The Design and Formalization of Mezzo, a Permission-Based Programming Language

THIBAUT BALABONSKI and FRANÇOIS POTTIER and JONATHAN PROTZENKO, INRIA

The programming language Mezzo is equipped with a rich type system that controls aliasing and access to mutable memory. We present the language at large, via an introduction along with some motivating examples. We then present a modular formalization of Mezzo’s core type system, in the form of a concurrent λ-calculus, which we extend with references and locks. We prove that well-typed programs do not go wrong and are data-race free. Our definitions and proofs are machine-checked.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; Polymorphism; Recursion; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Pre- and post-conditions

General Terms: Languages, Theory

Additional Key Words and Phrases: Aliasing, Concurrency, Ownership, Separation logic, Side effects, Static type systems

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

A strongly-typed programming language rules out certain programming mistakes by ensuring at compile-time that every operation is applied to arguments of appropriate nature. As per Milner’s slogan, “well-typed programs do not go wrong”. If one wishes to obtain stronger static guarantees, one must usually turn to static analysis or program verification techniques. For instance, separation logic [Reynolds 2002] can prove that private state is properly encapsulated; concurrent separation logic [O’Hearn 2007] can prove the absence of interference between threads; and, in general, program logics can prove that a program meets its specification.

The programming language Mezzo [Pottier and Protzenko 2013; Balabonski et al. 2014] is equipped with a static discipline that goes beyond traditional type systems and incorporates some of the ideas of separation logic. The Mezzo type-checker reasons about aliasing and ownership. This increases expressiveness, for instance by allowing gradual initialization, and rules out more errors, such as representation exposure or data races. Mezzo is descended from ML: its core features are immutable local variables, possibly-mutable heap-allocated data, and first-class functions. It also features a new mechanism, adoption and abandon, which marries the static ownership discipline with dynamic ownership tests.

In this paper, we offer a comprehensive overview of Mezzo, including an informal, user-level presentation of the language and a formal, machine-checked presentation of its meta-theory. This unifies and subsumes the conference papers cited above. Furthermore, we revisit the theory of adoption and abandon, which was presented informally in the first conference paper, and was absent from the second conference paper. Our new account of adoption and abandon is not only machine-checked, but also simpler and more expressive than that of the conference paper.

We begin with two short illustrative examples. The first one concerns a type of write-once references and shows how Mezzo guarantees that the client follows the intended usage protocol. The second example is a racy program, which the type system rejects. We show how to fix this ill-typed program by introducing a lock.
open worref

val f (x: frozen int, y; frozen int) : int =
  get x + get y

val _ : int =
  let r = new () in
  set (r, 3);
  f (r, r)

Fig. 1. Using a write-once reference. The Mezzo type system guarantees that the user must call set before using get, and can call set at most once.

A usage protocol. A write-once reference is a memory cell that can be assigned at most once and cannot be read before it has been initialized. Fig. 1 shows some client code that manipulates a write-once reference. The code refers to the module worref, whose implementation we show later on (§2.1).

At line 7, we create a write-once reference by calling worref::new. (Thanks to the declaration open worref, one can refer to this function by the unqualified name new.) The local variable r denotes the address of this reference. In the eyes of the type-checker, this gives rise to a permission, written r @ writable. This permission has a double reading: it describes the layout of memory (i.e., “the variable r denotes the address of an uninitialized memory cell”) and grants exclusive write access to this memory cell. That is, the type constructor writable denotes a uniquely-owned writable reference, and the permission r @ writable is a unique token that one must possess in order to write r.

Permissions are tokens that exist at type-checking time only. Many permissions have the form x @ t, where x is a program variable and t is a type. At a program point where such a permission is available, we say informally that “x has type t (now)”. Type-checking in Mezzo is flow-sensitive: at each program point, there is a current permission, which represents our knowledge of the program state at this point, and our rights to alter this state. The current permission is typically a conjunction of several permissions. The conjunction of two permissions p and q is written p * q.

Permissions replace traditional type assumptions. A permission r @ writable superficially looks like a type assumption r : writable. However, a type assumption would be valid everywhere in the scope of r, whereas a permission should be thought of as a token: it can be passed from caller to callee, returned from callee to caller, passed from one thread to another, etc. If one gives up this token (say, by assigning the reference), then, even though r is still in scope, one can no longer write it.

At line 8, we exercise our right to call the set function, and write the value 3 to the reference r. In the eyes of the type-checker, this call consumes the token r @ writable, and instead produces another permission, r @ frozen int. This means that any further assignment is impossible: the set function requires r @ writable, which we no longer have. Thus, the reference has been rendered immutable. This also means that the get function, which requires the permission r @ frozen int, can now be called. Thus, the type system enforces the desired usage protocol.

The permissions r @ writable and r @ frozen int are different in one important way. The former denotes a uniquely-owned, writable heap fragment. It is affine: once it has been consumed by a call to set, it is gone forever. The latter denotes an immutable heap fragment. It is safe to share it: this permission is duplicable. If one can get ahold of such a permission, then one can keep it forever (i.e., as long as r is in scope) and pass copies of it to other parts of the program, if desired. Such a permission behaves very much like a traditional type assumption r : frozen int.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

1. open thread
2. val r = newref 0
3. val f (| r @ ref int) : () =
   r := !r + 1
4. val () =
5.   spawn f; spawn f

Fig. 2. Ill-typed code. The function f increments the global reference r. The main program spawns two threads that call f. There is a data race: both threads may attempt to modify r at the same time.

At line 9, we call a function f with as parameter a pair of two copies of r. This function, defined at line 3, requires a pair of two arguments x and y and the right to use each of these at type frozen int. It returns an integer, and then has type (frozen int, frozen int) -> int (the next example will explain in detail what this type means, beyond the usual intuition). At the call at line 9, the duplicable permission r @ frozen int is implicitly copied, so as to justify the fact that the pair (r, r) has type (frozen int, frozen int), as required by the type of f. This in turn allows several calls to get.

A race. We now consider the tiny program in Fig. 2. This code exhibits a data race, hence is incorrect, and is rejected by the type system. Let us explain how the type-checker determines that this program must be rejected.

At line 3, we allocate a (classic) reference r, thus obtaining a new permission r @ ref int.

The function f at line 4 takes no argument and returns no result. Its type is not just () -> (), though. Because f needs access to r, it must explicitly request the permission r @ ref int and return it. (The fact that this permission is available at the definition site of f is not good enough: a closure cannot capture a non-duplicable permission.) This is declared by the type annotation. Thus, at line 6, in conjunction with r @ ref int, we have a new permission, f @ (| r @ ref int) -> (). This means that f is a function of no argument and no result (at runtime), which (at type-checking time) requires and returns the permission r @ ref int.

Before going further, we must say a little more about the syntax of types. The type t | p denotes a package of a value of type t and the permission p. It can be thought of as a pair; yet, because permissions do not exist at runtime, a value of type t | p and a value of type t have the same runtime representation. We write (| p) for (() | p), where () is the unit type. Furthermore, by default, a permission that appears in the domain of a function type is implicitly repeated in the codomain. By this convention, f @ (| r @ ref int) -> () means that f requires and returns the permission r @ ref int. When one wishes to indicate that a function requires some permission but does not return it, one must precede that permission with the keyword consumes.

On line 7, there is a sequencing construct. The second call to spawn is type-checked using the permissions that are left over after the first spawn. A call spawn f requires two permissions: a permission to invoke the function f, and r @ ref int, which f itself requires. It does not return these permissions: they are transferred to the spawned thread. Thus, in line 7, between the two spawns, we no longer have a permission for r. (We still have f @ (| r @ ref int) -> (), as it is duplicable.) Therefore, the second spawn is ill-typed. The racy program of Fig. 2 is rejected.

This behavior is to be contrasted with that of the earlier example. In Fig. 1, the permission r @ frozen int, which get requires, is duplicable. We can therefore obtain two copies of it and justify the call to a function that requires a pair of frozen int and/or several calls to get.
A fix. In order to fix this program, one must introduce enough synchronization so as to eliminate the race. A common way of doing so is to introduce a lock and place all accesses to \( r \) within critical sections. In Mezzo, this can be done, and causes the type-checker to recognize that the code is now data-race free. In fact, this common pattern can be implemented as a polymorphic, higher-order function, \textit{hide} (Fig. 3).

In Fig. 3, \( f \) is a parameter of \textit{hide}. It has a visible side effect: it requires and returns a permission \( s \). When \textit{hide} is invoked, it creates a new lock \( l \), whose role is to govern access to \( s \). At the beginning of line 7, we have two permissions, namely \( s \) and \( f \circ (\text{consumes } a \mid s) \rightarrow b \). At the end of line 7, after the call to \textit{lock}:\textit{new}, we have given up \( s \), which has been consumed by the call, and we have obtained \( l \circ \text{lock } s \). The lock is created in the “released” state, and the permission \( s \) can now be thought of as owned by the lock.

At line 8, we construct an anonymous function. This function does not request any permission for \( f \) or \( l \) from its caller: according to its header, the only permission that it requires is \( x \circ a \). Nevertheless, the permissions \( f \circ (\text{consumes } a \mid s) \rightarrow (b \mid s) \) and \( l \circ \text{lock } s \) are available in the body of this anonymous function, because they are duplicable, and a closure is allowed to capture a duplicable permission.

The fact that \( l \circ \text{lock } s \) is duplicable is a key point. Quite obviously, this enables multiple threads to compete for the lock. More subtly, this allows the lock to become hidden in a closure, as illustrated by this example. Let us emphasize that \( s \) itself is typically \textit{not} duplicable (if it were, we would not need a lock in the first place).

The anonymous function at line 8 does not require or return \( s \). Yet, it needs \( s \) in order to invoke \( f \). It obtains \( s \) by acquiring the lock, and gives it up by releasing the lock. Thus, \( s \) is available only to a thread that has entered the critical section. The side effect is now hidden, in the sense that the anonymous function has type \((\text{consumes } a) \rightarrow b\), which does not mention \( s \).

It is easy to correct the code in Fig. 2 by inserting the redefinition \texttt{val f = hide f} before line 6. This call consumes \( r \circ \text{ref } \text{int} \) and produces \( f \circ () \rightarrow () \), so the two \texttt{spawn} instructions are now type-correct. Indeed, the modified code is race-free.

Outline. In this paper, we give an in-depth presentation of Mezzo. We start off with a tutorial introduction to Mezzo (§2). We come back to the above examples and informally explain how they are type-checked. We move on to more advanced examples involving lists and trees. We demonstrate a few programming patterns that cannot be type-checked in ML, such as list concatenation in destination-passing style. We conclude this tutorial introduction with the example of a mutable graph data structure, which involves arbitrary aliasing. This example exploits adoption and abandon, a mechanism that defers some of the ownership tests to runtime.

```plaintext
1 open lock
2
3 val hide [a, b, s : perm] ( f : (consumes a | s) → b | consumes s
4 ) : (consumes a) → b =
5 let l : lock s = new () in
6 fun (consumes x : a) : b = acquire l;
7 let y = f x in
8 release l;
9 y
```

Fig. 3. The polymorphic, higher-order function \textit{hide} takes a function \( f \) of type \((\text{consumes } a | s) \rightarrow (b | s)\), which means that \( f \) needs access to a piece of state represented by the permission \( s \). \textit{hide} requires \( s \), and consumes it. It returns a function of type \((\text{consumes } a) \rightarrow b\), which does not require \( s \), hence can be invoked by multiple threads concurrently. The type variables \( a \) and \( b \) have kind \textit{type}. The square brackets denote universal quantification.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

The surface language that we expose to the user differs slightly from the internal language that the type-checker uses, and whose meta-theory we have formalized. The gap is not very large: it is mostly a matter of desugaring the syntax of types. We give an informal description of the translation of surface Mezzo down to Core Mezzo (§3).

Finally, we give a modular formalization of the core layers of Mezzo. We identify a kernel layer: a concurrent, call-by-value $\lambda$-calculus (§4). In its typed version, it is an affine, polymorphic, value-dependent system, which enjoys type erasure: values exist at runtime, whereas types and permissions do not. Although this calculus does not have explicit side effects, we endow it with an abstract notion of machine state, and we organize the proof of type soundness in such a way that the statements of the main lemmas need not be altered as we introduce new forms of side effects. The next three layers are heap-allocated references (§5), locks (§6), and adoption and abandon (§7). These three layers are almost independent of one another. There is one dependency: adoption and abandon is piggybacked on top of heap-allocated state. Yet, we are able to structure the meta-theory in such a way that there is very little interaction between these two features. Our definitions and proofs are machine-checked and are available online [Balabonski and Pottier 2014].

The paper ends with an overview of the features of Mezzo that we could not describe here (§8) and a discussion of the related work (§9).

2. A MEZZO TUTORIAL

In this section, we expand on the examples that we mentioned earlier (§1). We give a more thorough introduction to permissions, present more examples, including a few typical library functions, and show how to deal with the pervasive problem of arbitrary aliasing over mutable data structures.

2.1. Write-once references

In the introduction (§1), we showed a tiny client of the module `woref` of write-once references. We now explain how this module is implemented. Its code appears in Fig. 4.

```
data mutable writable =
  Writable { contents: () }

data frozen a =
  Frozen { contents: (a | duplicable a) }

val new () : writable =
  Writable { contents = () }

val set [a] (consumes r: writable, x: a | duplicable a)
  : (| r @ frozen a) =
  r.contents <- x;
  tag of r <- Frozen

val get [a] (r: frozen a) : a =
  r.contents
```

Fig. 4. Implementation of write-once references

To be or not to be duplicable. The type `writable` (line 1) describes a mutable heap-allocated block. Such a block contains a tag field (which must contain the tag `Writable`, as no other data constructors are defined for this type) and a regular field, called
contents, which has unit type. The function new (line 7) allocates a fresh memory block of type writable and initializes its contents field with the unit value. A call to this function, such as `let r = woref::new() in ...`, produces a new permission `r @ writable`.

The definition of writable contains the keyword mutable. This causes the type-checker to regard the type writable as affine (i.e., non-duplicable), as well as every permission of the form `r @ writable`. This ensures that `r @ writable` represents exclusive access to the memory block at address x. If one attempts to duplicate this permission (for instance, by writing down the static assertion `assert r @ writable * r @ writable`, or by attempting to call set `(r, ...)` twice), the type-checker rejects the program.

The parameterized data type frozen a (line 4) describes an immutable heap-allocated block. Such a block contains a tag field (which must contain the tag Frozen) and a regular field, also called contents, which has type `(a | duplicable a)`. This is a type of the form `t | p`: indeed, a is a type, while duplicable a is a permission. This means that the value stored at runtime in the contents field has type a, and is logically accompanied by a proof that the type a is duplicable.

Why do we impose the constraint duplicable a as part of the definition of the type frozen a? The reason is, a write-once reference is typically intended to be shared after it has been initialized. (If one did not wish to share it then one could use a standard read/write, uniquely-owned reference.) Thus, its content is meant to be accessed by multiple readers. This is permitted by the type system only if the type a is duplicable. Technically, the constraint duplicable a could be imposed either when the write-once reference is initialized, or when it is read. We choose the former approach because it is simpler to explain. The latter would work just as well, and would offer a little extra flexibility.

The definition of frozen does not contain the keyword mutable, so a block of type frozen a is immutable. Thus, it is safe to share read access to such a block. Furthermore, because we have imposed the constraint duplicable a, it is also safe to share the data structure of type a whose address is stored in the contents field. In other words, by inspection of the definition, the type-checker recognizes that the type frozen a is duplicable as a whole. This means that a write-once reference can be shared after it has been initialized.

