking to require proof, we choose the latter path. Adoption and abandon, which we introduce further on (§5), involve dynamic book-keeping and dynamic checks. This approach simplifies the static discipline while preserving its expressive power.

```
data list a =
  | Nil
  | Cons { head: a; tail: list a }
```

Figure 3. The definition of immutable lists

```
val length: [a] list a -> int
let rec length [a] xs =
  match xs with
  | Nil ->
      0
  | Cons { tail = tail }
      1 + length tail
  end
```

Figure 4. Measuring the length of a list

4. Lists

In order to illustrate how algebraic data types, function types, and permissions work together, we present a number of operations on lists. We study "immutable lists" (that is, lists whose spine is immutable; §4.1) as well as "mutable lists" (lists whose spine is mutable; §4.2). We also illustrate how a mutable list cell can be (permanently) turned into an immutable one and why one might wish to do this (§4.3).

4.1 Immutable lists

The algebraic data type of immutable lists is defined in the same manner as in ML or Haskell (Figure 3).

Because this definition does not contain the keyword `exclusive`, `list` objects are immutable. This means that the `head` and `tail` fields can never be written, and that the tag of a `list` object can never be modified either.

4.1.1 Reasoning about modes

It is important to understand that the `list` objects represent only the *spine* of the list. The *elements* of the list, which have type `a`, can be immutable or mutable. Technically, the type variable `a` can be instantiated with an arbitrary type, which could be duplicable, exclusive, or affine.

Depending on the nature of `a`, the type "`list a`" has subtly different meaning and properties.

When `a` is duplicable, the permission "`xs @ list a`" does *not* imply that the elements of the list `xs` are pairwise distinct. Indeed, since "`x @ a`" is a duplicable permission, two distinct list cells can contain the same element `x`. Furthermore, when `a` is duplicable, "`list a`" is duplicable as well. Thus, the conjunction "`xs @ list a * ys @ list a`" does *not* imply that `xs` and `ys` are distinct addresses. In other words, immutable lists can be shared.

When `a` is exclusive, the permission "`xs @ list a`" *does* imply that the elements of the list `xs` are pairwise distinct. Indeed, the definition of the algebraic data type `list` states that every cell contains a permission "`x @ a`" for the element `x` that is contained in the `head` field. We have seen earlier that, when `a` is exclusive, the conjunction "`x @ a * y @ a`" implies that `x` and `y` are distinct. Thus, a list of exclusive elements is a list of *pairwise distinct* elements.

When `a` is exclusive, what mode is "`list a`"? It can't be duplicable: because duplicating a permission for a list entails duplicating the permissions for its elements, that would be unsound. It can't be exclusive either. Indeed, if it were, then the conjunction "`xs @ list a * ys @ list a`" would imply that `xs` and `ys` are distinct values. This is not the case: `xs` and `ys` can be both `Nil`. In conclusion, since "`list a`" is neither duplicable nor exclusive, it must be affine.

In summary, the definition of the algebraic data type `list` implies the following fact, which the system infers automatically: the type "`list a`" is duplicable if `a` is duplicable and is otherwise affine.

4.1.2 List length

Our first example of an operation on lists is the `length` function (Figure 4), which measures the length of a list. This example is interesting on at least two counts. First, we take this opportunity to explain how the `match` construct performs *permission refinement*, that is, how it replaces an imprecise permission with a more precise one. Second, we emphasize the fact that the `length` function is implicitly mode-polymorphic, that is, it works equally well with lists of duplicable elements, lists of exclusive elements, and lists of affine elements.

The type of the `length` function, "`[a] list a -> int`", indicates that this function is polymorphic with respect to the element type `a`. It also indicates that `length` requires a permission "`xs @ list a`" for its argument `xs` and returns this permission together with an integer result. The fact that the permission for the argument is *not* consumed is indicated by the *absence* of the `consumes` keyword.

The code is standard. The manner in which it is type-checked, however, is new. Let us explain which permissions are available at each point in the code.

Immediately before the `match` construct, the permission "`xs @ list a`" is available, because `length` receives it as an argument. This permission allows matching `xs` against `Nil` and `Cons` patterns. The `match` construct examines the value `xs` (and possibly dereferences it) so as to determine which of the two patterns it matches. This construct yields new information about `xs`. This is reflected, at type-checking time, by *refining* the permission "`xs @ list a`". Within each branch, this permission is replaced with a more precise one, as follows.

In the `Nil` branch, it is replaced with the structural permission "`xs @ Nil {}`". This permission reflects the fact that the tag carried by `xs` is now known to be `Nil`. If desired, this permission can be converted back to "`xs @ list a`", which is less precise. This conversion is implicitly performed at the end of the `Nil` branch, where the type-checker must verify that the function returns not only an integer result but also the permission “xs @ list a”.

In the Cons branch, the permission “xs @ list a” is replaced with the following conjunction:

```
xs @ Cons { head: a; tail = tail } *
tail @ list a
```

The first conjunct reflects the fact that xs is now known to be the address of a Cons cell, whose tail field contains the value tail. The second conjunct reflects the fact that the value tail is a list.

The permission “tail @ list a” is exactly what we need in order to justify the recursive call “length tail”. The call requires this permission and returns it, so that, after the call, this permission is still available. The permission “xs @ Cons { ... }” is not needed by the recursive call, so it just “sits there” while the call is in progress, and remains available after the call. In the terminology of separation logic [22], it is “framed out” during the call. Thus, after the call, the above conjunction of permissions is still available.

At the end of the Cons branch, the type-checker must verify that the permission “xs @ list a” is returned. Again, this works because “xs @ Cons { ... } * tail @ list a” can be folded back to “xs @ list a”, which is less precise.

This concludes our explanation of why length is well-typed. Although the code appears analogous to ML or Haskell code, the manner in which one reasons about it is quite different.

It is worth emphasizing that no assumption is made about the type parameter a. In other words, a is assumed to be affine, because “affine” is the top element of the mode hierarchy (Figure 1). As a result, “list a” is considered affine as well. This implies that a permission for an element, or for a list of elements, cannot be duplicated. Fortunately, no duplication of permissions is required in the above analysis of length.

The length function can be applied to lists whose elements have a duplicable type, such as “int”, “point”, or “list point”. It can also be applied to lists whose elements have an exclusive type, such as “xpoint”, or an affine type, such as “list xpoint”. Even though, technically, length is just polymorphic in the type variable a, one can informally say that length is polymorphic both in the type and in the mode of the list elements. We believe that this approach is original and is more lightweight than some of the prior approaches. Tov and Pucella [27], for instance, view our “modes” as kinds. This leads them to introducing kind polymorphism (in addition to type polymorphism) as well as a notion of dependent kinds.

As a stylistic note, the pattern “Cons { tail = tail }”, which binds the tail variable to the contents of the tail field, could be abbreviated to “Cons { tail }”. This is known as a “pun”. It would also be possible to use the simpler pattern “Cons {}”, or just “Cons”, which does not bind the variable tail. Instead of “length tail”, one would then write “length xs.tail”. The presence of the structural permission “xs @ Cons { head: a; tail = tail }” guarantees that the field access expression xs.tail is valid and allows the system to deduce that the type of this expression is the singleton type =tail (see §2.6).

```
val concat:
  [a] (consumes list a,
       consumes list a) -> list a
let rec concat [a] (xs, ys) =
  match xs with
  | Nil ->
      ys
  | Cons ->
      Cons {
        head = xs.head;
        tail = concat (xs.tail, ys)
      }
  end
```

Figure 5. Immutable list concatenation

4.1.3 List concatenation

Our second example is a binary operation, namely the list concatenation function, concat (Figure 5). As usual, this function maps two lists to a list, and is polymorphic in the element type a.

What is more interesting is that the permissions for the two arguments are consumed. We will explain why these consumes keywords are required and discuss the consequences.

The code is standard: it follows and copies the spine of the first list. Let us discuss how it is type-checked.

The permissions “xs @ list a” and “ys @ list a” are available upon entry. Just as in the length function (§4.1.2), the permission “xs @ list a” justifies the match construct, and is refined by this construct.

In the Nil branch, the result is ys. Because the function concat promises to produce a permission for its result at type “list a”, the type-checker must verify that the permission “ys @ list a” is available. This is indeed the case.

At this point, we see why a consumes keyword for the second argument is necessary. Indeed, if concat promised to also return a permission for its second argument, then, at this particular point in the code, we would need *two* copies of the permission “ys @ list a”. Because this permission is affine, it cannot be duplicated. So, in the absence of a consumes keyword for the second argument, a type error would be reported at the end of the Nil branch.