Changing states: strong updates. The use of the consumes keyword in the type of set (line 10) means that the caller of set must give up the permission `r @ writable`. In exchange, the caller receives a new permission for `r`, namely `r @ frozen a` (line 11). One may say informally that the type of `r` changes from “uninitialized” to “initialized and frozen”.

The code of set is in two lines. First, the value x is written to the field `r.contents` (line 12). After this update, the memory block is described by the permission `r @ Writable { contents: a }`. In this intermediate state, the permission corresponds neither to writable nor to frozen a.

Then, the tag of `r` is changed from Writable to Frozen: this is a tag update (line 13). This particular tag update instruction is ghost code: it has no runtime effect, because both Writable and Frozen are represented at runtime as the tag 0. This pseudo-instruction is just a way of telling the type-checker that our view of the memory block `r` changes. After the tag update instruction, this block is described by the permission `r @ Frozen { contents: a }`.

1Contrary to OCaml, Mezzo allows two user-defined types to have a field by the same name.
abstract writable
abstract frozen a

fact duplicable (frozen a)

val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
-> (| r @ frozen a)
val get: [a] frozen a -> a

Fig. 5. Interface of write-once references

This permission can be combined with the permission duplicable a (which exists at this point, because set requires this permission from its caller) so as to yield r @ Frozen { contents: (a | duplicable a) }. This is the right-hand side of the definition of the type frozen a. By folding it, one obtains r @ frozen a. Thus, the permissions available at the end of the function set match what has been advertised in the header (line 11).

In general, the tag update instruction allows changing the type of a memory block to a completely unrelated type, with two restrictions: (i) the block must initially be mutable; and (ii) the old and new types must have the same number of fields. This instruction is compiled down to either a single write to the tag field, or nothing at all, as is the case above. This case of tag update by a no-op is materialized in §5 by a ghost instruction.

An interface for woref. Mezzo currently offers a simple notion of module, or unit. Each module has an implementation file (whose extension is .mz) and an interface file (whose extension is .mzi). This system supports type abstraction as well as separate type-checking and compilation. It is inspired by OCaml and by its predecessor Caml-Light.

The interface of the module woref is shown in Fig. 5.

The type writable is made abstract (line 1) so as to ensure that set is the only action that can be performed with an uninitialized reference. If the concrete definition of writable was exposed, it would be possible to read and write such a reference directly, without going through the functions offered by the module woref.

The type frozen is also made abstract (line 2). One could expose its definition without endangering the intended usage protocol. Nevertheless, it is good practice to hide the details of its implementation; this may facilitate future evolutions.

The fact that frozen a is a duplicable type is published (line 3). In the absence of this declaration, frozen a would by default be regarded affine, so that sharing access to an initialized write-once reference would not be permitted. This fact declaration is implicitly universally quantified in the type variable a. One can think of it as a universally quantified permission, [a] duplicable (frozen a), that is declared to exist at the top level. This permission is itself duplicable, hence exists everywhere and forever.

The remaining lines in Fig. 5 declare the types of the functions new, get, and set, without exposing their implementation. In the type of set, the first argument r must be named (line 5), because we wish to refer to it in the result type (line 6). In a function header or in a function type, the name introduction form r: t binds the variable r and at the same time requests the permission r @ t. In contrast, in the permission r @ t, the variable r occurs free. The second argument of set, x, need not be named; we name it anyway (line 5), for the sake of symmetry.

2.2. Lists
The example of write-once references has allowed us to discuss a number of concepts, including affine versus duplicable permissions, mutable versus immutable memory

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
blocks, and strong updates. References are, however, trivial data structures, in the sense that their exact shape is statically known. We now turn to lists. Lists are data structures of statically unknown length, which means that many functions on lists must be recursive. Lists are representative of the more general case of tree-structured data.

The algebraic data type of lists, \texttt{list a}, is defined in a standard way (Fig. 6). This definition does not use the keyword \texttt{mutable}. These are standard immutable lists, that is, lists with an immutable spine. The list elements may be mutable or immutable, depending on how the type parameter \(a\) is instantiated.

\textit{Concatenation.} Our first example of an operation on lists is concatenation. There are several ways of implementing list concatenation in Mezzo. We begin with the function \texttt{append}, also shown in Fig. 6, which is the most natural definition.

The type of \texttt{append} (line 5) states that this function takes two arguments \(xs\) and \(ys\), together with the permissions \(xs @ list a\) and \(ys @ list a\), and produces a result, say \(zs\), together with the permission \(zs @ list a\). The \texttt{consumes} keyword indicates that the permissions \(xs @ list a\) and \(ys @ list a\) are not returned: the caller must give them up. Before discussing the implications of this fact, let us first explain how \texttt{append} is type-checked.

At the beginning of line 6, the permission \(xs @ list a\) guarantees that \(xs\) is the address of a list, i.e., a memory block whose tag field contains either \texttt{Nil} or \texttt{Cons}. This justifies the \texttt{match} construct: it is safe to read \(xs\)'s tag and to perform case analysis.

Upon entry in the first branch, at the beginning of line 8, the permission \(xs @ list a\) has been refined into \(xs @ Nil\). We refer to the latter as a \textit{structural permission}. It is more precise than the former; it tells us not only that \(xs\) is a list, but also that its tag must be \texttt{Nil}. This knowledge, it turns out, is not needed here: \(xs @ Nil\) is not exploited when type-checking this branch. On line 8, we return the value \(ys\). The permission \(ys @ list a\) is used to justify that this result has type \(list a\), as advertised in the function header. This consumes \(ys @ list a\), which is an affine permission.

Upon entry in the second branch, at the beginning of line 10, our knowledge about \(xs\) also increases. The permission \(xs @ list a\) is refined into \(xs @ Cons\ \{ head: a; tail: list a \}\). Like \(xs @ Nil\), this is a structural permission. It is obtained by looking up the definition of the data type \(list a\) and specializing it for the tag \(Cons\).

The pattern \(Cons\ \{ head; tail \}\) on line 9 involves a pun: it is syntactic sugar for \(Cons\ \{ head = head; tail = tail \}\), which binds the variables \(head\) and \(tail\) to the contents of the fields \(xs.head\) and \(xs.tail\), respectively. Thus, we now have two names, \(head\) and \(tail\), to refer to the values stored in these fields. This allows the type-checker to decompose the structural permission above into a conjunction of three atomic permissions:

\[
\begin{align*}
xss @ Cons \{ head: =head; tail: =tail \} & * \\
head @ a & * \\
\text{tail} @ list a & 
\end{align*}
\]

The first conjunct describes just the memory block at address \(xs\). It indicates that this block has tag \(Cons\), that its head field contains the value \(head\), and that its tail field contains the value \(tail\). The types \(=head\) and \(=tail\) are \textit{singleton types} [Smith et al. 2000]: each of them is inhabited by just one value. This first conjunct can also be written \(xs @ Cons\ \{ head = head; tail = tail \}\). In the following, this permission is not used, so we do not repeat it, even though it remains available until the end.

The second conjunct describes just the first element of the list, that is, the value \(head\). It guarantees that this value has type \(a\), so to speak, or more precisely, that we...
data list a =
  | Nil
  | Cons { head: a; tail: list a }

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

Fig. 6. Definition of lists and list concatenation

data mutable cell a =
  Dummy | Cell { head: a; tail: () }

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

val append [a] (consumes (xs: list a, ys: list a)) : list a =
  match xs with
  | Nil ->
  | Cons ->
    let dst = Cell { head = xs.head; tail = () } in
    appendAux (dst, xs.tail, ys);
    dst
end

Fig. 7. List concatenation in tail-recursive style
have permission to use it at type a. The last conjunct describes just the value \( \text{tail} \), and means that we have permission to use this value as a list of elements of type \( a \).

In order to type-check the code on line 10, the type-checker automatically expands it into the following form, where every intermediate result is named:

\[
\text{let } ws = \text{append} (\text{tail}, ys) \text{ in } \\
\text{let } zs = \text{Cons} \{ \text{head} = \text{head}; \text{tail} = ws \} \text{ in } \\
zs
\]

The call \( \text{append} (\text{tail}, ys) \) on line 110 requires and consumes the permissions \( \text{tail} @ \text{list a} \) and \( \text{ys} @ \text{list a} \). It produces the permission \( ws @ \text{list a} \). Thus, after this call, at the beginning of line 111, the current permission is:

\[
\text{head} @ a * \\
ws @ \text{list a}
\]

The permission \( \text{head} @ a \), which was not needed by the call \( \text{append} (\text{tail}, ys) \), has been implicitly preserved. In the terminology of separation logic, it has been “framed out” during the call.

The memory allocation expression \( \text{Cons} \{ \text{head} = \text{head}; \text{tail} = ws \} \) on line 111 requires no permission at all, and produces a structural permission that describes the newly-allocated block in an exact manner. Thus, after this allocation, at the beginning of line 112, the current permission is:

\[
\text{head} @ a * \\
ws @ \text{list a} * \\
zs @ \text{Cons} \{ \text{head} = \text{head}; \text{tail} = ws \}
\]

At this point, since \( \text{append} \) is supposed to return a list, the type-checker must verify that \( zs \) is a valid list. It does this in two steps. First, the three permissions above can be conflated into one composite permission:

\[
zs @ \text{Cons} \{ \text{head} = a; \text{tail} = \text{list a} \}
\]

This step involves a loss of information, as the type-checker forgets that \( zs\text{.head} \) is head and that \( zs\text{.tail} \) is \( ws \). Next, the type-checker recognizes the definition of the data type \( \text{list} \), and folds it:

\[
zs @ \text{list a}
\]

This step also involves a loss of information, as the type-checker forgets that \( zs \) is a \( \text{Cons} \) cell. Nevertheless, we obtain the desired result: \( zs \) is a valid list. So, \( \text{append} \) is well-typed.

*When is a list duplicable?* It is natural to ask: what is the status of the permission \( xs @ \text{list t} \), where \( t \) is a type? Is it duplicable or affine?

Since the list spine is immutable, it is certainly safe to share (read) access to the spine. What about the list elements, though? If the type \( t \) is duplicable, then it is safe to share access to them, which means that it is safe to share the list as a whole. Conversely, if the type \( t \) is not duplicable, then \( \text{list t} \) must not be duplicable either. In short, the \textit{fact} that describes lists is:

\[
\text{fact duplicable } a \Rightarrow \text{duplicable } (\text{list } a)
\]

This fact is inferred by the type-checker by inspection of the definition of the type \( \text{list} \). If one wished to export \( \text{list} \) as an abstract data type, this fact could be explicitly written down by the programmer in the interface of the \( \text{list} \) module.
By exploiting this fact, the type-checker can determine, for instance, that `list int` is duplicable, because the primitive type `int` of machine integers is duplicable; and that `list (ref int)` is not duplicable, because the type `ref t` is affine, regardless of its parameter `t`.

A type variable `a` is regarded as affine, unless the permission `duplicable a` happens to be available at this program point. In the definition of `append` (Fig. 6), no assumption is made about `a`, so the types `a` and `list a` are considered affine.

To consume, or not to consume. Why must `append` consume the permissions `xs @ list a` and `ys @ list a`? Could it, for instance, not consume the latter?

In order to answer this question, let us attempt to change the type of `append` to `[a] (consumes xs: list a, ys: list a) -> list a`, where the `consumes` keyword bears on `xs` only. Recall that, by convention, the absence of the `consumes` keyword means that a permission is requested and returned. In other words, the above type is in fact syntactic sugar for the following, more verbose type:

```
[a] (consumes xs: list a, consumes ys: list a)
  -> (list a | ys @ list a)
```

It is not difficult to understand why `append` does not have this type. At line 8, where `ys` is returned, one would need two copies of the permission `ys @ list a`: one copy to justify that the result of `append` has type `list a`, and one copy to justify that the argument `ys` still has type `list a` after the call. Because the type `list a` is affine, the type-checker rejects the definition of `append` when annotated in this way.

A similar (if slightly more complicated) analysis shows that the `consumes` annotation on `xs` is also required.

These results make intuitive sense. The list `append (xs, ys)` shares its elements with the lists `xs` and `ys`. When the user writes `let zs = append (xs, ys) in ...`, she cannot expect to use `xs`, `ys` and `zs` as if they were lists with disjoint sets of elements. If the permission `xs @ list (ref int) * ys @ list (ref int)` exists before the call, then, after the call, this permission is gone, and `zs @ list (ref int)` is available instead. The integer references are now accessible through `zs`, but are no longer accessible through `xs` or `ys`.

The reader may be worried that this discipline is overly restrictive when the user wishes to concatenate lists of duplicable elements. What if, for instance, the permission prior to the call is `xs @ list int * ys @ list int`? There is no danger in sharing an integer value: the type `int` is duplicable. It would be a shame to lose the permissions `xs @ list int` and `ys @ list int`. Fortunately, these permissions are duplicable. So, even though `append` requests them and does not return them, the caller is allowed to copy each of them, pass one copy to `append`, and keep the other copy for itself. The type-checker performs this operation implicitly and automatically. As a result, after the call, the current permission is `xs @ list int * ys @ list int * zs @ list int`: all three lists can be used at will.

Technically, this phenomenon may be summed up as follows. In a context where the type `t` is known to be duplicable, the function types `(consumes t) -> u` and `t -> u` are equivalent, that is, subtypes of one another. It would be premature to prove this claim at this point; let us simply say that one direction is obvious, while the other direction follows from the frame rule and the duplication rule.

As a corollary, the universal type `[a] (consumes (list a, list a)) -> list a`, which is the type of `append` in Fig. 6, is strictly more general than the type `[a] (list a, list a | duplicable a) -> list a`, where the `consumes` keyword has been removed, but the type `a` of the list elements is required to be duplicable. In particular, this explains why `append` effectively does not consume its arguments when they have duplicable type.
List concatenation in tail-recursive style. The append function that we have discussed so far is a direct translation into Mezzo of the standard definition of list concatenation in ML. It has one major drawback: it is not tail-recursive, which means that it needs a linear amount of space on the stack, and may well run out of space if the operating system places a low limit on the size of the stack.

One can work around this problem by performing concatenation in two passes: that is, in OCaml, by composing List.rev and List.rev_append. Performing concatenation in one pass and in constant stack space requires breaking the ML type discipline. The authors of the OCaml library “Batteries included” [2014] have chosen to do so: they implement concatenation (and other operations on lists) by using an unsafe type cast.

Why is this code not well-typed in ML? There are two (related) reasons. 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 type system must allow the gradual 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, the type system may have to offer support for reasoning about list segments.

We now show how to write and type-check a tail-recursive version of append in Mezzo. The code appears in Fig. 7. It is written in destination-passing style [Larus 1989], and has constant space overhead. Roughly speaking, the list xs is traversed and copied on the fly. When the end of xs is reached, the last cell of the copy is made to point to ys. We emphasize that, even though mutation is used internally, the goal is to concatenate two immutable lists so as to obtain an immutable list.

A detailed look at the code. The append function (line 20) is where concatenation begins. If xs is empty, then the concatenation of xs and ys is ys (line 23). Otherwise (line 25), append allocates an unfinished, mutable cell dst. This cell contains the first element of the final list, namely xs.head. It is not a valid list cell: its tail field contains the unit value (). It is now up to appendAux to finish the work by constructing the concatenation of xs.tail and ys and by writing the address of that list into dst.tail. Once appendAux returns, dst has become a well-formed list (this is indicated by the postcondition dst @ list a on line 8) and is returned by append.

The function appendAux expects an unfinished, mutable cell dst and two lists xs and ys. Its purpose is to write the concatenation of xs and ys into dst.tail, at which point dst can be considered a well-formed list.

If xs is Nil (line 10), the address ys is written to the field dst.tail (line 11). Then, dst, a mutable block whose tag is Cell, is “frozen” by a tag update instruction (line 12) and becomes an immutable block, whose tag is Cons. As in §2.1, this instruction has no runtime effect, because these tags have the same runtime representation.

If xs is a Cons cell (line 13), we allocate a new destination cell dst’ (line 14), let dst.tail point to it (line 15), freeze dst (line 16), and repeat the process via a tail-recursive call (line 17).

Reasoning without segments. Operations on (mutable or immutable) lists with constant space overhead are traditionally implemented in an iterative manner, using a while loop. For instance, Berdine et al.’s formulation of mutable list melding [2005], which is proved correct in separation logic, has a complex loop invariant, involving two list segments, and requires an inductive proof that the concatenation of two list segments is a list segment. In contrast, in our tail-recursive approach, the “loop invariant” is the type of the recursive function appendAux (Fig. 7). This type is quite natural and does not involve list segments.
How do we get away without list segments and without inductive reasoning? The trick is that, even though appendAux is tail-recursive, which means that no code is executed after the call by appendAux to itself, a reasoning step still takes place after the call.

Let us examine lines 14–17 in detail. Upon entering the Cons branch, at the start of line 14, the permission for \( x_s \) is \( x_s @ \text{Cons} \{ \text{head}: a; \text{tail}: \text{list} \ a \} \). As in the earlier version of append (Fig. 6), the type-checker automatically decomposes it into a conjunction. Here, this requires introducing fresh internal names for the head and tail fields, because the programmer did not provide explicit names for these fields as part of the pattern on line 13. For clarity, we use the names head and tail. Thus, at the beginning of line 14, the current permission is:

\[
\begin{align*}
& \text{dst} @ \text{Cell} \{ \text{head}: a; \text{tail}: () \} * \\
& \text{xs} @ \text{Cons} \{ \text{head} = \text{head}; \text{tail} = \text{tail} \} * \\
& \text{head} @ a * \\
& \text{tail} @ \text{list} \ a * \\
& \text{ys} @ \text{list} \ a
\end{align*}
\]

On line 14, we read \( x_s.\text{head} \). According to the second permission above, this read is permitted, and produces a value whose type is the singleton type \( =\text{head} \). In other words, it must produce the value head. Then, we allocate a new memory block, \( \text{dst}' \). This yields one new permission, which comes in addition to those above:

\[
\begin{align*}
& \text{dst}' @ \text{Cell} \{ \text{head} = \text{head}; \text{tail}: () \}
\end{align*}
\]

Although this does not play a key role here, it is worth noting that these permissions imply that the fields \( x_s.\text{head} \) and \( \text{dst}'.\text{head} \) contain the same value, namely head. Besides, we have one (affine) permission for this value, \( \text{head} @ a \). So, the type-checker “knows” that \( x_s.\text{head} \) and \( \text{dst}'.\text{head} \) are interchangeable, and that either of them (but not both separately) can be used as a value of type \( a \). Thanks to this precise knowledge, we do not need a “borrowing” convention [Naden et al. 2012] so as to decide which of \( x_s.\text{head} \) or \( \text{dst}'.\text{head} \) has type \( a \). The idea of recording must-alias information (i.e., equations) via structural permissions and singleton types is taken from Alias Types [Smith et al. 2000]. Separation logic [Reynolds 2002] offers analogous expressiveness via points-to assertions and ordinary variables.

The assignment of line 15 and the tag update of line 16 are reflected by updating the structural permission that describes \( \text{dst} \). Thus, at the beginning of line 17, just before the recursive call, the current permission is:

\[
\begin{align*}
& \text{dst} @ \text{Cons} \{ \text{head}: a; \text{tail} = \text{dst}' \} * \\
& \text{xs} @ \text{Cons} \{ \text{head} = \text{head}; \text{tail} = \text{tail} \} * \\
& \text{head} @ a * \\
& \text{tail} @ \text{list} \ a * \\
& \text{ys} @ \text{list} \ a * \\
& \text{dst}' @ \text{Cell} \{ \text{head} = \text{head}; \text{tail}: () \}
\end{align*}
\]

The call consumes the last four permissions and produces a new permission for \( \text{dst}' \). Immediately, after the call, the current permission is thus:

\[
\begin{align*}
& \text{dst} @ \text{Cons} \{ \text{head}: a; \text{tail} = \text{dst}' \} * \\
& \text{xs} @ \text{Cons} \{ \text{head} = \text{head}; \text{tail} = \text{tail} \} * \\
& \text{dst}' @ \text{list} \ a
\end{align*}
\]

We have reached the end of the code. However, the type-checker still has to verify that the postcondition of appendAux is satisfied. By combining the first and last permissions above, it obtains \( \text{dst} @ \text{Cons} \{ \text{head}: a; \text{tail}: \text{list} \ a \} \). Then, it folds this
A minimal implementation of stacks, with a higher-order iteration function

permission into $\text{dst} @ \text{list} \ a$, thus proving that the postcondition is indeed satisfied: 
$\text{dst}$ is now a valid list.

The fact that the structural permission $\text{dst} @ \text{Cons} \ { \ldots }$ was framed out during the recursive call, as well as the folding steps that take place after the call, are the key technical mechanisms that obviate the need for list segments and inductive reasoning. In short, the code is tail-recursive, but the manner in which one reasons about it is recursive.

Minamide [1998] proposes a notion of “data structure with a hole”, or in other words, a segment, and applies it to the problem of concatenating immutable lists. Walker and Morrisett [2000] offer a tail-recursive version of mutable list concatenation in a low-level typed intermediate language, as opposed to a surface language. The manner in which they 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. [2011] study a slightly different approach, also in the setting of a typed intermediate language, where separating implication offers a way of defining list segments.

Our approach could be adapted to an iterative setting by adopting a new proof rule for while loops. This is noted independently by Charguéraud [Charguéraud 2010, §3.3.2] and by Tuerk [2010].

2.3. A higher-order function

We briefly present a minimal implementation of stacks on top of linked lists. This allows us to show an example of a higher-order function, which is later re-used in the example of graphs and depth-first search (§2.5).

The implementation appears in Fig. 8. A stack is defined as a mutable reference to a list of elements. The function new creates a new stack; the function push inserts
a list of elements into an existing stack. The latter relies on the list concatenation function (§2.2). The higher-order function \texttt{work} abstracts a typical pattern of use of a stack as a work list: as long as the stack is non-empty, extract one element out of it, process this element (possibly causing new elements to be pushed onto the stack), and repeat. This is a loop, expressed as a tail-recursive function. The parameter \texttt{s} is the stack; the parameter \texttt{f} is a user-provided function that is in charge of processing one element. This function has access to the permission \texttt{s @ stack a}, which means that it is allowed to update the stack. The code is polymorphic in the type \texttt{a} of the elements. It is also polymorphic in a permission \texttt{p} that is threaded through the whole computation: if \texttt{f} requires and preserves \texttt{p}, then \texttt{work} also requires and preserves \texttt{p}. One can think of the conjunction \texttt{s @ stack a * p} as the loop invariant. The pattern of abstracting over a permission \texttt{p} is typical of higher-order functions.

2.4. Borrowing elements from containers

In Mezzo, a container naturally “owns” its elements, if they have affine type. A list is a typical example of this phenomenon. Indeed, in order to construct a permission of the form \texttt{xs @ list t}, one must provide a permission \texttt{x @ t} for every element \texttt{x} of the list \texttt{xs}.

If the type \texttt{t} is affine, then one must give up the permission \texttt{x @ t} when one inserts \texttt{x} into the list. Conversely, when one extracts an element \texttt{x} out of the list, one recovers the permission \texttt{x @ t}. Other container data structures, such as trees and hash tables, work in the same way.

If the type \texttt{t} is duplicable, then the permission \texttt{x @ t} does not have an ownership reading. One can duplicate this permission, give away one copy to the container when \texttt{x} is inserted into it, and keep one copy so that \texttt{x} can still be used independently of the container.

An ownership problem. The fact that a container “owns” its elements seems fairly natural as long as one is solely interested in inserting and extracting elements. Yet, a difficulty arises if one wishes to borrow an element, that is, to obtain access to it and examine it, without taking it out of the container.

We illustrate this problem with the function \texttt{find}, which scans a list \texttt{xs} and returns the first element \texttt{x} (if there is one) that satisfies a user-provided predicate \texttt{pred}.

Transliterating the type of this function from ML to Mezzo, one might hope that this function admits the following type:

\begin{verbatim}
val find: [a] (xs: list a, pred: a -> bool) -> option a
\end{verbatim}

However, in Mezzo, \texttt{find} cannot have this type. There is an ownership problem: if a suitable element \texttt{x} is found and returned, then this element becomes reachable in two ways, namely through the list \texttt{xs} and through the value returned by \texttt{find}. Thus, somewhere in the code, the permission \texttt{x @ a} must be duplicated. In the absence of any assumption about the type \texttt{a}, this is not permitted.

One could assign \texttt{find} the following type, where the type parameter \texttt{a} is required to be duplicable:

\begin{verbatim}
val find: [a] (xs: list a, pred: a -> bool | duplicable a) -> option a
\end{verbatim}

Naturally, this does not solve the problem. This means that \texttt{find} is supported only in the easy case where the elements are shareable. This is an important special case:
we explain later on (§2.5) that, provided one is willing to perform dynamic ownership
tests, one can always arrange to be in this special case. Nevertheless, it is desirable to
offer a solution to the borrowing problem. In the following, we give an overview of two
potential solutions, each of which has shortcomings.

A solution in continuation-passing style. A simple approach is to give up control.
Instead of asking find to return the desired element, we provide find with a func-
tion that describes what we want to do with this element. The signature of find thus
becomes:

\[
\text{val find: } [a] \langle
\begin{array}{l}
xs: \text{list } a,
pred: a \rightarrow \text{bool},
f: (x: a) \rightarrow ()
\end{array}\rangle \rightarrow ()
\]

Recall that, in Mezzo, a function argument that is \textit{not} annotated with the keyword
\texttt{consumes} is preserved: that is, the function requires and returns a permission for this
argument. Thus, this version of find preserves \(xs \ @ \text{list } a\). The function \(f\), which the
user supplies, preserves \(x \ @ a\), where \(x\) is some element of the list. That is, \(f\) is allowed
to work with this element, but must eventually relinquish the permission to use this
element. Note that \(f\) does not have access to the list: it does not receive the permission
\(xs \ @ \text{list } a\). If it did, the ownership problem would arise again!

It is indeed possible to write a version of find that admits the above type. One soon
finds out, however, that this type is not expressive enough, as it does not provide any
permission to \(f\) beside \(x \ @ a\). This means that \(f\) cannot perform any side effect, except
possibly on \(x\). In order to relax this restriction, one must parameterize find over a
permission \(s\), which is transmitted to (and preserved by) \(f\). This is a typical idiom for
higher-order functions in Mezzo.

\[
\text{val find: } [a, s: \text{perm}] \langle
\begin{array}{l}
xs: \text{list } a,
pred: a \rightarrow \text{bool},
f: (x: a | s) \rightarrow ()
\end{array}\rangle \rightarrow ()
\]

This approach works, but is awkward for a variety of reasons. First, working in
continuation-passing style is unnatural and rigid: elements must be borrowed from
the container and returned to the container in a well-parenthesized manner. Second,
it is verbose, especially in light of the fact that, in Mezzo, anonymous functions must
be explicitly type-annotated. In fact, this version of find is just a restricted form of the
general-purpose higher-order function for iterating on a list, iter. So, one may just as
well provide iter and omit find.

A solution in direct style. The root of the problem lies in the fact that the permissions
\(xs \ @ \text{list } a\) and \(x \ @ a\) cannot coexist. Thus, the function find, if written in a standard
style, must consume \(xs \ @ \text{list } a\) and produce \(x \ @ a\). Of course, there must be a way
for the user to signal that she is done working with \(x\), at which point she would like to
relinquish \(x \ @ a\) and recover \(xs \ @ \text{list } a\).

Fig. 9 shows a version of find that follows this idea. The function find requires the
permission \(xs \ @ \text{list } a\), which it consumes (line 9). If no suitable element exists in the
list, then it returns a unit value, together with the permission \(xs \ @ \text{list } a\) (line 11).
If one exists, then it returns a \textit{focused element} (line 12). This alternative is expressed
alias wand (pre: perm) (post: perm) =
{ammo: perm} ( |
  | consumes (pre * ammo)) -> (| post)
| ammo

alias focused a (post: perm) =
(x: a, w: wand (x @ a) post)

val rec find [a] (consumes xs: list a, pred: a -> bool)
: either |
  | xs @ list a)
  | (focused a (xs @ list a))
  = match xs with |
  | Nil ->
    Left { contents = () } |
  | Cons { head; tail } ->
    if pred head then begin
      let w (| consumes (head @ a * tail @ list a))
       : (| xs @ list a) = () in
      Right { contents = (head, w) }
    end
  else
    match find (tail, pred) with |
    | Left ->
      Left { contents = () } |
    | Right { contents = (x, w) } ->
      let flex guess: perm in
      let w' (| consumes (x @ a * head @ a * guess))
       : (| xs @ list a) = w() in
      Right { contents = (x, w') }
    end
end

Fig. 9. Borrowing an element from a container in direct style

via the algebraic data type either (line 10), which is defined in the standard library as follows:

data either a b =
  | Left { contents: a }
  | Right { contents: b }

The notion of a focused element appears in our unpublished work on iterators, which pose a similar problem [Guéneau et al. 2013]. A focused element (lines 6–7) is a pair of an element x, which has type a, and a function w, a “magic wand” (or “separating implication” [Reynolds 2002]) that takes away x @ a and produces xs @ list a instead. The idea is, when the user is provided with a focused element (x, w), she can work with x as long as she likes; once she is done, she invokes the function w. This function in principle does nothing at runtime: by calling it, the user tells the type-checker that she is done with x and would now like to recover the permission to use the list xs.

Mezzo does not currently have magic wand as a primitive notion. Instead, we define a magic wand (lines 1–4) as a (runtime) function of no argument and no result, which consumes a permission pre and produces a permission post. Such a function typically
has some internal state, which, conjoined with \( \text{pre} \), gives rise to \( \text{post} \). In our definition, this internal state is represented by the existentially quantified permission \( \text{ammo} \). Within the existential quantifier, written \( \{ \text{ammo} : \text{perm} \} \), is a package of a function that consumes \( \text{pre} \) * \( \text{ammo} \) and of one copy of \( \text{ammo} \). Because \( \text{ammo} \) is affine, a magic wand can be used at most once: it is a one-shot function. The name “\( \text{ammo} \)” suggests the image of a gun that needs a special type of ammunition and is supplied with just one cartridge of that type.

Equipped with these (fairly elaborate, but re-usable) definitions, we may explain the definition of \textit{find}.

At line 15, we have reached the end of the list. We return a \textit{left} injection, applied to a unit value. The ownership of the (empty) list is returned to the caller.

At line 20, the element head is the one we are looking for. We return a right injection, applied to a pair of head and a suitable wand \( w \). This pair forms a focused element. What is a suitable wand in this case? It should have type \( \text{wand} (\text{head} @ \text{a}) (\text{xs} @ \text{list a}) \). The function \( w \) defined at lines 18–19 ostensibly has type \( (| \text{consumes} (\text{head} @ \text{a} * \text{tail} @ \text{list a})) \rightarrow (| \text{xs} @ \text{list a}) \). The type-checker is able to verify that the latter is a subtype of the former; this involves an existential quantifier introduction, taking \( \text{tail @ list a} \) as the witness for \( \text{ammo} \). The type-checker must also verify that the definition of \( w \) matches its declared type. This is indeed the case because \( w \) has access not only to \( \text{head @ a} * \text{tail @ list a} \), but also to the duplicable permission \( \text{xs @ Cons} \{ \text{head = head; tail = tail} \} \). (In Mezzo, a function has access to every \textit{duplicable} permission that exists at its definition site.) By combining these three permissions, one obtains \( \text{xs @ list a} \), as desired.