In the Cons branch, the permission “xs @ list a” is refined into “xs @ Cons { head: a; tail: list a }”. Furthermore, the type-checker is free to introduce names that stand for the values stored in the head and tail fields. That is, the type-checker automatically expands this permission into the conjunction:

```
xs @ Cons { head = head; tail = tail } *
head @ a *
tail @ list a
```

where `head` and `tail` are fresh auxiliary variables. This mechanism was illustrated earlier in the case of pairs (§2.5).

The first conjunct above implies that the field access expression `xs.tail` is valid and has type `=tail`, which means that its result must be the value `tail`. This allows the type system to recognize that the permissions required by the recursive call "`concat (xs.tail, ys)`" are "`tail @ list a`" and "`ys @ list a`". According to the explicitly-declared type of the function `concat`, these permissions are consumed by the call, and the new permission "`zs @ list a`" is produced, where the auxiliary variable `zs` is automatically introduced by the type-checker to stand for the result of the call. Thus, after the call, the current permission is:

```
xs @ Cons { head = head; tail = tail } *
head @ a *
zs @ list a
```

The field access expression `xs.head` has type `=head`, which means that its result must be the value `head`. The result of the call "`concat (...)`" has received the name `zs`. Thus, the permission produced by the expression "`Cons { ... }`", which allocates a new list cell, is:

```
c @ Cons { head = head; tail = zs }
```

where the auxiliary variable `c` is again introduced by the type-checker to stand for the result of this expression. This permission, along with "`head @ a`" and "`zs @ list a`", is implicitly folded back to the (less precise) permission:

```
c @ Cons { head: a; tail: list a }
```

which in turn is transformed into the (still less precise):

```
c @ list a
```

Thus, the type-checker is able to verify that `concat` returns the permission "`c @ list a`" for its result, which is `c`. This confirms that the code satisfies the declared signature.

In the `Cons` branch, the permission that grants read access to the cell `xs`, namely "`xs @ Cons { head = head; tail = tail }`", remains available until the end, but is unused, so it is silently discarded. We now see why a `consumes` keyword for the first argument is necessary. In the absence of this keyword, we would need to reconstruct the permission "`xs @ list a`". This means that we would need the permissions "`head @ a`" and "`tail @ list a`". However, these permissions have been consumed. The permission for `tail` was consumed by the recursive call, and the permission for `head` was consumed by the construction of "`c @ list a`". If we could copy these permissions prior to consuming them, this would not be a problem; however, they are not duplicable. Thus, in the absence of a `consumes` keyword for the first argument, a type error would be reported at the end of the `Cons` branch.

The function `concat` can also be assigned the type:

```
[a] duplicable a =>
  (list a, list a) -> list a
```

Here, the two `consumes` keywords have been removed, and the assumption that `a` is duplicable has been added, so that the code remains well-typed. This type is in fact subsumed by the type of Figure 5. Indeed, when `a` is duplicable, the permissions "`xs @ list a`" and "`ys @ list a`" can be copied at the call site, prior to invoking `concat`. This means that they are still available after the invocation, even though `concat` consumes one copy of them.

4.1.4 List map

All of the standard higher-order operations on lists can also be defined. Their types can be more complex than in ML or Haskell, both because *Mezzo* is more explicit about side effects and because it is more expressive (it allows strong updates). Let us consider the `map` function as an example. This function accepts two arguments, namely a function `f` and a list `xs`, and produces a new list, whose elements are the images of the elements of `xs` through `f`. It can of course be assigned its usual polymorphic type:

```
val map:
  [a, b]
    (f: (x: a) -> b,
     xs: list a)
  -> list b
```

For clarity, we have named the arguments of `map` and `f`, even though these names could be omitted, since they are never referred to. This type requires `f` to preserve its argument, and guarantees that `map` also preserves its argument `xs`. However, one might wish to apply `map` to a function `f` that destroys its argument. Then, `map` should be assigned the following type:

```
val map:
  [a, b]
    (f: (consumes x: a) -> b,
     consumes xs: list a)
  -> list b
```

These two candidate types for `map` are incomparable: each of them allows usage scenarios that the other forbids. Thus, it might seem that we are in trouble and that we need to define two versions of `map`. Fortunately, these types are subsumed by a more general one. In the general case, we must allow `f` to change the type of its argument from `a1` to `a2`, while producing a result of type `b`. Then, `map` changes the type of its argument from "`list a1`" to "`list a2`", while producing a result of type "`list b`". This is expressed as follows:

```
val map:
  [a1, a2, b]
    (f: (consumes x : a1) ->
        (b      | x @ a2),
      consumes xs : list a1)
  -> (list b | xs @ list a2)
```

If one takes both a1 and a2 to be a, one recovers the first candidate type above as a special case. If one takes a1 to be a and a2 to be unknown, one recovers the second candidate type. Thus, this new candidate type is indeed stronger than the previous two.

Are we done? Not quite. The above type is still not as permissive as one might wish. Indeed, it does not allow the function f to have a side effect on a data structure other than its argument x.

For instance, one might wish to supply a function f that increments a mutable counter c whenever it is invoked. In ML, this is possible: the closure of f captures the address of c. Here, the situation is slightly more complex. A closure can capture a value, just as in ML. However, a value is not good enough: a permission for this value is also required. Because our function types are duplicable, we cannot allow a closure to capture a non-duplicable permission. Thus, f cannot capture a permission to read and update the counter c.

(Allowing a closure to capture an affine permission would force the closure to be itself affine, which means that one would be allowed to invoke it at most once. Thus, we would have to introduce a distinction between ordinary and affine functions, as in Alms [27]. Because we prefer to avoid such a distinction, we disallow this scenario.)

This means that the permission to access c must be repeatedly passed to f and returned by f. This implies that this permission must be initially passed to map and eventually returned by map. Naturally, as the author of map, we do not know about the counter c, so we must use *permission polymorphism*. For an arbitrary permission p, if the function f requires p and returns it, then map itself requires p and returns it. We say that if f has effect p, then map also has effect p.

This is formally expressed as follows. For simplicity, we revert to the first of the three candidate types presented above, and we extend it with an effect over a permission parameter p. This yields the following type:

```
val map:
  [a, b] [p :: PERM]
    (f: (x: a | p) -> b,
     xs: list a | p)
  -> list b
```

The notation "p :: PERM" indicates that p is a permission variable, as opposed to a type variable. Since the occurrences of p are *not* preceded by the consumes keyword, the permission p is required and returned by f, and is also required and returned by map.

If one takes p to be the permission empty, one recovers the first candidate type above. Thus, this new type is indeed stronger than the previous version.

```
exclusive data xlist a =
  | XNil
  | XCons { head: a; tail: xlist a }
```

Figure 6. The definition of mutable lists

If one takes p to be the permission "c @ ref int", where c is the name of a counter, one finds that map can be applied to a function that increments c as a side effect.

We leave as an exercise for the reader to combine the two ideas discussed above, that is, to write down a type for map that allows f to perform a strong update on its argument and to have a side effect on some other piece of state.

This discussion hopefully sheds light on some of the strengths and weaknesses of the system. On the one hand, it is quite powerful and flexible. On the other hand, there is some danger that it is considered verbose and confusing. Experience will tell.

4.2 Mutable lists

Mutable lists are identical in structure to immutable lists. The only difference in their definition is the presence of the exclusive keyword (Figure 6). The xlist objects, which represent the spine of the list, are mutable. The elements of the list, which have type a, can be immutable or mutable.

4.2.1 Reasoning about modes

The permission "xs @ xlist a" is exclusive, regardless of the nature of the parameter a. Similarly, "xs @ XNil" and "xs @ XCons { ... }" are always exclusive. Thus, the conjunction "xs @ xlist a * ys @ xlist a" implies that xs and ys are disjoint lists: they cannot share a sub-list.

4.2.2 Mutable list copy

The system rules out any programming error that would cause an exclusive list to become shared. For instance, the identity function (which immediately returns its argument, without copying it) has type:

```
val id: [a]
  (consumes xlist a) -> xlist a
```

If the consumes keyword is omitted by mistake, a type error is reported. This artificial example illustrates the fact that forgetting to copy, an all-too-common error in traditional imperative programming languages, is detected.