At line 25, the desired element has not been found further down: the recursive call to \textit{find} returns \textit{Left}. Even though the code is terse, the reasoning is non-trivial. As we are in the \textit{Left} branch, we have \( \text{tail @ list a} \). Furthermore, we still possess \( \text{head @ a} \) and \( \text{xs @ Cons} \{ \text{head = head; tail = tail} \} \), which were framed out during the call. The type-checker recombines these permissions and verifies that we have \( \text{xs @ list a} \), as demanded in this case by the postcondition of \textit{find}.

At line 26 is the last and most intricate case. The desired element \( x \) has been found further down. The recursive call returns the value \( x \), the permission \( x @ a \), and a wand \( w \) that we are supposed to use when we are done with \( x \). This wand has type \( \text{wand} (x @ a) (\text{tail @ list a}) \). Using it, we build a new wand \( w' \), which has type \( \text{wand} (x @ a) (\text{xs @ list a}) \). Thus, the pair \( (x, w') \) has type focused \( a (\text{xs @ list a}) \), as desired. We return it, wrapped in a right injection.

Limits of this approach. The strength of this approach is that it allows the user to work in direct style, as opposed to continuation-passing style. The fact that Mezzo’s type discipline is powerful enough to express the concepts of one-shot function, magic wand, focused element, and to explain what is going on in the \textit{find} function, is good.

\footnote{For the benefit of the reader who would like to understand this code fragment in detail, let us say a little more. On line 26, the type-checker automatically expands the type of \( w \), which is an abbreviation for an existential type, and unpacks this existential type. Thus, it views \( w \) as a function of type \( (| \text{consumes} (x @ a * \text{ammo})) \rightarrow (| \text{tail @ list a}) \), where \( \text{ammo} \) is a fresh abstract permission; and it considers that one copy of \( \text{ammo} \) is available. At this point, \( \text{ammo} \) is anonymous: there is no way for the programmer to refer to it. Yet, this permission must be listed in the header of the function \( w' \); because \( w' \) calls \( w \), it needs the permission \( \text{ammo} \). We solve this problem via the \textit{let flex} construct on line 27. This construct introduces a flexible variable guess. When the type-checker examines the call \( w() \) on line 29, it is able to guess that guess must be unified with \( \text{ammo} \), as there is otherwise no way for this call to be well-typed. Finally, the type-checker verifies that the definition of \( w' \) matches its explicitly-declared type (the reasoning is analogous to that of lines 18–19) and performs an existential type introduction, automatically picking \( \text{head @ a} * \text{guess} \) as the witness, in order to prove that \( w' \) has type \( \text{wand} (x @ a) (\text{xs @ list a}) \), as desired.}
Nevertheless, we are well aware that this solution is not fully satisfactory, and illustrates some of the limitations of Mezzo, as it stands today.

For one thing, the code is verbose, and requires non-trivial type annotations, in spite of the fact that the type-checker already performs quite a lot of work for us, including automatic elimination and introduction of existential quantifiers. The effort involved in writing this code is well beyond what most programmers would expect to spend.

A related issue is that the definition of find contains two eta-expansions, which, ideally, should be unnecessary. On lines 28–29, the function w’ is defined by

\[
\text{let } w'() = w() \text{ in } \ldots
\]

One would like to identify w’ with w. Then, one would see that, on line 23, the case analysis construct is of the form

\[
\text{match } \ldots \text{ with Left\{\ldots\} } \rightarrow \text{ Left\{\ldots\} } \mid \text{ Right\{\ldots\} } \rightarrow \text{ Right\{\ldots\}, and one would wish to replace this entire match construct with just find(tail, pred). The code would be much more transparent, and it would be clear that the recursive call is a tail call. At present, the Mezzo compiler can in principle perform these optimizations behind the scene, but that is not quite satisfactory.}

Another criticism is that we encode a magic wand as a runtime function, even though this function has no runtime effect. Ideally, there should be a way of declaring that a function is “ghost”. The system would check that this function has no runtime effect (including non-termination). This would eliminate the need for allocating a pair (x, w) at runtime.

Limits of both approaches. In either approach, when one borrows an element x from a list xs, one gains the permission x @ a, but loses xs @ list a. This means that at most one element at a time can be borrowed from a container.

In a way, this restriction makes sense. One definitely cannot hope to borrow a single element x twice, as that would entail duplicating the affine permission x @ a. Thus, in order to borrow two elements x and y from a single container, one must somehow prove that x and y are distinct. Such a proof is likely to be beyond the scope of a type system; it may well require a full-fledged program logic.

At this point, the picture may seem quite bleak. One thing to keep in mind, though, is that the whole problem vanishes when the type a is duplicable. This brings us naturally to the next section. We propose a mechanism, adoption and abandon, which can be viewed as a way of converting between an affine type a and a universal duplicable type, dynamic. One can then use a container whose elements have type dynamic, and look up multiple elements in this container, without restriction. Naturally, the conversion from type dynamic back to type a involves a runtime check, so that attempting to borrow a single element twice causes a runtime failure. The proof obligation x ≠ y is deferred from compile time to runtime.

2.5. Breaking out: arbitrary aliasing of mutable data structures

The type-theoretic discipline that we have presented up to this point allows constructing a composite permission out of several permissions and (conversely) breaking a composite permission into several components. For instance, a permission for a list is interconvertible with a conjunction of separate permissions for the head cell and for the tail (§2.2). More generally, a permission for a tree is interconvertible with a conjunction of separate permissions for the root record and for the sub-trees. Thus, every tree-shaped data structure can be described in Mezzo by an algebraic data type.

There are two main limitations to the expressive power of this discipline.

First, because we adopt an inductive (as opposed to co-inductive) interpretation of algebraic data types, a permission cannot be a component of itself. In other words, it cannot be used in its own construction. This holds of both duplicable and affine permissions. Thus, every algebraic data type describes a family of acyclic data structures.
The permission `xs @ list int`, for instance, means that `xs` is a finite list of integers. (In this sense, Mezzo differs from OCaml, which allows constructing a cyclic list.) This choice is intentional: we believe that it is most often desirable to ensure the absence of cycles in an algebraic data structure.

Second, an affine permission cannot serve as a component in the construction of two separate composite permissions. Because every mutable memory block (and, more generally, every data structure that contains such a block) is described by an affine permission, this means that mutable data structures cannot be shared. Put in another way, this discipline effectively imposes an ownership hierarchy on the mutable part of the heap.

When one wishes to describe a data structure that involves a cycle in the heap or the sharing of a mutable sub-structure, one must work around the restrictions described above. This requires extra machinery.

**Illustration.** In order to illustrate the problem, let us define a naïve type of graphs and attempt to construct the simplest possible cyclic graph, where a single node points to itself.

The definition of the type node is straightforward (Fig. 10, lines 1–6). Every node stores a value of type `a`, where the type variable `a` is a parameter of the definition; a Boolean flag, which allows this node to be marked during a graph traversal; and a list of successor nodes. The type node is declared mutable: it is easy to think of applications where all three fields must be writable.

Next (lines 8–16), we allocate one node `n`, set its neighbors field to a singleton list of just `n` itself, and claim (via the type annotation on line 8) that, at the end of this construction, `n` has type node `int`. This code is ill-typed, and is rejected by the type-checker. Perhaps surprisingly, the type error does not lie at line 15, where a cycle in the heap is constructed. Indeed, at the end of this line, the heap is described by the following permission:

```ocaml
n @ Node {
  value : int;
  visited : bool;
  neighbors = ns;
} *
```

Fig. 10. A failed attempt to construct a cyclic graph
data mutable node a =
  Node {
    content : a;
    visited : bool;
    neighbors: list dynamic;
  }

data mutable graph a =
  Graph {
    roots : list dynamic;
  } adopts node a

val g : graph int =
  let n = Node {
    content = 10;
    visited = false;
    neighbors = ()
  } in
  let ns = Cons { head = n; tail = Nil } in
  n.neighbors <- ns;
  assert n @ node int * ns @ list dynamic;
  let g : graph int = Graph { roots = ns } in
  give n to g;
  g

val dfs [a] (g: graph a, f: a -> () : ()) : () =
  let s = stack::new g.roots in
  stack::work (s, fun (n: dynamic
    | g @ graph a * s @ stack dynamic) : () =
    take n from g;
    if not n.visited then begin
      n.visited <- true;
      f n.content;
      stack::push (n.neighbors, s)
    end;
    give n to g
  )

ns @ Cons { head = n; tail: Nil }

This illustrates the fact that a cycle of statically known length can be described in terms of structural permissions and singleton types. The type error lies on line 16, where (due to the type annotation on line 8) the type-checker must verify that the above permission entails n @ node int. This entailment is invalid because it violates the second limitation that was discussed earlier: since n @ Node { ... } is affine, it cannot be used to separately justify that n is a node and ns is a list of nodes. Furthermore, even if n @ Node { ... } was duplicable, this entailment would still be invalid, as it also violates the first limitation that was discussed above: since the algebraic data type node is interpreted inductively, a node cannot participate in its own construction.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
To sum up, the type node at lines 1–6 is not a type of possibly cyclic graphs, as one might have naïvely imagined. It is in fact a type of trees, where each tree is composed of a root node and a list of disjoint sub-trees.

A solution. The problem with this naïve approach stems from the fact that types have an ownership reading. Saying that neighbors is a list of nodes amounts to claiming that every node owns its successors, which does not make sense, because ownership must be a hierarchy.

In order to solve this problem, we must allow a node to point to a successor without implying that there is an ownership relation between them. “Who” then should own the nodes? A natural answer is, the set of all nodes should be owned as a whole by a single distinguished object, say, the “graph” object.

Fig. 11 presents a corrected definition of graphs, and shows how to build the cyclic graph of one node. It also contains code for an iterative version of depth-first search, using an explicit stack. Let us explain this example step by step.

The type dynamic. The only change in the definition of node a is that the neighbors field now has type list dynamic (line 5).

The meaning of n @ dynamic is that n is a valid address in the heap, i.e., it is the address of a memory block. When one allocates a new memory block, say via let n = Node { ... } in ..., one obtains not only a structural permission n @ Node { ... }, but also n @ dynamic. Although the former is affine (because Node refers to a mutable algebraic data type), the latter is duplicable. Intuitively, it is sound for the type dynamic to be considered duplicable because the knowledge that n is a valid address can never be invalidated, hence can be freely shared. However, the permission n @ dynamic does not allow reading or writing at this address. In fact, it does not even describe the type of the memory block that is found there—and it cannot: this block is owned by “someone else” and its type could change with time.

Because it is duplicable, the type dynamic does not have an ownership reading. The fact that neighbors has type list dynamic does not imply that a node owns its successors; it means only that neighbors is a list of heap addresses.

Constructing a cyclic graph. As an example, we construct a node that points to itself (lines 14–21). The construction is the same as in Fig. 10. This time, it is well-typed, though. Because we have n @ dynamic, we can establish ns @ list dynamic, and, therefore, n @ node int. Furthermore, since ns @ list dynamic is duplicable, it is not consumed in the process. The (redundant) static assertion on line 21 shows that the desired permissions for n and ns co-exist.

The type graph a (lines 8–11) defines the structure of a “graph” object. This object contains a list of so-called root nodes. Like neighbors, this list has type list dynamic. Furthermore, the adopts clause on line 11 declares that an object of type graph a adopts a number of objects of type node a. This is a way of saying that the graph “owns” its nodes. Thus, an object of type graph int is an adopter, whose adoptees are objects of type node int. The set of its adoptees changes with time, as there are two instructions, give and take, for establishing or revoking an adoptee-adopter relationship.

The give and take instructions. The runtime effect of the adoption instruction give n to g (line 23) is that the node n becomes a new adoptee of the graph g. At the beginning of this line, the permissions n @ node int and g @ graph int are available. Together, they justify the instruction give n to g. (The type-checker verifies that the type of g has an adopts clause and that the type of n is consistent with this clause.) After the give instruction, at the end of line 23, the permission n @ node int has been consumed, while g @ graph int remains. A transfer of ownership has taken place: whereas the node n was “owned by this thread”, so to speak, it is now “owned by g".
permission \( g @ \text{graph int} \) should be interpreted intuitively as a proof of ownership of the object \( g \) (which has type \text{graph int}) and of its adoptees (each of which has type \text{node int}). It can be thought of as a conjunction of a permission for just the memory block \( g \) and a permission for the group of \( g \)'s adoptees; in fact, in our formalization (§7), these permissions are explicitly distinguished.

Although \( g @ \text{graph int} \) implies the ownership of all of the adoptees of \( g \), it does not indicate who these adoptees are: the type system does not statically keep track of the relation between adopters and adoptees. After the \text{give} instruction at line 23, for instance, the system does not know that \( n \) is adopted by \( g \). If one wishes to assert that this is indeed the case, one can use the \text{abandon} instruction, \text{take} \( n \) \text{from} \( g \). The runtime effect of this instruction is to check that \( n \) is indeed an adoptee of \( g \) (if that is not the case, the instruction fails) and to revoke this fact. After the instruction, the node \( n \) is no longer an adoptee of \( g \); it is unadopted again. From the type-checker's point of view, the instruction \text{take} \( n \) \text{from} \( g \) requires the permissions \( n @ \text{dynamic} \), which proves that \( n \) is the address of a valid block, and \( g @ \text{graph int} \), which proves that \( g \) is an adopter and indicates that its adoptees have type \text{node int}. It preserves these permissions and (if successful) produces \( n @ \text{node int} \). This is a transfer of ownership in the reverse direction: the ownership of \( n \) is taken away from \( g \) and transferred back to "this thread".

Conceptual model. The adopter owns its adoptees. Conceptually, one can think of the adopter as an object which maintains a list of the elements that it owns, inserting new elements as \text{give} operations take place, and taking them out as \text{take} operations are performed. More precisely, if the adopter "adopts \( t \)" , then one can think of it as owning a "list \( t \)" of its adoptees.

This conceptual view of adoption and abandon can be implemented in Mezzo as a library. In this case, an adopter \( y \) would contain a reference to a list of adoptees. Calling \text{give} \( (x,y) \) would consume a permission \( x @ t \) and store \( x \) in the list of adoptees, leaving the caller with \( x @ \text{dynamic} \). The permission \( x @ \text{dynamic} \) would be implemented as \( x @ (=x) \), a duplicable, always-available permission. Calling \text{take} \( (x,y) \) would perform an address comparison between \( x \) and each element in the list of adoptees. Upon finding \( x \), the element \( x \) would be removed from the list of adoptees, and the permission \( x @ t \) would be returned to the caller.

We propose, however, an alternative representation which performs the \text{give} and \text{take} operations in constant time.

Runtime model. We maintain a pointer from every adoptee to its adopter. Within every object, there is a hidden adopter field, which contains a pointer to the object's current adopter, if it has one, and \text{null} otherwise. This information is updated when an object is adopted or abandoned. In terms of space, the cost of this design decision is one field per object\(^3\).

The runtime effect of the instruction \text{give} \( n \) \text{to} \( g \) is to write the address \( g \) to the field \( n.\text{adopter} \). The static discipline guarantees that this field exists and that its value, prior to adoption, is \text{null}.