When one must copy a list, one must decide whether a shallow copy or a deep copy is required. A shallow copy copies the spine, but not the elements, which become shared. A deep copy copies both the spine and the elements. Again, the type system helps us distinguish between these variants: the shallow copy function, copy, requires the elements to be duplicable, whereas the deep copy function, map, imposes no such requirement.

```
val copy: [a] duplicable a =>
val concat: [a]
  (consumes xlist a,
   consumes xlist a) -> xlist a
let concat [a] (xs, ys) =
  match xs with
  | XNil ->
      ys
  | XCons ->
      xs.tail <- concat (xs.tail, ys);
      xs
  end
```

Figure 7. Mutable list concatenation (naïve version)

```
val concat1: [a]
  (xs: XCons { head: a; tail: xlist a },
   consumes ys: xlist a) -> ()
let rec concat1 (xs, ys) =
  match xs.tail with
  | XNil ->
      xs.tail <- ys
  | XCons ->
      concat1 (xs.tail, ys)
  end

val concat: [a]
  (consumes xlist a,
   consumes xlist a) -> xlist a
let concat (xs, ys) =
  match xs with
  | XNil ->
      ys
  | XCons ->
      concat1 (xs, ys);
      xs
  end
```

Figure 8. Mutable list concatenation (tail-recursive version)

```
    xlist a -> xlist a

  val map: [a, b]
    (a -> b, xlist a) -> xlist b
```

4.2.3 Mutable list concatenation

Because exclusive lists are never shared, they can be concatenated in place, that is, melded. A simple recursive function that performs this task is presented in Figure 7. Checking that this code is well-typed is left as an exercise for the reader. The mechanisms involved are exactly the same as in the case of immutable lists (§4.1.2, §4.1.3).

Naturally, the code in Figure 7 is naïve. It performs a linear number of memory writes, whereas one should suffice, and (because it is not tail-recursive) it uses linear space, whereas constant space should suffice.

We can do better. The idea is to define a version of `concat` that is specialized for the case where its argument carries the tag `XCons`. In this new function, `concat1`, it is easy to identify and eliminate the redundant write, and as a result, the function becomes tail-recursive. Then, the main function, `concat`, can be defined in terms of `concat1`.

The code appears in Figure 8. In the type of `concat1`, we use a structural permission to require that `xs` be non-empty. This permission justifies the read and write accesses to `xs.tail`. The absence of a `consumes` keyword means that this permission is returned by `concat1`. That is, after the call, `xs` is still a valid, non-empty list. The permission for `ys`, on the other hand, is consumed.

Whether concatenation should be implemented as a tail-recursive function, as in Figure 8, or as a `while` loop, as traditionally done by C programmers, may at first appear to be purely a matter of taste. Indeed, the two implementations should in principle have identical efficiency, and, when comparing untyped source code, they have roughly the same length. (The recursive version is at a slight disadvantage.) When it comes to reasoning about memory safety, however, we believe that our tail-recursive version has a definite edge over the iterative version. In Figure 8, the “loop invariant” is just the type of `concat1`, which is pretty natural. In contrast, Berdine *et al.*’s iterative version [2], which is proved correct in separation logic, requires a more complex loop invariant, which involves two “list segments”, as well as an inductive proof that the concatenation of two list segments is a list segment.

How are we able to get away without list segments and without inductive reasoning? The trick is that, even though `concat1` is tail-recursive, which means that no code is executed after the call by `concat1` to itself, a *reasoning step* still takes place after the call. Immediately before the recursive call to `concat1`, the current permissions are:

```
  xs @ XCons { head: a; tail = xt } *
  xt @ XCons { head: a; tail: xlist a } *
  ys @ xlist a
```

where the auxiliary variable `xt` stands for `xs.tail`. The call “`concat1 (xs.tail, ys)`” requires only the last two of the above permissions. The first one is “framed out”, which means that it is implicitly preserved by the call. The call returns the permission for `xt` and consumes the permission for `ys`. In summary, after the call, the permissions for `xs` and `xt` remain available. The type-checker verifies that these permissions can be folded back to:

```
  xs @ XCons { head: a; tail: xlist a }
```

which means that the original permission for `xs` is returned as promised. The framing out of a permission during the recursive call, as well as the folding step that takes place after the call, are the key technical mechanisms that allow us to avoid the need for list segments and inductive reasoning.

```
val concat1: [a]
  (consumes dst: XCons{head: a; tail: ()},
   consumes xs: list a,
   consumes ys: list a) ->
  (| dst @ list a)
let rec concat1 (dst, xs, ys) =
  match xs with
  | Nil ->
      dst.tail <- ys;
      dst <- Cons
  | Cons { head; tail } ->
      let dst' = XCons {
        head = head; tail = ()
      } in
      dst.tail <- dst';
      concat1 (dst', tail, ys);
      dst <- Cons
  end

val concat: [a]
  (consumes xs: list a,
   consumes ys: list a) -> list a
let concat (xs, ys) =
  match xs with
  | Nil ->
      ys
  | Cons { head; tail } ->
      let dst = XCons {
        head = head; tail = ()
      } in
      concat1 (dst, tail, ys);
      dst
  end
```

Figure 9. Tail-recursive concatenation of immutable lists

In short, one might say that our code is tail-recursive, but the manner in which the type-checker reasons about it is not.

Walker and Morrisett [30] also offer a tail-recursive version of mutable list concatenation. Their code is formulated in a low-level typed intermediate language, as opposed to a surface language. The manner in which they are able to avoid reasoning about list segments is analogous to ours. There, because the code is formulated in continuation-passing style, the reasoning step that takes place "after the recursive call" amounts to composing the current continuation with a coercion. Maeda *et al.* [18] study a slightly different approach, also in the setting of a typed intermediate language, where separating implication offers a generic way of defining list segments.

Our approach can be adapted to an iterative setting by adopting a new proof rule for `while` loops. This is noted independently by Charguéraud [9, §3.3.2] and by Tuerk [28].

4.3 Tail-recursive concatenation of immutable lists

Whereas our implementation of mutable list concatenation (§4.2.3) uses tail recursion and has constant space overhead, our implementation of immutable list concatenation (§4.1.3) uses general recursion and (as a result) has linear space overhead.

Must concatenating immutable lists be more expensive than concatenating mutable lists? No. In an untyped setting, it is not difficult to write a tail-recursive function that constructs the concatenation of two immutable lists. The challenge lies in convincing a type-checker that this code makes sense. There are two (related) reasons why this is difficult. One reason is that the code allocates a fresh list cell and initializes its `head` field, but does not immediately initialize its `tail` field. Instead, it makes a recursive call and delegates the task of initializing the `tail` field to the callee. Thus, the programming language must somehow allow the delayed initialization of an immutable data structure. The other reason is that, while concatenation is in progress, the partly constructed data structure is not (yet) a list: it is a list segment. Thus, it seems that the programming language must allow defining and reasoning about list segments.

We solve the first difficulty via strong updates, that is, instructions that update an exclusive permission. Writing a value `xs` into the `tail` field of a mutable list cell is a strong update: this instruction changes the type of the `tail` field to the singleton type `=xs`. Turning a mutable list cell `dst` into an immutable one is also a strong update: the permission "`dst @ XCons { ... }`" is changed into "`dst @ Cons { ... }`". This change is performed via a tag update instruction (§2.7), written "`dst <- Cons`". In this particular case, because `XCons` and `Cons` are ultimately represented by the same integer tag, this instruction does nothing at runtime.

We solve the second difficulty in the same manner as in the case of mutable list concatenation (§4.2.3). The code is tail-recursive, but the manner in which the type-checker reasons about it is not.

The tail-recursive version of immutable list concatenation is presented in Figure 9. The auxiliary function `concat1` is written in destination-passing style. It accepts a partially-initialized, mutable list cell `dst`, together with two immutable lists `xs` and `ys`. Its task is to construct the concatenation of `xs` and `ys` and to write the address of this list into `dst.tail`, thus transforming `dst` itself into an immutable list. It returns no result. The type of `concat1` expresses the key idea that, after the call, `dst` is a well-formed immutable list. Indeed, this type states that the original permissions for `dst`, `xs` and `ys` are consumed and that the permission "`dst @ list a`" is returned instead.

The function `concat1` is tail-recursive because the tag update instruction "`dst <- Cons`" does nothing at runtime. This instruction serves as a type annotation: it indicates that, at this point, the cell `dst` becomes immutable.

Why is `concat1` well-typed? We let the reader check that, after the allocation of the new cell `dst'`, the current permission can be written under the following form:

```
dst @ XCons { head: a; tail: () } *
tail @ list a *
ys @ list a *
dst' @ XCons { head: a; tail: () }
```

That is, at this point, we have two partially-initialized mutable list cells and two immutable lists at hand. The field update instruction "`dst.tail <- dst'`" changes the permission that describes the cell `dst`, so that, after this instruction, the current permission is:

```
dst @ XCons { head: a; tail = dst' } *
tail @ list a *
ys @ list a *
dst' @ XCons { head: a; tail: () }
```

Next comes the recursive call to `concat1`. The first permission above is "framed out" during the call. The last three are consumed by the call, which produces "`dst' @ list a`". Thus, after the call, the current permission is:

```
dst @ XCons { head: a; tail = dst' } *
dst' @ list a
```

This can be folded back to:

```
dst @ XCons { head: a; tail: list a }
```

The tag update instruction "`dst <- Cons`" changes this to:

```
dst @ Cons { head: a; tail: list a }
```

which can be folded back to

```
dst @ list a
```

Thus, we are able to verify that the code meets its promise.

This approach, which involves delayed initialization and the transformation of a mutable memory block into an immutable one, can be used to implement other operations on immutable lists, such as `map`, in a tail-recursive manner.

Minamide [19] approaches this problem by introducing "data structures with a hole" as a primitive notion. These data structures are equipped with primitive operations for application and composition and with an affine type discipline. Our approach is less expressive, but more elementary: we use ordinary reads and writes, together with an instruction that makes an object immutable.

5. Adoption and abandon

The algebraic data types that we have presented so far are limited in their expressive power. The heap structures that they can describe are either immutable or tree-shaped. In other words, they cannot describe mutable data structures that involve sharing. This stems from the fact that permissions for mutable objects are (by design) exclusive.

Thus, we must extend the system with a way of describing and manipulating mutable data structures with sharing.

5.1 How to organize mutable objects in groups?

One way to approach the problem is to consider how one might implement a FIFO queue that satisfies the bag signature (Figure 2). Let us choose a simple data structure, namely a mutable singly-linked list. One inserts elements at the tail and extracts elements at the head. There is a root object `b`, which the client considers as "the bag". This object contains pointers to the head and tail of the list, so as to allow constant-time insertion and extraction.

The last cell in the list is shared. It is accessible via two distinct paths. This prevents us from declaring that "each cell owns its successor", that is, each cell contains an exclusive permission for its successor. Indeed, if we declared such a thing, then we would be unable to associate a non-trivial permission with the `tail` pointer, which points directly from the bag to the last cell. In short, because the heap is not tree-structured, the ownership hierarchy cannot coincide with the structure of the heap.

In order to overcome this problem, a natural idea is to consider that the list cells are collectively owned by the bag. This allows the ownership diagram to remain a hierarchy. To do so, instead of keeping track of one exclusive permission per cell, we keep track of one exclusive permission for the *group* formed by all of the list cells. In fact, we go one step further and fuse this permission with the permission that governs the bag `b`.

This idea is not new: groups of objects, or "regions", have received sustained interest in the literature [11, 12, 15], including in our own previous work [10]. The idea that there can be a single permission for a group of objects is present in these papers.

In this light, the adoption and abandon operations which we are about to introduce can be presented as the operations by which an object joins and leaves a group.

We require both adoptees and adopters to be exclusive. The exclusive permission that controls the adopter also serves as a permission for the entire group.

Adoption is the operation by which an object `x` joins a group. Adoption requires and consumes a permission for `x`: the ownership of `x` is transferred to the group.

As long as the object `x` remains a member of the group, there is no way to read or write this object, because there is no longer a permission for it. Indeed, a permission for a group does not directly give access to the group members. The only way of recovering read and write access to `x` is to make it leave the group.

Abandon is the operation by which an object `x` leaves a group. It produces a permission for `x`: the group relinquishes the ownership of `x`.

Abandon must be carefully controlled. If an object `x` could be abandoned twice by a group, one would gain two permissions for `x`. This would be unsound. In the presence of aliasing, ensuring that this situation does not arise is potentially difficult. When a group successively abandons two objects `x` and `y`, one must somehow ensure that `x` and `y` are distinct objects.

Similarly, one must ensure that, at any single time, every object is a member of at most one group. If an object could be at once a member of two distinct groups, then (by abandon) one would again be able to obtain two permissions for this object, which would be unsound.

5.2 What permission describes a member of a group?

Although we have announced that we intend to gather the cells that form the FIFO queue into a group, we have not yet explained how to solve the original problem, which is the following. There exist two pointers to the last cell, which is a mutable object. Each of these pointers must be accompanied with a permission. What permission could this be?

- It cannot be the trivial permission `unknown`, because we would then be unable to dereference these pointers.
- It cannot be an exclusive permission, because we would be unable to exhibit two copies of such a permission.
- It could be a duplicable permission that grants non-exclusive read and write access to a mutable object. Bierhoff and Aldrich's "`share`" permissions [3] are of this kind. However, non-exclusive permissions are quite weak. They do not support strong updates, which means that they do not allow keeping precise track of must-alias information as updates are performed. Furthermore, a shared object cannot contain exclusive components, unless elaborate precautions are taken [3, §5.2]. Thus, it is unclear whether "`share`" permissions could help us implement bags whose elements can be exclusive and whose internal representation involves sharing.

We answer this question by introducing a new type, "`dynamic`". The permission "`x @ dynamic`" is duplicable. It does not grant read and write access to the object `x`. Instead, it represents a permission to perform a *dynamic group membership check*, which, if successful, yields an ordinary exclusive permission to read and write `x`. This is in fact the abandon operation.

This resolves the tension between the desire to work with exclusive permissions (which allow strong updates, and can be constructed by combining a number of simpler exclusive permissions) and the desire to use duplicable permissions (which allow uncontrolled sharing). We work alternatively with one and with the other, and offer operations for converting one to the other, and back. These operations are *adoption* and *abandon*.

5.3 Design decisions

In order to make the use of groups as simple and as flexible as possible, while ensuring type soundness, we make the following two design decisions:

1. An adopter serves as a proxy for the group of its adoptees.
2. We keep track *at runtime* of which object is a member of which group. We use this information to ensure *at runtime* that abandon is used in a safe way.

The first decision means that instead of considering that "the cells are part of some group g, and a permission for g is stored in the object `b`", the programmer declares directly that "the cells are adopted by the object `b`". Thus, `b` is not only the name of an object that exists at runtime in the heap, but also, in the eyes of the type-checker, the name of a group of objects, namely, the group of the objects that are presently adopted by `b`. We consider that a permission for `b` controls not only the object `b`, but also all of its adoptees.

This allows us to avoid explicitly introducing the notion of a "group" as a first-class concept: instead, we encourage the programmer to think in terms of "adoption". In the case of bags and (we believe) in many other situations, this allows the programmer to work with fewer names: there is no need to come up with a name for the group.

The second decision implies that we implicitly maintain a pointer from every adoptee to its adopter. More precisely, within every exclusive object, we maintain an "`adopter`" field, which contains either a pointer to the object's current adopter, if there is one, or `null`, otherwise. This information is updated whenever an object is adopted or abandoned. This makes it possible to dynamically test, for two objects `x` and `o`, whether `x` is currently adopted by `o`. We build this check into the dynamic semantics of abandon: an attempt to make `o` abandon `x` fails if `x` is not currently adopted by `o`. This ensures that abandon is safe.

In terms of space, the cost of this design decision is one extra field per exclusive object. It is possible to lessen this cost by letting the programmer declare that certain objects cannot be adopted and don't need this field. This is discussed later on (§5.7.2); until there, we assume that every exclusive object can be adopted.

Because abandon involves a dynamic check, it can cause the program to encounter a (fatal) failure at runtime. In principle, if the programmer knows what she is doing, no failure should ever occur. There is some danger, but such is the price to pay for a simpler static discipline. And, after all, the danger is effectively less than in ML or Java, where a programming error that creates an undesired alias goes completely undetected—until the program misbehaves in one way or another.

Adoption and abandon can be viewed as a dynamic discipline for ensuring that affine permissions are never duplicated. Tov and Pucella's stateful contracts [26] are another such discipline. Wolff *et al.* [31] also study dynamic permission checking. For the moment, their approach "mandates a fully-instrumented runtime semantics".

5.4 The details of adoption and abandon

Let us now explain in detail the dynamic semantics of adoption and abandon (what these operations do) as well as their static semantics (what the type-checker requires).

5.4.1 `adopter` fields

As explained earlier (§5.3), every exclusive object contains a hidden field, named "`adopter`", which contains either `null` or a pointer to another (exclusive) object. We maintain the following informal invariant:

1. If `y.adopter` is `null`, then the object `y` is not currently a member of any group. The object `y` is possibly currently tracked by the type system: there may currently exist an exclusive permission for this object.
2. If `y.adopter` is a non-null pointer to `x`, then the object `y` is currently a member of the group represented by `x`. The object `y` is definitely not tracked by the type system: there currently exists no exclusive permission for `y`.