The runtime effect of the instruction \text{take} \( n \) \text{from} \( g \) is to check that the field \( n.\text{adopter} \) contains the address \( g \) and to write \text{null} into this field. If this check fails,

---

\(^3\) It would be possible to lessen this cost by letting the programmer decide, for each data type, whether the members of this type should be adoptable (hence, should contain an adopter field) or not. In particular, one could note that there should never be a need for an immutable object to be adoptable. One should then restrict the tag update instruction so as to forbid going from an adoptable data type to a non-adoptable one, or vice-versa. For the moment, for the sake of simplicity, we consider only the uniform model where every object has an adopter field.
the execution of the program is aborted. We also offer an expression form, \( g \text{ adopts } n \), which tests whether \( n \text{.adopter} \) is \( g \) and produces a Boolean result. It is not described in this paper.

**Illustration.** We illustrate the use of adoption and abandon with the example of depth-first search (Fig. 11, lines 26–37). The frontier (i.e., the set of nodes that must be examined next) is represented as a stack; we rely on the stack module of Fig. 8. The stack \( s \) has type \( \text{stack dynamic} \). We know (but the type-checker does not) that the elements of the stack are nodes, and are adoptees of \( g \).

The function \( \text{dfs} \) initializes the stack (line 27) and enters a loop, encoded as a call to the higher-order function \( \text{stack::work} \). At each iteration, an element \( n \) is taken out of the stack; it has type \( \text{dynamic} \) (line 28). Thus, the type-checker does not know a priori that \( n \) is a node. The \( \text{take} \) instruction (line 30) recovers this information. It is justified by the permissions \( n @ \text{dynamic} \) and \( g @ \text{graph int} \) and (if successful) produces \( n @ \text{node int} \). This proves that \( n \) is indeed a node, which we own, and justifies the read and write accesses to this node that appear at lines 31–34. Once we are done with \( n \), we return it to the graph via a \( \text{give} \) instruction (line 36).

There are various mistakes that the programmer could make in this code and that the type-checker would not catch. For instance, forgetting the final \( \text{give} \) would lead to a runtime failure at a later \( \text{take} \) instruction, typically on line 30. In order to diminish the likelihood of this particular mistake, we propose \( \text{taking } n \text{ from } g \text{ begin ... end} \) as syntactic sugar for a well-parenthesized use of \( \text{take} \) and \( \text{give} \).

**Discussion.** Because adoption and abandon are based on a runtime test, they are simple and flexible. If one wished to avoid this runtime test, one would probably end up turning it into a static proof obligation. The proof, however, may be far from trivial, in which case the programmer would have to explicitly state subtle logical properties of the code, and the system would have to offer sufficient logical power for these statements to be expressible. The dynamic discipline of adoption and abandon avoids this difficulty, and meshes well with the static discipline of permissions. We believe that we have a clear story for the user: “when you need to share mutable data structures, use adoption and abandon”.

Adoption and abandon is a flexible mechanism, but also a dangerous one. Because abandon involves a dynamic check, it can cause the program to fail at runtime. In principle, if the programmer knows what she is doing, this should never occur. There is some danger, but, one may argue, that is the price to pay for a simpler static discipline. After all, the danger is effectively less than in ML or Java, where a programming error that creates an undesired alias is undetected at compile time, and can lead to incorrect runtime behavior or security flaws [Tschantz and Ernst 2005].

3. TRANSLATING SURFACE MEZZO DOWN TO CORE MEZZO

The examples presented in the previous section (§2) are valid Mezzo code, expressed in the surface syntax. However, the formal definition of the type and permission discipline (§4–§7) is expressed in terms of a simpler core syntax. The type-checker translates surface syntax down to core syntax, and performs the bulk of the type-checking work at that level.

In this section, we give an informal presentation of this translation, so as to bridge the gap between the examples of the previous section (§2) and the formal definitions that follow (§4–§7). This translation and its properties have not been machine-checked; they are outside of the scope of our Coq formalization.

As far as types and permissions are concerned, there are two differences between surface syntax and core syntax. One difference is that the surface syntax offers a name introduction construct \( x : t \), together with a set of rules that dictate the scope
of the name \( x \). This construct does not exist in the core syntax, where we only have more traditionnal quantifiers. The second difference is that the surface syntax adopts the convention that functions by default do not consume their argument, and offers a \texttt{consumes} keyword to indicate that (part of) the argument is in fact consumed. In contrast, the core syntax does not have a \texttt{consumes} keyword; it adopts the convention that functions do consume their argument, and repeat the non-consumed parts of their argument in their return type.

The surface and core languages also differ in the syntax of terms. We do not describe these differences, which consist mainly in syntactic sugar for function definitions.

This section is structured as follows. First, we illustrate the translation of types from the surface syntax to the core syntax with a few examples (§3.1). These examples have been chosen so as to highlight the main features of the translation, so the reader who feels satisfied with it can safely skip ahead to the beginning of §4. Then, we proceed to give a precise definition of the translation. In §3.2, we define the (combined) surface and core syntaxes of types and permissions. We give a well-kindness judgement, which defines the scope of every name. Finally (§3.3), we define a translation of the surface syntax into the core syntax.

3.1. Examples

Let us consider the following type, which is a simplified version of the type of \texttt{find} (§2.4, Fig. 9).

\[
\begin{align*}
\texttt{[a]} & \ (\texttt{consumes} \ \texttt{xs} : \texttt{list} \ a) \rightarrow \\
& \quad (x : a, \ \texttt{wand} \ (x \ @ \ a) \ (xs \ @ \texttt{list} \ a))
\end{align*}
\]

The name introduction construct \texttt{xs} : \texttt{list} \ a binds the variable \( xs \). The scope of \( xs \) encompasses the domain and codomain of this function type. Consequently, the second occurrence of \( xs \) (in the permission \( xs \ @ \texttt{list} \ a \)) is bound by the name introduction.

The codomain of this function type is a pair \((..., ...)\). The left-hand component of this pair is another name introduction construct \texttt{x} : \texttt{a}. The scope of \( x \) is the whole pair. Consequently, the occurrence of \( x \) in the permission \( x \ @ \texttt{a} \) in the right-hand component of the pair is bound by this second name introduction.

One way of explaining the meaning of these name introduction constructs, and of making it clear where the names \( xs \) and \( x \) are bound, is to translate away the name introductions. In this example, this can be done as follows. This type is equivalent to the previous formulation, and is also valid surface syntax:

\[
\begin{align*}
\texttt{[a]} & \ [xs : \texttt{value}] \\
& \quad (\texttt{consumes} \ (\texttt{=}xs \ | \ xs \ @ \texttt{list} \ a)) \rightarrow \\
& \quad \{x : \texttt{value}\} \\
& \quad \ (\texttt{=}x \ | \ x \ @ \texttt{a}), \ \texttt{wand} \ (x \ @ \texttt{a}) \ (xs \ @ \texttt{list} \ a))
\end{align*}
\]

The name \( xs \) is now universally quantified (at kind \texttt{value}) above the function type. Thus, its scope encompasses the domain and codomain of the function type. The name \( x \) is existentially quantified (also at kind \texttt{value}) above the codomain. Thus, its scope is the codomain.

The name introduction \texttt{xs} : \texttt{list} \ a is now replaced with \( (=xs \ | \ xs \ @ \texttt{list} \ a) \). This is a conjunction of a (singleton) type and a permission. This means that the function \texttt{find} expects a value (which is passed at runtime) and a permission (which exists only at type-checking time). Although placing a singleton type in the domain of a function type may seem absurdly restrictive, the universal quantification on \( xs \) makes the function type general again. By instantiating \( xs \) with \( ys \), one finds that, for any value \( ys \), the call \texttt{find} \( ys \) is well-typed, provided the caller is able to provide the permission \( ys \ @ \texttt{list} \ a \). Similarly, the name introduction \texttt{x} : \texttt{a} is replaced with \( (=x \ | \ x \ @ \texttt{a}) \).
The encoding of dependent products and dependent sums in terms of quantification and singleton types is standard. It is worth noting that our name introduction form is more expressive than traditional dependent products and sums, as it does not have a left-to-right bias. For instance, in the type \((x: t, y: u)\), both of the variables \(x\) and \(y\) are in scope in both of the types \(t\) and \(u\).

It is easy to translate the above type into the core syntax:

\[
\forall (a: \text{type}) \\
\forall (xs: \text{value}) \\
(\neg xs | xs @ \text{list}\ a) \rightarrow \\
\exists (x: \text{value}) \\
((\neg x | x @ a), \text{wand} (x @ a) (xs @ \text{list}\ a))
\]

In this example, because the argument is entirely consumed, the translation is trivial. All we have to do is erase the \text{consumes} keyword. In the core syntax, by convention, the plain arrow \(\rightarrow\) denotes a function that consumes its argument, so this type has the desired meaning.

The translation of \text{consumes} is slightly more complex when only part of the argument is consumed: e.g., when the argument is a pair, one component of which is marked with the keyword \text{consumes}. Consider, for instance, the type of a function that merges two sets, updating its first argument and destroying its second argument:

\[
[a] (\text{set}\ a, \text{consumes} \text{set}\ a) \rightarrow ()
\]

The domain of this function type is a pair, whose second component is marked with the keyword \text{consumes}. We translate this into the core syntax by introducing a name, say \(x\), for this pair, and by writing explicit pre- and postconditions that refer to \(x\):

\[
\forall (a: \text{type}) \\
\forall (x: \text{value}) \\
(\neg x | x @ (\text{set}\ a, \text{set}\ a)) \rightarrow \\
(()) | x @ (\text{set}\ a, \top)
\]

The symbol \(\top\) is an abbreviation for \(\exists (y: \text{value}) = y\), which is a duplicable but non-informative type. Thus, in order for the call \(\text{merge} (s1, s2)\) to be accepted, the caller must provide proof that \(s1\) and \(s2\) are valid sets; but, after the call, only \(s1\) is known to still be a set.

3.2. Well-kindedness

The combined surface and core syntaxes of Mezzo are presented in Fig. 12. The surface syntax has the function type \(T_1 \Rightarrow T_2\), which is exposed to the user under the ASCII form \(T1 \rightarrow T2\). The core syntax has the function type \(T_1 \rightarrow T_2\) instead. The constructs \(x: T\) and \text{consumes} \(T\) appear only in the surface syntax. All of the other constructs are shared between the two levels.

The well-kindedness judgement checks (among other things) that every name is properly bound. Thus, in a slightly indirect way, it defines the scope of every name. In addition to the universal and existential quantifiers, which are perfectly standard, Mezzo offers the name introduction construct \(x: T\), which is non-standard, since \(x\) is in scope not just in the type \(T\), but also “higher up”, so to speak. For instance, in the type \((x_1: T_1, x_2: T_2)\), both \(x_1\) and \(x_2\) are in scope in both \(T_1\) and \(T_2\).

In order to reflect this convention, in the well-kindedness rules, one must at certain well-defined points go down and \text{collect} the names that are introduced by some name introduction form, so as to \text{extend} the environment with assumptions about these names.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

\[ \kappa ::= \text{value} \mid \text{type} \mid \text{perm} \mid \ldots \]  

(types kinds)

\[ T, P ::= \]  

(types and permissions)

\[ x \]  

(variable)

\[ \text{=} \]  

(singleton type)

\[ T \to T \]  

(function type, internal)

\[ (T \mid P) \]  

(type/permission conjunction)

\[ (T, T) \]  

(pair type)

\[ x \neq T \]  

(atomic permission)

\[ \text{empty} \]  

(empty permission)

\[ P \ast P \]  

(permission conjunction)

\[ \text{duplicable} T \]  

(duplicability assertion)

\[ (x : \kappa) \]  

(universal quantification)

\[ \exists(x : \kappa) T \]  

(existential quantification)

\[ T \rightsquigarrow T \]  

(function type, external)

\[ x : T \]  

(name introduction, external)

\[ \text{consumes} T \]  

(consumes annotation, external)

Fig. 12. Types and permissions: combined internal and external syntaxes

\[
\begin{align*}
\text{names}(x : T) & = \{(x, \text{value})\} \uplus \text{names}(T) & \text{(name introduction)} \\
\text{names}(T_1, T_2) & = \text{names}(T_1) \uplus \text{names}(T_2) & \text{(pair type)} \\
\text{names}(T \mid P) & = \text{names}(T) & \text{(type/permission conjunction)} \\
\text{names}(\text{consumes} T) & = \text{names}(T) & \text{(consumes annotation)} \\
\text{names}(T) & = \emptyset & \text{(any other type)}
\end{align*}
\]

Fig. 13. Name collection function

\[
\frac{\text{K-OpenNewScope}}{\Gamma; \text{names}(T) \vdash T : \kappa}
\]

\[
\frac{\Gamma; \#T : \kappa}{\Gamma \vdash T : \kappa}
\]

Fig. 14. Types and permissions: well-kindness (auxiliary judgement)

The auxiliary function \(\text{names}(T)\), which collects the names introduced by the type \(T\), is defined in Fig. 13. In short, it descends into tuples, looking for name introduction forms, and collects the names that they introduce.

The well-kindness judgement \(\Gamma \vdash T : \kappa\) means that under the kind assumptions in \(\Gamma\), the type \(T\) has kind \(\kappa\). Its definition (Fig. 15) relies on an auxiliary judgement, \(\Gamma; \text{names}(T) \vdash T : \kappa\) (Fig. 14). Intuitively, \# is a “beginning-of-scope” mark: it means that, at this point, the names collected by the auxiliary function \(\text{names}\) are in scope.

There are additional restrictions that the well-kindness rules should impose: for instance, the \text{consumes} keyword should appear only in the left-hand side of an arrow, and should not appear under another \text{consumes} keyword. This can be expressed by extending the well-kindness judgement with a Boolean parameter, which indicates whether \text{consumes} is allowed or disallowed. In order to reduce clutter, we omit this aspect.

3.3. Translation

We now define the translation of (well-kind) types and permissions from the surface syntax into the core syntax. For greater clarity, we present it as the composition of two phases. In the first phase, we eliminate the name introduction construct. In the second phase, we transform the external function type into its internal counterpart, and at the same time eliminate the \text{consumes} construct.
Fig. 15. Types and permissions: well-kindness

\[ T_{1}\text{-OPENNEWSCOPE} \]
\[ \Gamma \vdash \forall \Psi (\text{name}(\Psi)) \Gamma' \]

Fig. 16. Types and permissions: first translation phase (auxiliary judgement)

\[ T_{1}\text{-VAR} \]
\[ x \vdash x \]
\[ =x \vdash =x \]

\[ T_{1}\text{-SING} \]
\[ P \vdash P' \]
\[ (T \mid P) \vdash (T \mid P') \]

\[ T_{1}\text{-BAR} \]
\[ P \vdash P' \]
\[ T' \vdash T'' \]
\[ (T \mid P) \vdash (T' \mid P') \]

\[ T_{1}\text{-PAIR} \]
\[ T_1 \vdash T_2 \]
\[ T_1' \vdash T_2' \]
\[ (T_1, T_2) \vdash (T_1', T_2') \]

\[ T_{1}\text{-CONJUNCTION} \]
\[ P_1 \vdash P_1' \]
\[ P_2 \vdash P_2' \]
\[ P_1 \cdot P_2 \vdash P_1' \cdot P_2' \]

\[ T_{1}\text{-DUPLICABLE} \]
\[ \#T \vdash \#T' \]
\[ \text{duplicable } T \vdash \text{duplicable } T' \]

\[ T_{1}\text{-CONSUMES} \]
\[ \Gamma \vdash T : \kappa \]
\[ \Gamma \vdash \kappa \in \{\text{name}(\Psi)\} \]
\[ \kappa \in \{\text{name}(\Psi)\} \]

Fig. 17. Types and permissions: first translation phase

\[ T_{2}\text{-EXTERNALARROW} \]
\[ T_1 \vdash T_2 \]
\[ T_1' \vdash T_2' \]
\[ (T_1, T_2) \vdash (T_1', T_2') \]

\[ T_{2}\text{-INTERNALARROW} \]
\[ T_1' \vdash T_2' \]
\[ T_1 \vdash T_2 \]
\[ T_1' \vdash T_2' \]

\[ T_{2}\text{-CONSUMES} \]
\[ \Gamma \vdash (x : T) : \kappa \]
\[ \Gamma \vdash \kappa \in \{\text{name}(\Psi)\} \]
\[ \kappa \in \{\text{name}(\Psi)\} \]

Fig. 18. Types and permissions: second translation phase (only one rule shown)
The Design and Formalization of Mezzo, a Permission-Based Programming Language

Phase 1. The first phase is described by the translation judgement \( T \triangleright T' \), whose definition (Fig. 17) relies on the auxiliary judgement \(#T \triangleright T'\) (Fig. 16).

The main rules of interest are \( T1-OPENNEWSCOPE \), which introduces explicit existential quantifiers for the names whose scope begins at this point; \( T1-EXTERNALARROW \), which introduces explicit universal quantifiers, above the function arrow, for the names introduced by the domain of the function; and \( T1-NAMEMEPRO \), which translates a name introduction form to a conjunction of a singleton type \( \pi x \) and a permission \( x \mathbin{@} T' \). The two occurrences of \( x \) in this conjunction are free: they refer to a quantifier that has necessarily been introduced higher up by \( T1-OPENNEWSCOPE \) or \( T1-EXTERNALARROW \).

Claim 3.1. Well-kindness is preserved by the first translation phase:

\[ \Gamma \vdash T : \kappa \text{ and } T \triangleright T' \implies \Gamma \vdash T' : \kappa. \]

\[ \Gamma \vdash \#T : \kappa \text{ and } \#T \triangleright T' \implies \Gamma \vdash T' : \kappa. \]

Claim 3.2. If \( T \triangleright T' \) or \( \#T \triangleright T' \) holds, then \( T' \) does not contain a name introduction construct.

Phase 2. The second phase is described by the translation judgement \( T \triangleright T' \), whose definition appears in Fig. 18. Only one rule is shown, as the other rules (omitted) simply encode a recursive traversal.

The rule \( T2-EXTERNALARROW \) does several things at once. First, it transforms an external arrow \( \to \) into an internal arrow \( \to \). Second, it introduces a fresh name, \( x \), which refers to the argument of the function; this is imposed by the singleton type \( \pi x \). Finally, in order to express the meaning of the \( \text{consumes} \) keywords that may appear in the type \( T' \), it constructs distinct pre- and postconditions, namely \( x \mathbin{@} T'_{\text{in}} \) and \( x \mathbin{@} T'_{\text{out}} \). These permissions respectively represent the properties of \( x \) that the function requires (prior to the call) and ensures (after the call).

The type \( T'_{\text{in}} \) is defined as \( [T/\text{consumes} T'_{\text{in}}] \). By this informal notation, we mean “a copy of \( T' \), where every subterm of the form consumes \( T \) is replaced with just \( T' \)”, or in other words, “a copy of \( T' \), where every \( \text{consumes} \) keyword is erased”.

The type \( T'_{\text{out}} \) is defined as \( [?/\text{consumes} T'] \). By this informal notation, we mean “a copy of \( T' \), where every subterm of the form consumes \( T \) is replaced with \( \exists(x : \kappa) \ x \), where the kind \( \kappa \) is either type or perma, as appropriate” (this expression \( \exists(x : \kappa) \ x \) is a type or permission that is of the appropriate kind but that does not provide any useful information, in this sense it is similar to \( \text{TyTop} \)).

Thus, the permission \( x \mathbin{@} T'_{\text{in}} \) represents the ownership of the argument, including the components marked with \( \text{consumes} \), whereas the permission \( x \mathbin{@} T'_{\text{out}} \) represents the ownership of the argument, deprived of these components.

Claim 3.3. Well-kindness is preserved by the second translation phase: assuming that \( T \) contains no name introduction forms, \( \Gamma \vdash T : \kappa \) and \( T \triangleright T' \) imply \( \Gamma \vdash T' : \kappa. \)

Claim 3.4. If \( T \triangleright T' \) holds, then \( T' \) contains no external arrow \( \to \) and no \( \text{consumes} \) keyword.

4. KERNEL

Translating types (and terms) in the manner described in the previous section (§3) yields a Core Mezzo program. The remainder of this paper is devoted to:

— defining what it means for a Core Mezzo program to be well-typed;
— proving that a well-typed program cannot go wrong and must be data-race free.

Another important question is: how does one effectively determine whether a program is well-typed? This question is not addressed here: we provide a declarative definition
of well-typedness, not a type-checking algorithm. The type-checking algorithms that we have implemented are described in Protzenko's dissertation [2014].

A (pseudo-)modular approach. Instead of defining Core Mezzo in a monolithic way, we set it up in a modular manner. We begin with a kernel, on top of which sit several relatively independent layers. This approach makes our presentation more gentle and makes the maintenance of the Coq formalization easier.

In order to avoid any confusion, we must emphasize the fact that our Coq code is, strictly speaking, not modular. The kernel and its extensions are not independent artifacts: they cannot be independently type-checked and later brought together. In reality, the Coq code is monolithic: there is a single inductive type of the syntax of Core Mezzo, which includes the kernel and its extensions. Still, the code is “pseudo-modular” in the sense that the presence or absence of one extension in principle has little to no impact on the code of the kernel or of the other extensions. We briefly come back to this issue when we discuss the related work (§9).

Organization. The kernel, described in this section (§4), is a call-by-value λ-calculus, equipped with a construct for dynamic thread creation. The three layers that we describe in this paper are heap-allocated references (§5), locks (§6), and adoption and abandon (§7). Yet more layers would be needed in order to account for all of the features of Mezzo, as implemented today; we describe them briefly in §8.

We now present the core elements of the formalization of Mezzo, that is, the kernel of the proof. First, we axiomatize a notion of machine state, a notion of instrumented machine state (also known as a resource), and a connection between the two (§4.1). We present the syntax (§4.2) and operational semantics (§4.3) of the untyped calculus. We equip the calculus with a type discipline (§4.4, §4.5, §4.6) and prove a type soundness theorem (§4.7).

4.1. Machine states and resources
The kernel calculus does not include any explicit effectful operations. We will however add various kinds of such operations at a later stage. We would like this addition to take the form of an extension: that is, we would like to add new syntactic forms, new reduction rules, new typing rules, new auxiliary lemmas, etc. We do not wish (insofar as possible) to modify existing definitions or statements.

For this purpose, we build into the kernel calculus the notions of machine state and instrumented machine state. We refer to the latter also as a resource. Even though we do not know at this stage what machine states are, they already appear in the operational semantics. Even though we do not know yet what resources are, they already appear in the typing rules; and the type preservation theorem for the kernel calculus essentially means that the type system “keeps correct track of resources”.

Machine states. We write $s$ for a machine state. At this stage, the nature of machine states is unspecified. A machine state should be thought of as a tuple, some of whose components are specified at a later stage in this paper: a reference heap (§5), a lock heap (§6). There could be more: the type of machine states is informally considered open-ended. We assume that there is a distinguished machine state initial in which the execution of a program begins.

Resources. A running program is composed of multiple threads, each of which has partial knowledge of the current machine state and partial rights to alter this state. We account for this by working with a notion of resource, of which one can think as the “view” of a thread [Dinsdale-Young et al. 2013]. We write $R$ for a resource.

At this stage, again, the nature of resources is unspecified. One should think of a resource as a partial, instrumented machine state.
A resource is “partial” because it represents possibly incomplete knowledge about the machine state. A heap fragment in the style of Separation Logic [Reynolds 2002] is an example of a partial resource: about a memory location in its domain, it contains precise information (i.e., it indicates which value is stored there); about a memory location outside its domain, it contains no information at all.

A resource is “instrumented” because it may contain information that does not exist at runtime, but helps express an invariant of the type system. An example is a heap fragment where a memory location is mapped not just to a value, but also to an access right (e.g., read-only versus read-write; or a fraction between 0 and 1). Another example is an ML store typing in the style of Wright and Felleisen [1994], where every memory location is mapped to its type (which is fixed upon allocation). In Core Mezzo, the lock heap (§6) plays a similar role: it maps every lock address to the invariant (a permission) associated with this lock.

Axiomatization of resources. We require resources to form a monotonic separation algebra [Pottier 2013, §10], also known as an MSA, for short. That is, we make the following assumptions:

— A composition operator \( \ast \) allows two resources (i.e., the views of two threads) to be combined. It is total, commutative, and associative.

— A consistency predicate, \( R \text{ ok} \), identifies the well-formed resources. It is preserved by splitting, i.e., \( R \ast R_1 \text{ ok} \) implies \( R_1 \text{ ok} \).

— A total function \( \hat{\cdot} \) maps every resource \( R \) to its core \( \hat{R} \), which represents the duplicable (shareable) information contained in \( R \).

— This element is a unit for \( \ast \), i.e., \( R \ast \hat{R} = R \).

— Two compatible elements have a common core, i.e., \( R_1 \ast R_2 = R \) and \( R \text{ ok} \) imply \( \hat{R_1} = \hat{R} \).

— A duplicable resource is its own core, i.e., \( R \ast R = R \) implies \( R = \hat{R} \).

— Every core is duplicable, i.e., \( \hat{R} \ast \hat{R} = \hat{R} \).

— A relation \( R_1 \triangleright R_2 \), the rely, represents the interference that “other” threads are allowed to inflict on “this” thread, by specifying how a known resource \( R_1 \) can evolve in a resource \( R_2 \). For instance, the allocation of new memory blocks, or of new locks, is typically permitted by this relation.

— This relation is reflexive.

— It preserves consistency, i.e., \( R_1 \text{ ok} \) and \( R_1 \triangleright R_2 \) imply \( R_2 \text{ ok} \).

— It is preserved by core, i.e., \( R_1 \triangleright R_2 \) implies \( \hat{R}_1 \triangleright \hat{R}_2 \).

— Finally, it is compatible with \( \ast \), in the following sense:

\[
R_1 \ast R_2 \triangleright R' \quad R_1 \ast R_2 \text{ ok} \\
\exists R_1', R_2', R_1' \ast R_2' = R' \land R_1 \triangleright R_1' \land R_2 \triangleright R_2'
\]

That is, an evolution of a composite resource can always be explained as evolutions of the components.

The consistency predicate is of no inherent interest, but plays a technical role of easing the reasoning about combinations of resources. Indeed, at the level of Coq it is important that \( \ast \) be total and (unconditionally) commutative and associative. This remark has been formulated by others before [Nanevski et al. 2010]. Now for \( \ast \) to be total, it must always produce some result, even in situations where its two arguments cannot be meaningfully combined. In such a case, it produces an inconsistent result, i.e., a resource \( \hat{R} \) such that \( \hat{R} \text{ ok} \) does not hold. This allows us to reason relatively easily about combinations of resources; in particular, we rely on Braibant and Pous’ plug-in [2011] for proving equalities and performing matching up to AC.
The above axiomatization is identical to that found in Pottier’s previous work [2013, §10], up to a few technical simplifications. In particular, we are able to get away with just one relation on resources, $<$, whereas the previous paper used two.

Connecting machine states and resources. We assume that a correspondence relation between a machine state and a resource, written $s \sim R$, is given. In the case of heaps, for instance, this would mean that the heap $s$ and the instrumented heap $R$ have a common domain and that, by erasing the extra information in $R$, one finds $s$. We assume that the initial machine state corresponds to a distinguished void resource, i.e., $\text{initial} \sim \text{void}$. We assume that $s \sim R$ implies $R \text{ok}$. No other assumptions are required at this abstract stage.

4.2. Syntax

Values, terms, soups (parallel compositions of several threads), types, and permissions form a single syntactic universe. There is a single name space of variables. Within this universe, defined in Fig. 19, we impose a kind discipline, so as to distinguish several syntactic categories. We find this approach particularly pleasant, as it obviates the need for a quadratic number of weakening and substitution lemmas (type-in-type, term-in-type, term-in-term, etc.).

For the sake of conciseness, we omit the (fairly mundane) definition of the well-kindness judgement. The part concerning types and permissions can already be deduced from Fig. 15, while the complete definition can be found in the Coq code [Balabonski and Pottier 2014]. Furthermore, throughout the paper, we hide the well-kindness premises in every typing rule and theorem. Instead, we use conventional metavariables ($v$, $t$, etc.) to indicate the intended kind of each syntactic element.

There are five kinds, or syntactic categories (Fig. 19).

The values $v$ have kind value. At this stage, they are the variables of kind value (the $\lambda$ binder introduces such a variable) and the $\lambda$-abstractions. Since we do not have a unit value, we also write $()$ for a fixed, arbitrary, closed value.

The terms $t$ have kind term. Every value is a term. Function application $v t$ and thread creation spawn $v \, v_1 \, v_2$ are also terms (the latter is meant to execute the function call $v_1 \, v_2$ in a new thread). The sequencing construct $\text{let } x = t_1 \text{ in } t_2$ is encoded as $(\lambda x. t_2) \, t_1$. By requiring the left-hand side of an application to be a value, and by requiring the arguments of spawn to be values, we reduce the number of evaluation contexts to one. The only (shallow) evaluation context is the right-hand side of an application, $v \, []$. This does not cause any loss of expressiveness: for instance, the application $t_1 \, t_2$ can
4.3. Operational semantics

The calculus is equipped with a small-step operational semantics. The reduction relation acts on configurations $c$, which are pairs of a machine state $s$ and a closed term or soup $t$. In the rules of Fig. 20, the machine state is carried around, but never consulted or modified.

Fig. 20. Kernel: operational semantics

\[
\begin{array}{lll}
\text{initial configuration} & \text{new configuration} & \text{side condition} \\
\hline
s \parallel (\lambda x.t) v & \rightarrow s \parallel [v/x]t & s \parallel t \rightarrow s' \parallel t' \\
\hline
s \parallel E[t] & \rightarrow s' \parallel E[t'] & s \parallel t \rightarrow s' \parallel t' \\
\hline
s / \text{thread} (t) & \rightarrow s' / \text{thread} (t') & s / t \rightarrow s' / t' \\
\hline
s \parallel t_1 \parallel t_2 & \rightarrow s' \parallel t_1 \parallel t_2' & s \parallel t_1 \rightarrow s' \parallel t_2 \\
\hline
s / \text{thread} (D[\text{spawn} v_1 v_2]) & \rightarrow s / \text{thread} (D[()]) \parallel \text{thread} (v_1 v_2) & \\
\end{array}
\]

be encoded as let $x = t_1$ in $x \, t_2$. The Mezzo type-checker performs this transformation (which makes the evaluation order explicit, and names every intermediate result) on the fly.

The soups, also written $t$, have kind soup. They are parallel compositions of threads. A thread has the form thread $(t)$, where $t$ has kind term.

The types $T$, $U$ have kind type; the permissions $P$, $Q$ have kind perm. We write $\theta$ for a syntactic element of kind type or perm.

The types $T$ include the singleton type $=v$, inhabited by the value $v$ only; the function type $T \rightarrow U$; and the conjunction $T \parallel P$ of a type and a permission. As in §3, we write $\top$ for the type $\exists x : \text{value} = v$. Every value has this type. We write $\bot$ for the type $\forall x : \text{type}, x$, which is uninhabited and can be thought of as the least type.

The permissions $P$ include the atomic form $v \, @ \, T$, which can be viewed as an assertion that the value $v$ currently has type $T$, or can be used at type $T$; the trivial permission empty; the conjunction of two permissions, $P \ast Q$; and the permission duplicable $\theta$, which asserts that the type or permission $\theta$ is duplicable.

Universal and existential quantification is available in the syntax of both types and permissions. The bound variable $x$ has kind $\kappa$, which must be one of value, type, or perm: we never quantify over terms or soups.