5.4.2 The type `dynamic`

As also explained earlier (§5.2), we introduce a new type, named `dynamic`. Thus, "`y @ dynamic`" is a permission. This permission guarantees that the field `y.adopter` exists and grants the right to read it. Abandon exploits this permission.

if the field `y.adopter` exists now, then it exists forever. This means that it is sound for the permission "`y @ dynamic`" to be duplicable.

By convention, every exclusive object has an `adopter` field. Thus, whenever one holds a permission of the form "`y @ u`", where the type `u` is exclusive, it is sound for the permission "`y @ dynamic`" to spontaneously appear, in addition to "`y @ u`". The type-checker is aware of this rule and automatically applies it where needed.

The permission "`y @ dynamic`" does not allow reading or writing the normal (non-hidden) fields of the object `y`. In fact, it does not even tell how many normal fields there are. In short, it does not represent the ownership of `y`.

In the FIFO bag implementation, shown in Figure 10, the `head` and `tail` fields of a non-empty `bag` object, as well as the `next` field of every `cell` object, have type `dynamic` (lines 2 and 6). Thus, none of these "points-to" relationships implies an "owns" relationship. Owning a cell `c` does not imply that one owns the cell `c.next`; in fact, it does not even imply that the object at address `c.next` is a cell. Similarly, owning a bag `b` does not imply that one owns the objects `b.head` and `b.tail`, nor does it imply that these objects are distinct. The only fact that is known for sure is that it is permitted to dynamically test whether (say) `b.head.adopter` is `b`. Of course, we expect this test to always succeed, otherwise we must have made a mistake in the code.

```
1  mutable data cell a =
2    Cell { elem: a; next: dynamic }
3
4  mutable data bag a =
5       Empty { head, tail: () }
6  | NonEmpty { head, tail: dynamic }
7  adopts cell a
8
9  val create [a] () : bag a =
10   Empty { head = (); tail = () }
11
12 val insert [a] (consumes x: a, b: bag a) : () =
13   let c = Cell { elem = x; next = () } in
14   c.next <- c;
15   give c to b;
16   match b with
17   | Empty ->
18       tag of b <- NonEmpty;
19       b.head <- c;
20       b.tail <- c
21   | NonEmpty ->
22       take b.tail from b;
23       b.tail.next <- c;
24       give b.tail to b;
25       b.tail <- c
26   end
27
28 val retrieve [a] (b: bag a) : option a =
29   match b with
30   | Empty ->
31       None
32   | NonEmpty ->
33       take b.head from b;
34       let x = b.head.elem in
35       if b.head == b.tail then begin
36         tag of b <- Empty;
37         b.head <- ();
38         b.tail <- ()
39       end else begin
40         b.head <- b.head.next
41       end;
42       Some { value = x }
43   end
```

Figure 10. A FIFO implementation of bags

5.4.3 `adopts` declarations

Once an object `y` has been adopted by some group, there no longer exists an exclusive permission for this particular object. That is, there cannot exist a permission of the form "`y @ u`", where the type `u` is exclusive. There can be permissions of the form "`y @ dynamic`", which guarantee that `y` has an `adopter` field, but these permissions do not tell how many fields exist in the object at address `y` and what their type is. If at some point we decide to make the group abandon `y`, so as to recover an exclusive permission of the form "`y @ u`", how can we find out which type `u` correctly describes `y`?

Our answer is, this information must be associated with the group. That is, groups must be homogeneous: all members of a common group must have a common type `u`. We require the programmer to fix this type as part of the definition of the type of the adopter. More precisely, for `x` to be able to act as an adopter, we require a permission of the form "`x @ t`", where `t` is an exclusive algebraic data type, and we require that, as part of the definition of the data type `t`, the programmer declare "`adopts u`", where `u` must be an exclusive type.

This is illustrated in the FIFO bag implementation, where the definition of the algebraic data type "`bag a`" includes the declaration "`adopts cell a`" (Figure 10). Thus, if `b` has type "`bag a`", then the type-checker must ensure that every object that is adopted by `b` has type "`cell a`". In return, the type-checker can assume that every object that is abandoned by `b` has type "`cell a`".

In this mechanism, the address of the adopter effectively serves as a tag that contains type information. Indeed, by confirming that `y.adopter` is `b`, one gains the information that `y` is a cell.

5.4.4 Adoption

The syntax of adoption is "`give y to x`". When the type-checker encounters this instruction, it checks that, at the program point that precedes it, two permissions "`x @ t`" and "`y @ u`" are available, where the definition of the algebraic data type `t` contains the declaration "`adopts u`".

At runtime, the effect of this operation is to write the address `x` to the field `y.adopter`. The presence of an exclusive permission "`y @ u`" guarantees that this field exists and that the value of this field, prior to the adoption, is `null`.

At the program point that follows this instruction, the type-checker considers that the permission "`x @ t`" is still available, but the permission "`y @ u`" has been consumed. As explained before, this means that the ownership of the adoptee now resides with the group. That is, the permission "`x @ t`" represents not just the ownership of `x`, but also the ownership of its adoptees.

At the program point that follows the `give` instruction, the permission "`y @ dynamic`" is available. In fact, this permission was present already before the `give` instruction. This follows from the fact that "`y @ dynamic`" spontaneously appears out of "`y @ u`" (§5.4.2). While "`y @ u`" is consumed during adoption, "`y @ dynamic`" is not.

In the FIFO bag implementation (Figure 10), adoption is used at the beginning of `insert` (line 15), after a fresh cell `c` has been allocated and initialized. It allows us to make this new cell a member of the group, so as to maintain the (unstated) invariant that every cell that is reachable from `b` is a member of the group `b`.

When the type-checker encounters the instruction "`give c to b`", it first checks that we own `b`. This is the case: at this point, we have "`b @ bag a`". Then, the type-checker looks up the definition of the data type `bag`, and finds that `b` can adopt objects of type "`cell a`". The next step is to check that we have the permission "`c @ cell a`". At this point, we have:

1. "`c @ Cell { elem = x; next = c }`", as a result of the memory allocation expression and of the assignment that follows it;
2. "`x @ a`", by assumption;
3. "`c @ dynamic`", because, as soon as there is an exclusive permission for `c`, the permission "`c @ dynamic`" appears.

By combining these three permissions, the type-checker is indeed able to construct "`c @ cell a`". Thus, adoption is legal. After the `give` instruction, we lose "`c @ cell a`". We retain "`b @ bag a`" and "`c @ dynamic`". The presence of the latter permission, as well as the fact that this permission is duplicable, justify why, in the code that follows, it is permitted to write the value `c` into a subset of `b.head`, `b.tail`, and `tail.next` (lines 19, 20, 23, and 25).

5.4.5 Abandon

The syntax of abandon is "`take y from x`". When the type-checker encounters this instruction, it checks that, at the program point that precedes it, two permissions "`x @ t`" and "`y @ dynamic`" are available. It also checks that the definition of the algebraic data type `t` contains a declaration of the form "`adopts u`", for some type `u`.

At runtime, the effect of this operation is to first check that the field `y.adopter` contains the address `x`. (The permission "`y @ dynamic`" guarantees that this field exists.) If this is the case, the operation succeeds: the value `null` is written into `y.adopter`, so as to reflect the fact that `x` abandons `y`. Otherwise, the operation fails: the execution of the program is aborted.

At the program point that follows this instruction, the permission "`x @ t`" remains available. Furthermore, the permission "`y @ u`" appears. This is sound because `y.adopter` is now `null`. If the programmer (intentionally or mistakenly) attempts to have some group `x` twice abandon some object `y` (perhaps under two different names `y1` and `y2`, which at runtime happen to denote the same address), then the second abandon operation will fail at runtime.

In the FIFO bag implementation (Figure 10), abandon is used near the beginning of `retrieve`, at line 33. There, the first cell in the FIFO queue, whose address is found in `b.head`, must be abandoned by the group. It is absolutely necessary that this cell be abandoned: indeed, otherwise, we would be unable to recover a permission "`x @ a`" for the value `x` that is found in the `elem` field of this cell.

The abandon operation "`take b.head from b`" is allowed by the permissions "`b @ NonEmpty { ... }`" and "`head @ dynamic`". After the operation, these permissions remain available, and, in addition, the permission "`head @ cell a`" appears. This explains why, in the code that follows, it is permitted to read `head.next` and `head.elem` (lines 40 and 34) and why we recover the permission "`x @ a`", where `x` is the value found in the field `head.elem`.

Abandon and adoption are also used inside `insert`, at lines 22 and 24. There, the bag `b` is non-empty, and the cell `b.tail` must be updated in order to reflect the fact that it is no longer the last cell in the queue. However, we cannot just go ahead and access this cell, because the only permission that we have at this point about `tail` is "`tail @ dynamic`". The group must abandon this cell, yielding the permission "`tail @ cell a`". We can then update this cell. When we are done, we relinquish the permission "`tail @ cell a`" by letting the group adopt this cell again. This well-parenthesized use of `take` and `give` is related to Fähndrich and DeLine's "focus" [15] and to Sing#'s "expose" [14]. One could conceivably offer syntactic sugar for such a well-parenthesized use.

5.5 Summary

Let us briefly summarize what has been said so far about the FIFO bag implementation.

The main ideas that we wish to convey via this example are the following:

1. this data structure involves sharing (it is not a forest);
2. this data structure is mutable; furthermore, it implements a container whose elements can be mutable as well;
3. hence, this data structure must involve a group; inserting a new element must involve adoption, and retrieving an element must involve abandon.

We would like the programmer to reason in this manner, so as to recognize when and where adoption and abandon are needed.

5.6 On tag updates and state changes

Our implementation exploits the fact that it is permitted to mutate not just the fields, but also the tag of an exclusive object (§2.7). We declare that an object of type "`bag a`" carries either the tag `Empty`, in which case the `head` and `tail` fields have the unit type `()`, or the tag `NonEmpty`, in which case these fields have type `dynamic`. When the status of a bag `b` changes from empty to non-empty (lines 18–20) or vice-versa (lines 36–38), we reflect this change by updating the tag and the fields of `b`. At line 18, for instance, the permission that initially describes `b` is:

```
b @ Empty { head, tail : () }
```

After the first assignment, it is replaced with:

```
b @ NonEmpty { head, tail: () }
```

This structural permission may seem disturbing, because it cannot be folded back to "`b @ bag a`". This is not a problem: at this point, nothing requires us to produce the permission "`b @ bag a`". After the second assignment, the current permission is:

```
b @ NonEmpty { head = c; tail: () }
```

Finally, after the last assignment, the current permission is:

```
b @ NonEmpty { head = c; tail = c }
```

Because we also have "`c @ dynamic`", this permission can be folded back to "`b @ bag a`", so that, when `insert` completes, we are able to return "`b @ bag a`", as promised.

If we did not have the ability to mutate an object's tag, we could get away by using the `option` type. Instead of tagging bags, we would declare that the `head` and `tail` fields have type "`option dynamic`". This is how things are usually done in ML. This approach, however, incurs a space and time overhead, because objects of type `option dynamic` are heap-allocated.

Our algebraic data type definitions allow the types of the fields of an object to depend on this object's tag. Our tag update instructions allow an object's tag to change with time. Our permissions include not only nominal permissions, such as "`b @ bag a`", but also structural permissions, such as "`b @ Empty { head = c; tail = c }`", which offer precise information about an object's tag and fields. These ideas seem to be closely related to the treatment of *states* in Plaid [1]. The design of Plaid is more ambitious in that Plaid's states are organized in a subtyping and inheritance hierarchy, whereas we do not have any inheritance and our subtyping hierarchy is only one level deep: the nominal permission "`b @ bag a`" lies above the more precise structural permissions "`b @ Empty { ... }`" and "`b @ NonEmpty { ... }`".

5.7 Further refinements and extensions

Up to this point, we have tried to present groups, adoption and abandon in the simplest possible manner. We now discuss a few extensions and refinements of the basic proposal.

5.7.1 Testing membership in a group

Abandon involves a dynamic membership *check*. The instruction "`take y from x`" succeeds if `y` is currently a member of the group `x` and fails otherwise. This failure is considered fatal.

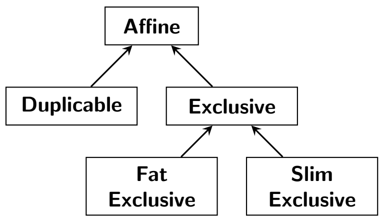


Figure 11. The full mode hierarchy

It seems natural to also offer a dynamic membership *test*, that is, an expression that tests whether `y` is currently a member of the group `x`, and produces a Boolean result.

The syntax of this expression is "`x owns y`". Just like abandon, this expression requires the permissions "`x @ t`" and "`y @ dynamic`", where `t` is an algebraic data type whose definition contains an `adopts` declaration. It preserves these permissions. Unlike abandon, this expression does not produce a new permission. It yields a Boolean result, which represents the outcome of the comparison "`y.adopter == x`".

We have observed earlier that, during abandon, the address of the adopter serves as a tag that allows recovering information about the adoptee. Introducing the construct "`x owns y`" allows the programmer to branch on this tag.

5.7.2 Finer-grained control of hidden fields

So far, we have assumed that every exclusive object can be adopted. This implies that every exclusive object must have an `adopter` field, which is costly.

There is another reason why this is undesirable. It interacts with another feature, namely the transformation of an exclusive object into an immutable one via a tag update instruction (§4.3). Type soundness dictates that, after the transformation, the `adopter` field must continue to exist, and must contain `null` forever. Furthermore, we would like this field to remain hidden after the transformation, because its existence is an implementation detail that we do not wish to expose. The combination of these constraints makes it difficult to compile field access in immutable objects.

In order to mitigate this issue, we propose a slightly more fine-grained approach. We distinguish two kinds of exclusive objects, namely *slim* and *fat* ones. A slim object has no hidden fields, so it cannot be adopted. A fat object has a hidden `adopter` field and can be adopted, The mode hierarchy is extended as shown in Figure 11.

When the programmer defines an algebraic data type, she chooses between immutable, fat exclusive, and slim exclusive. In the FIFO bag implementation of Figure 10, the type `cell` would have to be declared "`fat exclusive`", because cells must be adopted. The type parameter `a` remains unconstrained, because we do not directly adopt elements of type `a`, but instead adopt cells that serve as wrappers for elements. The type `bag` can be declared "`slim exclusive`". (Thus, a client who wishes to adopt a bag cannot do so directly, but can still wrap the bag and adopt the wrapper.)

Adoption and abandon are restricted to fat adoptees. That is, a declaration "`adopts u`" (§5.4.3) is allowed only if `u` is fat, and the permission "`y @ u`" gives rise to the permission "`y @ dynamic`" (§5.4.2) only if `u` is fat.

The tag update instruction (§2.7) is restricted to one of the following three kinds of transitions: slim to duplicable; slim to slim; and fat to fat.

5.7.3 Merging groups

Imagine that we wish to extend the bag interface and implementation (Figures 2 and 10) with an operation that adds the contents of one bag en masse to another bag:

```
val transfer:
  [a] (consumes bag a, bag a) -> ()
```

The idea is, the call "`transfer (b1, b2)`" inserts all of the elements of the bag `b1` into the bag `b2`. The bag `b1` is destroyed in the process: that is, the permission "`b1 @ bag a`" is consumed.

A priori, there is hope that this operation can be implemented in constant time. In our singly-linked list implementation, for instance, it seems that it suffices to meld the two lists in place. Indeed, in a traditional setting, this is true. Here, however, this is not quite sufficient: in addition, we must also arrange for every element of `b1` to be abandoned by `b1` and adopted by `b2`.

Using the instructions `give` and `take`, the only way of doing this is to iterate over every element of `b1`. However, this is unacceptable: it prevents us from achieving constant time complexity.

In order to address this issue, it seems desirable to offer a primitive operation for efficiently merging a group into another group. Let us now describe how this operation might be presented to the programmer and implemented.

The syntax is "`merge x1 into x2`". This operation requires two permissions "`x1 @ t1`" and "`x2 @ t2`", where `t1` and `t2` are algebraic data types whose definitions contain the declaration "`adopts u`", for a common type `u`. (The two groups must agree on the type of the adoptees). The effect of this operation is that every adoptee of `x1` becomes adopted by `x2` instead. The permission "`x1 @ t1`" is consumed: the group `x1` disappears. The permission "`x2 @ t2`" is preserved.

How do we implement this mechanism? We now need two basic operations over groups, instead of one. As before, in order to implement abandon, we must be able to find which group an object belongs in. In addition, we must now be able to merge one group into another. This is exactly the union-find problem, for which efficient solutions are known [25].