The syntax is meant to be stable under a kind-preserving substitution. In particular, it should be stable under substitution of a value $v$ for a variable $x$ of kind value. This explains why we allow $=v$ and $v \, @ \, T$, even though, in surface Mezzo, the programmer has access only to $=x$ and $x \, @ \, T$. Thus, a type can refer to a value: this is a value-dependent type system.

It is easy to see that a term cannot refer to a type or permission\(^4\). In other words, the syntax of terms is untyped. This means that the operational semantics can be defined without any reference to the type discipline: Core Mezzo enjoys type erasure. Naturally, this makes type-checking undecidable: an (informal) argument is that Core Mezzo contains System F, whose type-checking problem, in the absence of any type annotation, is undecidable already. In the present paper, this is not a problem, as we are interested only in defining the type system and establishing its soundness. In surface Mezzo, some type annotations are necessary: for instance, every $\lambda$-abstraction must be annotated with its (argument and result) type, and type applications must sometimes be made explicit.

---

\(^4\)In particular, a variable $x$ that appears in a closed term must be $\lambda$-bound, as there are no other binding forms in the syntax of terms.
### 4.4. The typing judgement and the permission interpretation judgement

The main two judgements, whose definitions are mutually inductive, are the **typing judgement** $R; K; P ⊢ t : T$ and the **permission interpretation judgment** $R; K ⊩ P$.

**Overview.** The kind environment $K$ is a finite map of variables to kinds. It introduces the variables that may occur free in $P$, $t$, and $T$. The parameter $K$ is used only in the well-kindness premises, all of which we have elided in this paper. Nevertheless, we mention $K$ as part of the typing judgements, as this helps clarify where variables are bound, and at what kind.

We must sometimes require **canonical type derivations**, that is, typing judgements whose derivation does not use certain rules in certain places. We write $R; K; P ⊢ t : T$ for a typing judgement whose derivation is unrestricted, $R; K; P ⊩ v : T$ for a typing judgement whose derivation is canonical (in that case, the term $t$ must in fact be a value $v$), and use the meta-variable $\circ$ to stand for one of the turnstiles $\vdash$ or $\triangleright$.

A typing judgement $R; K; P ⊢ t : T$ states that, under the assumptions represented by the resource $R$ and by the permission $P$, the term $t$ must have type $T$. One can view the typing judgement as a Hoare triple, where $R$ and $P$ form the precondition and $T$ is the postcondition. The parameter $R$ can be thought of as the “resource” that the term owns and is allowed to exploit, or as the term’s “view” of the machine state. When type-checking an inert program (a source program), $R$ is always $\text{void}$. Non-trivial resources $R$ arise only at runtime: they are used to describe what it means for a running program to be well-typed.

A permission interpretation judgement $R; K ⊩ P$ states that the resource $R$ justifies the permission $P$. If one thinks of $R$ as an (instrumented) heap fragment and of $P$ as a separation logic assertion, one finds that this judgement plays the same role as the interpretation of assertions in separation logic. It gives meaning, in terms of resources, to the syntax of permissions.

**Definition of the typing judgement.** The typing judgement is defined in Fig. 21. The manner in which the turnstiles $\vdash$ and $\circ$ are used can be summed up as follows: a canonical derivation must concern a value $v$ (as opposed to an arbitrary term $t$) and

![Fig. 21. Kernel: the typing judgement
](image-url)
cannot use the rules \textbf{ExistsElim}, \textbf{SubLeft}, or \textbf{SubRight} outside of a \(\lambda\)-abstraction. In other words, a canonical derivation uses only the first six rules of Fig. 21, except within a \(\lambda\)-abstraction, where it may use all of the rules.

The first five rules of Fig. 21 can be viewed as introduction rules: when applied to a value, they define the meaning of the five type constructors. Some of them can also be applied to a term.

\textbf{Singleton} states that \(v\) is one (and the only) inhabitant of the singleton type \(=v\). This rule cannot be applied to a term: that would not make any sense, since \(=t\) is not a well-kindred type.

When applied to a value, \textbf{Frame} is the introduction rule for the conjunction of a type and a permission, \(T \mid P\). When applied to a term, it serves as a frame rule in the sense of separation logic: the permission \(Q\), which is not needed by the computation \(t\), is added simultaneously to its pre- and postconditions.

As usual, \textbf{Function} states that a function \(\lambda x.t\) has type \(T \rightarrow U\) if the body \(t\) has type \(U\) under the assumption that the formal parameter \(x\) has type \(T\). Here, one must separately extend the kind environment \(K\) with the binding \(x : \text{value}\) and augment the precondition \(P\) with the assumption \(x @ T\). The last unusual aspect of this rule is its treatment of duplication. In Mezzo, by convention, every function type is considered duplicable. (We comment on this design choice in §9.) In other words, every function type carries an implicit, built-in \("!"\) modality. This entails a necessary restriction: a permission \(P\) that is available at the function definition site is available also in the function body only if \(P\) is duplicable. For this reason, in the conclusion of \textbf{Function}, the precondition contains duplicable \(P\). The resource \(R\) must be treated in the same manner. If \(R\) is available at the function definition site, only its duplicable core \(R\) is available in the function body.

Applied to a value, \textbf{ForallIntro} is the introduction rule for universal quantification. It can also be applied to a term: we do not enforce the value restriction [Wright 1995] in its strictest form. Mezzo has strong (uniquely-owned) references, whose interaction with polymorphism is sound [Charguéraud and Pottier 2008]. Some restriction \textit{is} necessary, though. The unrestricted combination of polymorphism and weak (shareable) references is unsound [Damas 1985; Tofte 1988; Wright and Felleisen 1994]. This issue appears in Mezzo when one introduces a form of \textit{hidden state}. Indeed, as noted by Pottier [2013], the interaction between hidden state and polymorphism is unsound. In Mezzo, hidden state appears when one introduces locks (§6). When a new lock is allocated by newlock, its invariant (a permission \(P\)) becomes hidden, and the current resource \(R\) (which can be thought of as a “lock typing”) is extended with a mapping of the new lock to \(P\). It is necessary, at this point, to ensure that \(P\) is closed. Thus, newlock must not be allowed to execute under \textbf{ForallIntro}. For this reason, we restrict this rule to a class of \textit{harmless} terms. We omit the definition of this class; suffice it to say that it must encompass the values, must be stable by substitution and by reduction, and (in §6) must not contain a term of the form \(\text{D[newlock]}\).

\textbf{ExistsIntro} is the introduction rule for existential quantification. It is applicable to a value only. A version of \textbf{ExistsIntro} that is applicable to a term can be derived using \textbf{SubRight}, as \(K \vdash [U/x]T \leq \exists x : \kappa.T\) is part of the subtyping relation.

\textbf{Cut} moves information between the parameters \(P\) and \(R\) of a typing judgement. In short, it says, if \(t\) is well-typed under the syntactic assumption \(P_1\), then it is well-typed under the resource \(R_1\), provided \(R_1\) justifies \(P_1\). This justification takes the form of a permission interpretation judgement (the second premise). \textbf{Cut} is the only rule in Fig. 21 with two premises. As is standard in an affine type system, the resource that appears in its conclusion, \(R_1 \ast R_2\), is split between the premises.

\textbf{ExistsElim} is a left-elimination rule for the existential quantifier. The universal quantifier is eliminated via \textbf{SubRight}, as \(K \vdash \forall x : \kappa.T \leq [U/x]T\) is part of the sub-
There are two subsumption judgements:

- for permissions, \( K \vdash P \leq Q \);
- for types, \( K \vdash T \leq U \).
The Design and Formalization of Mezzo, a Permission-Based Programming Language

The latter judgement is defined in terms of the former: by definition, the type subsumption judgement $K, x : \text{value} \vdash T \leq U$ holds when the permission subsumption judgement $K, x : \text{value} \vdash T \leq U$ holds. Again, one might wish to define permission subsumption $K \vdash P \leq Q$ “semantically” by “for every $R$, if $R; K \vdash P$ holds, then $R; K \vdash Q$ holds as well”. However, because a canonical derivation can use subsumption under a $\lambda$-abstraction, the interpretation of permissions depends on permission subsumption. So, this naïve approach is forbidden; it would lead to an ill-formed recursive definition. Because the circularity must go through a $\lambda$-abstraction, it is quite likely that this problem could be resolved via step indexing [Birkedal et al. 2011]. Instead, for the sake of simplicity, we give an axiomatic (i.e., inductive) definition of permission subsumption. We prove, a posteriori, that this axiomatization is sound with respect to the intended “semantic” definition (Lemma A.8). From a pragmatic standpoint, this works well. When some desirable subsumption rule is found to be missing, it can be easily added. This usually gives rise to one new case in the proof of Lemma A.8 and does not affect the rest of the system.

The permission subsumption judgement $K \vdash P \leq Q$ is inductively defined by many rules, of which we show just a few (Fig. 23). In every rule, we omit the assumption “$K \vdash _{}$”, as the parameter $K$ is used only in the well-kindness side conditions, which (by convention) we omit everywhere.

Among the rules not shown in Fig. 23 are: subsumption is reflexive and transitive; conjunction is commutative, associative, and has empty as a unit; every permission is affine (i.e., can be silently discarded); equality of values is reflexive, symmetric, transitive, and a congruence (i.e., equals can be substituted for equals); the universal and existential quantifiers commute with many other type constructors; the universal quantifier can be eliminated, and the existential quantifier can be introduced; each type constructor is contravariant or covariant in each of its parameters.

The rules shown in Fig. 23 are the following.

**MixStarIntroElim** is a compact way of summing up the relationship between the two forms of conjunction, $T \mid P$ and $P \ast P$. One could say that it defines the former in terms of the latter.

**FrameSub** is a version of the frame rule (Frame, Fig. 21), stated as a subsumption axiom. It asserts that a function with fewer side effects can be supplied in a context where a function with more side effects is expected.

**HideDuplicablePrecondition** states that if some function $v$ has precondition $P$, and if $P$ is provably duplicable and exists now, then one may pretend that $v$ has no precondition. This allows a closure to capture a duplicable permission after it has been constructed, whereas **Function** (Fig. 21) allows a closure to capture such a permission when it is constructed.

**Duplicate** states that if $P$ is provably duplicable, then $P$ can be turned into $P \ast P$. We note that if Core Mezzo was extended with support for first-class erasable coercions, i.e., extended in such a way that $P \leq Q$ is itself a permission, then duplicable $P$ would hold.

---

**Fig. 23.** Kernel: permission subsumption (a few rules only; $K \vdash _{}$ omitted)
be just an abbreviation for \( P \leq P \star P \), and the rule \textsc{duplicate} would be replaced with a more general form of modus ponens: \( K \vdash P \star (P \leq Q) \leq Q \). We have not investigated this extension.

\textsc{duplicates} and a family of similar rules (not shown) produce permissions of the form duplicable \( \emptyset \). These rules repeat, at the object level, the rules that define the meta-level predicate \( \emptyset \) is duplicable.

4.6. Typing judgements for soups and configurations

The typing judgement for soups \( R \vdash t \) (Fig. 24, first two rules) ensures that every thread is well-typed (the type of its eventual result does not matter) and constructs the composition of the resources owned by the individual threads. This judgement means that, under the precondition \( R \), the thread soup \( t \) is safe to execute.

The typing judgement for configurations \( \vdash s / t \) (Fig. 24, last rule) ensures that the thread soup \( t \) is well-typed under some resource \( R \) that corresponds to the machine state \( s \). This judgement means that \( s / t \) is safe to execute.

4.7. Type soundness

The kernel calculus is quite minimal: in its untyped form, it is a pure \( \lambda \)-calculus. As a result, there is no way that a program can “go wrong”. Nevertheless, it is useful to prove that (the typed version of) the kernel calculus enjoys subject reduction and progress properties. Because abstract notions of machine state \( s \), resource \( R \), and correspondence \( s \sim R \) have been built in, our proofs are parametric in these notions. Instantiating these parameters with concrete definitions (as we do when we introduce references in §5, locks in §6, and adoption and abandon in §7) does not require any alteration to the statements or proofs of the main lemmas. Introducing new primitive values (such as memory locations in §5 and lock addresses in §6) and operations also does not require altering the statements, but creates new proof cases.

For the sake of brevity, we state only the main two lemmas. A more detailed outline of the proof is provided in an appendix (§A).

**Lemma 4.1 (Subject Reduction).** If \( c_1 \) reduces to \( c_2 \), then \( \vdash c_1 \) implies \( \vdash c_2 \).

**Lemma 4.2 (Progress).** \( \vdash c \) implies that \( c \) is acceptable.

At this stage, a configuration is deemed acceptable if every thread either has reached a value or is able to take a step. This definition is later extended (§6) to allow for the possibility for a thread to be blocked (i.e., waiting for a lock).

5. REFERENCES

In this section, we extend the kernel calculus with heap-allocated references. We also extend the type system, and prove that it ensures data-race freedom.

A reference is a memory block with no tag and just one field. The type of references, \( \text{ref}_m T \), should be viewed as a simplified form of the structural types for memory blocks, such as \#11 and Cons \{ head: a; tail: list a \}, which exist in Mezzo. Mutable and immutable references are modeled, and freezing is supported. The reader is referred to the end of the paper (§8) for a discussion of the features that are not formalized here.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

\[ v, t, T, P ::= \ldots \]
\[ \ell \]
\[ \text{newref } v \mid !v \mid v ::= v \mid \text{ghost} \]
\[ \text{ref}_m T \]
\[ m ::= D \mid X \]

Fig. 25. References: syntax

\begin{tabular}{l|l|l}
  initial configuration & new configuration & side condition \\
  \hline
  \( s / \text{newref } v \) & \( s' / \text{limit } h \) & \( h = \downarrow s \land h' = h :: s' = h'^{*} s \) \\
  \( s / \ell \) & \( s' / v \) & \( h = \downarrow s \land h(\ell) = v \) \\
  \( s / \ell ::= v' \) & \( s' / (\ell) \) & \( h = \downarrow s \land h(\ell) = v \land h' = h[\ell \mapsto v'] \land s' = h'^{*} s \) \\
  \( s / \text{ghost} \) & \( s / (\ell) \) & \( h = \downarrow s \land h(\ell) \) \\
\end{tabular}

Fig. 26. References: operational semantics

5.1. Syntax

We extend the syntax as per Fig. 25.

Values now include the memory locations \( \ell \), which are natural numbers.

Terms now include the three standard primitive operations on references, namely allocating, reading, and writing. We add a special instruction, ghost, which has no runtime effect. It is a syntactic marker that requests special action by the type-checker. Freezing a reference is considered such an action, and must be signaled by a ghost instruction. This instruction does not specify which block one wishes to freeze: at this level, this is not necessary. In the surface syntax, the programmer must use an explicit tag update instruction (§2.1).

Types now include the type \( \text{ref}_m T \) of references whose current content is a value of type \( T \). The mode \( m \) indicates whether the reference is shareable (or duplicable, \( D \)) or uniquely-owned (or exclusive, \( X \)). Only the latter allows writing: this is key to enforcing data-race freedom.

5.2. Heaps

A value heap (or just a heap) \( h \) is either \( \bot \) or a function of an initial segment of the natural numbers to values. (In Coq, such a function is represented as a list.) As far as the operational semantics is concerned, a heap is never \( \bot \). We introduce this “error” element because it allows us, later on, to equip heaps with the structure of an MSA (§5.6). A memory location is a natural number. We write \( \emptyset \) for the empty heap. We write \( \text{limit } h \) for the first unallocated location in the heap \( h \). We write \( h :: v \) for the heap that extends \( h \) with a mapping of \( \text{limit } h \) to the value \( v \). If the memory location \( \ell \) is in the domain of \( h \), then \( h[\ell \mapsto v] \) is the heap that maps \( \ell \) to \( v \) and agrees with \( h \) elsewhere. Later in the paper, we use the same notation for other kinds of heaps: for instance, an instrumented value heap maps memory locations to instrumented values (§5.6).