We add one new hidden field, named `link`, to every object that can serve as an adopter. We use this field to implement a union-find data structure over adopters. We maintain the

```
1  val transfer [a]
2    (consumes b1: bag a, b2: bag a) : () =
3    match b1 with
4    | Empty ->
5        ()
6    | NonEmpty ->
7        match b2 with
8        | Empty ->
9            b2 <- NonEmpty;
10           b2.head <- b1.head;
11           b2.tail <- b1.tail
12       | NonEmpty ->
13           take b2.tail from b2;
14           b2.tail.next <- b1.head;
15           give b2.tail to b2;
16           b2.tail <- b1.tail
17       end;
18       merge b1 into b2
19   end
```

Figure 12. An implementation of `transfer` for FIFO bags

invariant that, if there exists an exclusive permission for an adopter `x`, then `x.link` is `null`, that is, `x` is the root of a union-find tree. Such a permission for `x` can then effectively be regarded as a permission for every adopter in the tree whose root is `x`.

The instruction "`merge x1 into x2`" is implemented by one memory write: "`x1.link <- x2`". Because this operation requires a permission for `x1`, the previous value of `x1.link` must be `null`.

The implementation of adoption (§5.4.4) is unchanged. In the implementation of abandon (§5.4.5) and of the group membership test (§5.7.1), the test "`y.adopter == x`" is replaced with "`find(y.adopter) == x`", where the auxiliary function `find` follows the `link` pointers up to the root of the adopter tree and performs path compression on the fly. Because these operations require a permission for `x`, the field `x.link` must be `null`, hence there is no need to call `find(x)`.

For simplicity, we adopt the convention that "fat exclusive" objects (§5.7.2) have both an `adopter` field and a `link` field, while "slim exclusive" and immutable objects have neither. Thus, only a fat exclusive object can be an adopter or an adoptee, and it can be both at once.

In summary, by using a simple union-find data structure, we are able to implement merging in constant time, while slightly degrading the performance of abandon and of the group membership test, whose amortized time complexity becomes logarithmic in the number of merge operations. The cost in terms of space is one more hidden field per fat exclusive object.

Figure 12 shows how one might implement `transfer` for our FIFO bags. One potential trap is that, if the programmer were to forget the `merge` instruction (line 18), then the code would still be well-typed, and a call to `transfer (b1, b2)`

```
exclusive data tree a =
  | Leaf
  | Node { size: a; left, right: tree a }
```

Figure 13. A type of mutable trees with size annotations

would still succeed at runtime. The program would fail only later on, upon attempting to retrieve out of `b2` an element that was transferred from `b1` to `b2`.

5.7.4 Static groups

Our groups are "dynamic" in the sense that membership in a group is not controlled by the type system, but controlled at runtime. Following other authors [15, 7, 10], one could also offer "static" groups, where the type system keeps track of which object is a member of which group (or "region") and no runtime machinery is required. Static groups can in some situations offer stronger static guarantees of correctness as well as better performance. However, they are less flexible and more complex than dynamic groups. In particular, abandon (known as "focus" [15, 10] or "carving out" [7]) is not permanent, but temporary: when an object is abandoned, the group becomes disabled until the object is returned. (Boyland and Retert [7] do allow "carving" multiple objects out of a group, but this requires the type system to be able to recognize that these objects are distinct.) While in principle we could offer both flavors of groups in a single programming language, we feel that this would have a high cost in terms of conceptual complexity, for little practical benefit.

6. Typestate checking

In this section, we show how the sytem can be used to perform what is usually known as *typestate checking*. This section is presently incomplete.

6.1 In-place tree annotation

In this example, we define an algebraic data type for mutable trees. We present a straightforward recursive traversal which annotates every node in a tree with the size of its subtree. We explain how the typechecker is able to differentiate between an unannotated tree and an annotated one, and to recognize that the traversal function turns one into the other.

We begin with a definition of the algebraic data type of trees (Figure 13). Our trees are binary: a tree either is empty or has a binary node at its root. A node contains pointers to its left and right children. In addition, every node contains a `size` field.

Because we wish to allow constructing nodes whose `size` field is uninitialized, we do not declare that this field has type `int`. Instead, we declare that it has type `a` and we make this type a parameter. Thus, "`tree a`" is the type of a tree where every node has a `size` field of type `a`. This type can be instantiated in several useful ways. For instance, the type "`tree int`" describes trees that carry size information.

```
val size: [a] (consumes t: tree a) ->
          (int | t @ tree int)

let rec size t =
  match t with
  | Leaf {} ->
      0
  | Node { left = left; right = right } ->
      let lsize = size left in
      let rsize = size right in
      let total = lsize + rsize + 1 in
      t.size <- total;
      total
  end
```

Figure 14. In-place tree annotation

The type "`tree ()`" describes trees that definitely do not carry size information. The type "`tree a`", where `a` is a type variable, describes trees where nodes carry information of unknown nature.

Because we wish to be able to update the `size` field, we declare the type `tree` as exclusive. This implies that, at every node, the left and right sub-trees are disjoint. Thus, the type "`tree a`" describes trees, as opposed to DAGs or graphs.

Now, imagine that we have a tree `t` that either does not carry size information or carries stale information (perhaps because we have re-organized its structure by mutating the `left` and `right` fields in some nodes). Both situations are described by the permission "`t @ tree a`", for an appropriate choice of `a`.

We would like to traverse this tree, computing sizes and updating `size` fields in a bottom-up manner. To this end, we define a recursive function, which we name `size` (Figure 14). We intend the function call "`size t`" to return the number of nodes in the tree `t` and as a side effect to update the `size` field of every node in the tree `t`. The type of the function `size` reflects this informal specification. The function requires the permission "`t @ tree a`", but does not return it. Instead, it returns an integer result, together with a new permission for `t`, namely "`t @ tree int`". In short, a call to "`size t`" performs a *strong update*: the "type" (or the "typestate") of the tree `t` changes.

Let us now review how the `size` function is type-checked. At the beginning, just before the `match` construct, the permission "`t @ tree a`" is available. (This is determined by consulting the type signature of the function `size`, which the programmer provides.) This permission allows case analysis to take place, and is refined by the case analysis.

In the `Leaf` branch, the permission "`t @ Leaf {}`" appears. The type-checker verifies that this permission can be folded back to "`t @ tree int`". Thus, the code of the `Leaf` branch satisfies the explicitly-declared result type of the function `size`.

In the `Node` branch, the following permissions are present:

```
t @ Node {
  size: a;
  left = left; right = right
} *
left @ tree a *
right @ tree a
```

In the code that follows, we explicitly name the intermediate results `lsize` and `rsize`. We do so only in order to better explain, step by step, what is going on. If we instead wrote "`let total = size left + size right + 1`", the type-checker would automatically introduce auxiliary variables to stand for the results of the recursive calls.

Is the recursive call "`size left`" legal? Yes. This call requires the permission "`left @ tree a`", which we have. This permission is consumed. After the call, it is replaced with "`left @ tree int`". In addition, the call yields the permission "`lsize @ int`".

Similarly, the recursive call "`size right`" is permitted. This call replaces the permission "`right @ tree a`" with "`right @ tree int`", and yields "`rsize @ int`".

At this point, the left and right sub-trees have been annotated with up-to-date size information, but the root node `t` has not been updated yet. The following permissions are available:

```
t @ Node {
  size: a;
  left = left; right = right
} *
left @ tree int *
right @ tree int *
lsize @ int *
rsize @ int
```

It is interesting to note that these permissions cannot be folded back to "`t @ tree int`". We are in an intermediate state where the data structure at address `t` cannot be described as an instance of the `tree` algebraic data type. Thus, we must now update `t.size` with an integer value, or the type-checker will reject the code. The definition "`let total = ...`" yields the permission "`total @ int`". Then, the instruction "`t.size <- total`" causes the type-checker to update the structural permission that describes `t`. After this instruction, the available permissions are:

```
t @ Node {
  size = total;
  left = left; right = right
} *
left @ tree int *
right @ tree int *
lsize @ int *
rsize @ int *
total @ int
```

These permissions can be folded back to:

```
t @ Node {
  size: int;
```

```
    left, right: tree int
  } *
  lsize @ int *
  rsize @ int *
  total @ int
```

which in turn can be transformed to:

```
  t @ tree int *
  lsize @ int *
  rsize @ int *
  total @ int
```

Thus, the code of the `Node` branch satisfies the explicitly-declared result type of the function `size`.

We have demonstrated that the system supports a strong update of an entire tree structure. Of course, this is an "easy" example: there is no sharing or aliasing. Nevertheless, we believe that this example illustrates the power and relative simplicity of the system.

7. Future features

In this section, we briefly discuss a few features which we believe can be fairly easily added to *Mezzo*, and which we plan to include.

7.1 Arrays

Arrays are compact: their use can lead to space savings and improved data locality. Furthermore, in a high-level language, arrays are the only way of exploiting the machine's ability to perform address arithmetic and "random" memory accesses. For these reasons, arrays play a central role in many efficient algorithms.

It seems straightforward to extend *Mezzo* with support for arrays. We would distinguish two types of arrays, namely an (exclusive) type of mutable arrays and a (duplicable) type of immutable arrays. The former would be equipped with five main operations, namely creation, read, update, length query, and transformation into an immutable array. The latter would support only two of these operations, namely reading the array and retrieving its length.

Because the last two operations are supported both by mutable arrays and by immutable arrays, we would arrange for these operations to be polymorphic in the mutability of the array to which they are applied.

Because retrieving a value out of an array duplicates this value, the element type of an array would be required to be duplicable. Arrays of elements of arbitrary type ("arrays that own their elements", so to speak) could then be defined by the user in terms of primitive arrays and adoption and abandon. This could be done as part of the standard library.

7.2 Concurrency

Up to this point, we have illustrated the use of permissions in a sequential setting. We have explained that permissions forbid certain programming errors (undesired aliasing, representation exposure, etc.) and allow certain new idioms (delayed initialization, etc.).

```
type lock :: PERM -> TYPE
fact [p :: PERM] duplicable (lock p)
val create: [p :: PERM] () -> lock p
val acquire: [p :: PERM] lock p -> (| p)
val release: [p :: PERM]
  (lock p | consumes p) -> ()
```

Figure 15. A signature for first-class locks

However, the single strongest argument in favor of using permissions is perhaps the fact that, in a shared-memory concurrent setting, this discipline guarantees the absence of race conditions between threads. In other words, every valid *Mezzo* program is data-race-free.

Data-race freeness is a very desirable property, because it allows one to work in a simple and traditional memory model, as opposed to a "relaxed" memory model [5]. Yet, in the absence of a mechanically-enforced discipline, it is notoriously difficult to achieve.

Why is it the case that *Mezzo* programs are data-race-free? In the absence of adoption and abandon (§5), this property is an immediate consequence of the fact that our write permissions are exclusive: a write permission and a read permission for the same object can never co-exist. Basic separation logic [22] enjoys this property for the same reason. In the presence of adoption and abandon, things are slightly more subtle. The instruction "`take y from x`" reads the field `y.adopter`, yet does not require an exclusive permission for the object `y`. It requires only "`y @ dynamic`", a duplicable permission, which can co-exist with an exclusive permission for `y`. Thus, there is a possibility that one thread may attempt to read `y.adopter` while another thread attempts to write it. More precisely, the first thread, say "thread 1", must be in the process of executing "`take y from x1`", while the second thread, say "thread 2", must be in the process of executing either "`give y to x2`" (which changes `y.adopter` from `null` to `x2`) or "`take y from x2`" (which involves changing `y.adopter` from `x2` to `null`). This implies that thread 1 must have an exclusive permission for `x1` and that thread 2 must have an exclusive permission for `x2`. Thus, `x1` and `x2` must be distinct addresses. This means that the outcome of the test "`y.adopter == x1`" performed by thread 1 is unaffected by the write instruction performed by thread 2. Whether the value read by thread 1 is `null` or `x2`, the test must fail. Thus, there is a race condition on `y.adopter`, but it is possible to argue that this is a benign race, one that does not cause non-deterministic behavior. In short, the implementation of adoption and abandon in terms of reads and writes that was presented earlier (§5) remains correct in a concurrent setting.

In the absence of any primitive synchronization operations, data-race-freeness means that threads are completely isolated from one another. Naturally, for shared memory to be useful, there must exist one or more mechanisms that allow threads to communicate with one another and that are not considered as causing data races.

Locks are one such mechanism. Concurrent separation logic [21] has shown how a lock can be used to mediate access to a permission `p` by several threads. The idea is, a thread that successfully acquires the lock acquires the permission `p` at the same time; and, when releasing the lock, this thread must give up the permission `p`. In the terminology of separation logic, `p` is the "resource invariant" that the lock protects. This approach extends to the case where locks are first-class, dynamically-allocated values [16, 17, 8].

It seems straightforward to extend *Mezzo* with support for primitive locks. A slightly simplified version of the interface that might be offered by these locks is presented in Figure 15. In short, the type "`lock p`" of locks is parameterized with a permission `p`. A specific instance of `p` is chosen by the user at lock creation time. The type "`lock p`" is duplicable: this allows multiple threads to attempt to acquire a single lock at the same time. Acquiring a lock of type "`lock p`" produces the permission `p`; releasing such a lock consumes this permission. This interface guarantees that "well-typed programs do not go wrong", but does not guarantee the absence of deadlocks.

Locks are useful not only because they allow synchronization, but also because they enable "hidden state". In the absence of locks, whenever a function needs access to a mutable memory area, it must explicitly require a permission. In the presence of locks, however, it becomes possible for a function of type "`() -> ()`" to have a "hidden" side effect. Indeed, because locks are duplicable, it is permitted for a closure to capture the address of a lock.

Locks are one synchronization mechanism among many others. We hope to be able to support other mechanisms as well, such as communication channels along which permissions can travel [14, 29].

7.3 Permissions within algebraic data structures

Structural permissions carry accurate information about an object's tag and fields. They evolve with time: they are updated by the field update and tag update instructions (§2.7). For these reasons, one might claim that the system, as presented so far, is able to perform "typestate checking", that is, to keep precise track of the manner in which the "type" or "state" of an object evolves over time.

To some extent, this is true. Yet, one more feature seems required in order to achieve sufficient expressive power for realistic "typestate checking" applications. This feature is the ability for a dynamic test on an object `x` to produce new or refined permissions about another object `y`. Bierhoff and Aldrich [3] use Java iterators to illustrate this phenomenon. The method `hasNext` requires a permission for an iterator in an arbitrary state, and produces a Boolean outcome. By examining whether this Boolean value is `true` or `false`, one recovers a permission for the iterator in a specific state, which is either "`available`" or "`end`". Thanks to this, in the `true` branch, it is permitted to invoke the iterator's `next` method, whose precondition is that the iterator be in the "`available`" state.

In the system that we have presented so far, the `match` construct does perform permission refinement (§4.1.2), albeit only in a limited manner: a dynamic test on an object `x` produces a refined permission *for this object*. In order for a test on `x` to be able to produce an arbitrary permission, it seems natural to add the ability for an algebraic data type to have permission components, in the same way that values can be packaged with permissions (§3.1). The only novelty, with respect to §3.1, is that the permission that is packaged with an object can now depend on the object's tag. This allows us to define a type of permission-carrying Boolean values, as follows:

```
data outcome (p :: PERM) (q :: PERM) =
| No  {| p }
| Yes {| q }
```

A value of type "`outcome p q`" is just a tag: either `No` or `Yes`. If it is `No`, then it carries the permission `p`; otherwise, it carries the permission `q`. A `match` construct can be used to distinguish between these cases and gain access to the corresponding permission. It seems straightforward to extend the system with support for this feature.

8. Conclusion

We have presented the design of a high-level functional and imperative programming language where the central concept of "type" is replaced with the more powerful concept of "permission". This design strives to achieve a balance between simplicity and expressiveness by marrying a static discipline of permissions and a dynamic mechanism of adoption and abandon. We believe that the system is sound in the sense that well-typed programs cannot go wrong (unless an abandon operation fails) and are data-race-free. We have used a wide range of examples in order to illustrate the expressiveness of the language. Ongoing and future work includes implementing a type-checker and compiler, proving that the system is sound, and gaining deeper practical experience with the use of the language.

References

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Companion to the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1015–1022, 2009.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, 2007.

[4] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219. Springer, 2009.

[5] Hans-J. Boehm and Sarita V. Adve. You don't know jack about shared variables or memory models. *Communications of the ACM*, 55(2):48–54, 2012.

[6] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

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

[8] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. A step-indexed Kripke model of separation logic for storable locks. *Electronic Notes in Theoretical Computer Science*, 276:121–143, 2011.

[9] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010.

[10] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, 2008.

[11] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275, 1999.

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

[13] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[14] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190, 2006.

[15] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, 2002.

[16] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. Technical Report MSR-TR-2007-39, Microsoft Research, 2007.

[17] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[18] Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. Extended alias type system using separating implication. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2011.

[19] Yasuhiko Minamide. A functional representation of data structures with a hole. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 75–84, 1998.

[20] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 557–570, 2012.

[21] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[23] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

[24] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In *ACM International Conference on Functional Programming (ICFP)*, pages 251–260, 1998.

[25] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[26] Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.

[27] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 447–458, 2011.

[28] Thomas Tuerk. Local reasoning about while-loops. Unpublished, 2010.

[29] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2009.

[30] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation (TIC)*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer, 2000.

[31] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2011.