5.3. Operational semantics

In the kernel calculus (§4), the nature of machine states was completely unspecified. At this point, we need a machine state \( s \) to contain at least a heap \( h \). We specify that a machine state is a tuple of several components, one of which is a heap: \( s ::= (\ldots, h, \ldots) \).

If \( s \) is a machine state, we write \( \downarrow s \) (pronounced “get”) for its “heap” component. If \( h \) is a heap and \( s \) is a machine state, we write \( h'^{*} s \) (pronounced “set”) for the machine state obtained by updating the “heap” component of \( s \) with \( h \).

The reduction rules for references are standard, up to the noise introduced by the conversions between machine states and heaps (Fig. 26). The expression \text{newref } v ex-
NewRef
\[ R; K; \text{newref } v : \text{ref}_m T \vdash !v : T \mid (v \circ \text{ref}_m T) \]

Write
\[ R; K; (v \circ \text{ref}_X T) \ast (v' \circ T') \vdash \text{ref} : T \mid (v \circ \text{ref}_X T') \]

Freeze
\[ R; K; v \circ \text{ref}_X T \vdash \text{ghost} : T \mid (v \circ \text{ref}_D T) \]

Fig. 27. References: typing rules for terms

DecomposeRef
\[ \equiv \exists x : \text{value}( (v \circ \text{ref}_m x) \ast (x \circ T)) \]

UnifyRef
\[ (v \circ \text{ref}_m x \rightleftharpoons v_1) \ast (v \circ \text{ref}_m x \rightleftharpoons v_2) \leq (v \circ \text{ref}_m x \rightleftharpoons v_1) \ast (v \circ \text{ref}_m x \rightleftharpoons v_2) \ast (v_1 = v_2) \]

DUPRef
duplicable \( T \leq \) duplicable (ref\( D \) \( T \))

CoRef
\[ T \leq U \]

Ref
\[ \frac{R_1; K \mid v \circ T}{R_1 \ast R_2; K; \circ \ell : \text{ref}_m T} \]

Fig. 28. References: subsumption rules

Fig. 29. References: typing rules for values

pands the heap with a new binding of \( \text{limit} h \) (the first unallocated memory location) to the value \( v \), and reduces to the memory location \( \text{limit} h \). The expression \( !\ell \) looks up the value stored at location \( \ell \) in the heap. The expression \( \ell := v' \) stores the value \( v' \) at location \( \ell \). The instruction \text{ghost} does nothing: it reduces in one step to the unit value.

5.4. Assigning types to terms
The typing rules for the operations on references appear in Fig. 27.

According to \text{NewRef}, a memory allocation expression \text{newref } v \) consumes \( v \circ T \) and produces a memory location of type \text{ref}_m T \). The mode \( m \) is arbitrary\(^5\). If \( m \) is \( X \), it can be later changed to \( D \) by freezing this reference.

One could restrict \text{NewRef} to the case where \( T \) is the singleton type \( =v \). In that case, the precondition \( v \circ =v \) is a tautology, and the postcondition \text{ref}_m =v \) is an exact description of the newly allocated memory block. This explains our comments of page 10 on memory allocation. Restricting \text{NewRef} in this manner does not cause any loss of expressive power\(^6\).

Reading a reference \( x \) requires a permission \( x \circ \text{ref}_m T \), which guarantees that \( x \) is a valid memory location and stores a value of type \( T \). Because reading a reference creates a new copy of its content without consuming \( x \circ \text{ref}_m T \), \text{Read} requires \( T \) to be duplicable. This is not a problem: in fact, thanks to the subsumption rule \text{DecomposeRef} (§5.5), one could without loss of expressive power restrict \text{Read} to the case where \( T \) is a singleton type. This is illustrated by the expression \text{xs}.head on line 14 of Fig. 7 and its explanation on page 13.

\(^5\)In surface Mezzo, the data constructor determines \( m \). For instance, in the memory allocation expression \text{Cons} (\{ head = x; tail = xs \}), the mode is \( D \), because the data constructor \text{Cons} is part of an immutable algebraic data type.

\(^6\)By using the subsumption rule that introduces an existential quantifier, followed with the subsumption rule \text{DecomposeRef} (§5.5), used from right to left, one can combine the permissions \( x \circ \text{ref}_m =v \) and \( v \circ T \) so as to obtain \( x \circ \text{ref}_m T \).

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
**WRITE** requires an exclusive permission \( x \odot \text{ref}_x T \), which ensures not only that \( x \) is a valid memory location and stores a value of type \( T \), but also that “nobody else” knows about (or has access to) \( x \). The rule allows strong update: the type of \( x \) changes to \( \text{ref}_x T' \), where \( T' \) is the type of \( v' \). Again, one could without loss of expressive power restrict **WRITE** to the case where \( T \) is a singleton type and \( T' \) is the singleton type \( \equiv v' \). This is illustrated by the assignment on line 15 of Fig. 7 and its explanation on page 13.

**FREEZE** states that the instruction ghost can transform \( v \odot \text{ref}_x T \) into \( v \odot \text{ref}_D T \).

All operations on references are considered harmless (§4.4). In particular, memory allocation is harmless: it may be used under read-only access right, whereas \( \text{ref} \) denotes an exclusive read-write access right. Thus, the conjunction of the permissions \( x \odot \text{ref}_D =_1 x_1 \) and \( x \odot \text{ref}_D =_2 x_2 \) implies the equation \( x_1 =_2 x_2 \), which is sugar for \( x_1 = x_2 \).

**DecomposeRef** introduces a fresh name \( x \) for the content of the reference \( v \). This allows reasoning separately about the reference and about its content. Decomposition was used on page 13 when we examined lines 14–17 of Fig. 7. It is reversible: the rule can be used in both directions.

**UnifyRef** states that if we have two names for the content of a reference, then these names must denote the same value. (In that case, we must also have \( m_1 = m_2 = D \).)

For instance, the conjunction of the permissions \( x \odot \text{ref}_D =_1 x_1 \) and \( x \odot \text{ref}_D =_2 x_2 \) implies the equation \( x_1 = x_2 \), which is sugar for \( x_1 = x_2 \).

**DupRef** states that the type \( \text{ref}_D T \) of immutable references is duplicable, provided the type \( T \) of the content is duplicable.

**Coref** states that \( \text{ref}_m \cdot \) is covariant, regardless of \( m \). This is safe, even in the case that \( m \) is \( X \) and (therefore) the reference is mutable, because this is a type of uniquely-owned references [Charguéraud and Pottier 2008; Pottier 2013].

5.6. Resources

An **instrumented value** is \( \# \), \( N \), \( D v \), or \( X v \), where \( v \) is a value. \( \# \) is an “error” element: \( \# \) does not hold, whereas \( iv \) holds of every other instrumented value \( iv \). The instrumented value \( N \) means that “this” memory location is uniquely owned by “someone else”, hence “we” have no information about its content and no right to access it. For any \( m \in \{ D, X \} \), the instrumented value \( m v \) represents full information about a memory location: “we” know that the value stored there is \( v \). Moreover, \( D v \) denotes a shared read-only access right, whereas \( X v \) denotes an exclusive read-write access right.

Instrumented values form an MSA (§4.1), which is defined as follows:

\[
\begin{align*}
D v \times D v &= D v \\
N \times X v &= X v \\
X v \times X v &= X v \\
N \perp N &= N \\
N \perp D v &= N \quad D v \perp iv \quad D v \perp ok \\
X v \perp ok
\end{align*}
\]

The definition of composition \( \times \) requires agreement about which locations are shared (\( D v \)) versus uniquely-owned (\( N \) or \( X v \)). Furthermore, it requires agreement about the content of shareable memory locations (\( D v_1 \times D v_2 \) is \( \# \) if the values \( v_1 \) and \( v_2 \) differ) and requires separation at uniquely-owned memory locations (\( X v_1 \times X v_2 \) is \( \# \)).
The definition of the core \( \hat{\cdot} \) contains the clause \( (Xv) = N \), which means that an exclusive instrumented value contains no shareable information.

The definition of rely \( \hat{\cdot} \) contains the clause \( NDV \), which means that “someone else” may decide to turn a memory location that “they” own exclusively into a read-only, shared location. This clause is needed when proving subject reduction for the operation of freezing a reference.

A heap resource is an instrumented value heap (i.e., either \( \hat{\cdot} \) or a function of an initial segment of the natural numbers to instrumented values). Let us write \( iv \) for an instrumented value and \( ih \) for a non-\( \hat{\cdot} \) instrumented value heap. Heap resources form an MSA, which is defined as follows. First, a non-\( \hat{\cdot} \) heap resource \( ih \) is consistent (i.e., \( ihok \) holds) if and only if all the instrumented values \( iv \) in \( ih \) are consistent (i.e., satisfy \( ivok \)).

The composition operation \( \ast \) is defined pointwise:

\[
\emptyset \ast \emptyset = \emptyset \\
(ih_1 :: iv_1) \ast (ih_2 :: iv_2) = (ih_1 \ast ih_2) :: (iv_1 \ast iv_2)
\]

This definition requires agreement on the allocation limit. This reflects the fact that which locations are allocated (or unallocated) is shared knowledge.

The function “core” is also defined pointwise:

\[
\hat{\emptyset} = \emptyset \\
\hat{ih :: iv} = \hat{ih} :: \hat{iv}
\]

The relation “rely” looks slightly more complex:

\[
\forall \ell \leq \text{limit} \ i h_1 \Rightarrow i h_1(\ell) \vdash i h_2(\ell) \\
\forall \ell \leq \text{limit} \ i h_1 \Rightarrow i h_1(\ell) \vdash i h_2(\ell) \\
\hat{\cdot} \preceq \hat{\cdot} \\
\hat{\emptyset} \preceq \hat{\emptyset} \\
\hat{ih :: iv} \preceq \hat{ih :: iv}
\]

The rule on the right-hand side has three premises. The first premise states that the allocation limit can only increase with time. Thus, deallocation is forbidden\(^7\), and allocation is permitted, subject to the next two premises. The second premise requires that, at every existing memory location, one follows the “rely” relation over instrumented values. The last premise requires that every newly allocated location \( \ell \) be mapped by \( ih_2 \) to some consistent and duplicable instrumented value. The requirement that \( ih_2(\ell) \) be consistent is necessary in order to prove that “rely preserves consistency”, i.e., \( ih_1ok \) and \( ih_1 \vdash ih_2 \) imply \( ih_2ok \). This is one of the MSA axioms (§4.1). The requirement that \( ih_2(\ell) \) be duplicable is not essential at this point; it is exploited in order to establish that certain predicates are stable in §7.6.

The manner in which we have just constructed an “instrumented value heap” MSA on top of an “instrumented value” MSA is generic. We make this definition parametric in the underlying MSA, and re-use it when we define lock resources (§6.4) and adoption resources (§7.6).

In the same way that we have taken a machine state to be a tuple of a heap and possibly other components (§5.3), we take a resource \( R \) to be a tuple of a heap resource

\(^7\)Naturally, in practice, one can use a garbage collector to reclaim unreachable objects. The fact that this is a
and possibly other components. Again, we write $\downarrow R$ for the “heap resource” component of the resource $R$.

A notion of agreement between a value and an instrumented value is defined by "$v$ and $m v$ agree". On top of it, agreement between a heap and an instrumented heap is defined pointwise. It is taken as the definition of correspondence between a machine state and a resource, $s \sim R$.

5.7. Assigning types to values

\textsc{Ref} (Fig. 29) is the introduction rule for the type constructor \textsc{ref}. For now, it is the only rule that assigns a type to a memory location. (New rules that concern memory locations are introduced when we describe adoption and abandon in §7.) This rule splits the current resource into two fragments. The fragment $R_2$ must map $\ell$ to $m v$: this means that $R_2$ grants $m$-access to the location $\ell$ and, at the same time, that $v$ is the value stored at this location. The fragment $R_1$ justifies that $v$ has type $T$. The rule concludes that $\ell$ has type $\textsc{ref}_m T$. Thus, intuitively, the type $\textsc{ref}_m T$ represents the separate ownership of the memory cell at address $\ell$ and of the value $v$ that is currently stored there, to the extent dictated by the type $T$.

Whereas all of the typing rules presented up to this point are independent of the nature of resources, \textsc{Ref} assumes that every resource has a “heap resource” component (§5.6). This appears in the premise $\downarrow R_2(\ell) = m v$, which looks up the location $\ell$ in the instrumented value heap $\downarrow R_2$.

5.8. Type soundness and data-race freedom

Type soundness, as stated earlier (§4.7), still holds in the presence of references. We need not say more: although new cases appear in the proofs of several lemmas, the proof outline (§A) is unchanged.

We now express and prove the fact that “well-typed programs are data-race free”. We need an auxiliary judgement $t$ accesses $\ell$ for $am$. This judgement (whose definition is omitted) means that the term $t$ (which represents either a single thread or a thread soup) is ready to access the memory location $\ell$ for reading or writing, as indicated by the access mode $am$, which is $R$ or $W$. Using this judgement, we define a \textit{racy} thread soup $t$ as one where two distinct threads are about to access a single memory location $\ell$ and at least one of these accesses is a write.

The key reason why racy programs are ill-typed is the following lemma. If a thread soup $t$ is well-typed with respect to $R$ and is about to access $\ell$, then the instrumented heap $R$ must contain a right to access $\ell$; moreover, in the case of a write access, this access right must be exclusive. The proof of this lemma is immediate.

\textbf{Lemma 5.1 (Typed Access).} Every memory access is justified by a suitable access right.

\[ R \vdash t \quad t \text{ accesses } \ell \text{ for } am \quad R \text{ ok} \]

\[ \exists m, \exists v, (\downarrow R(\ell) = m v) \land (am = W \Rightarrow m = X) \]

There follows that a well-typed configuration cannot be racy. Indeed, if two distinct threads are about to access $\ell$, then these threads must be well-typed with respect to two resources $R_1$ and $R_2$, respectively, such that $\downarrow R_1(\ell) = m_1 v$ and $\downarrow R_2(\ell) = m_2 v$ and $R_1 \ast R_2 \text{ ok}$. It is not difficult to check that this implies $m_1 = m_2 = D$, i.e., both accesses are read accesses.

valid implementation technique could be proved separately, if desired. This proof would not exploit the type discipline in any way.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
v, t, T, P ::= ...
| k
| newlock | acquire v | release v
| lock P | locked

(Values: v)
(Problems: t)
(Types: T)

Fig. 30. Locks: syntax

<table>
<thead>
<tr>
<th>initial configuration</th>
<th>new configuration</th>
<th>side condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>s / newlock</td>
<td>s' / limit kh</td>
<td>kh = ↓s ∧ kh' = L ∧ s' = kh'↑s</td>
</tr>
<tr>
<td>s / acquire k</td>
<td>s' / ()</td>
<td>kh = ↓s ∧ kh(k) = U ∧ kh'(k) = kh(k) [→ L] ∧ s' = kh'↑s</td>
</tr>
<tr>
<td>s / release k</td>
<td>s' / ()</td>
<td>kh = ↓s ∧ kh(k) = L ∧ kh'(k) = kh(k) [→ U] ∧ s' = kh'↑s</td>
</tr>
</tbody>
</table>

Fig. 31. Locks: operational semantics

Theorem 5.2 (Data-Race Freedom). A well-typed configuration is not racy.

\[ \vdash h / t \]
\[ \neg (t \text{ is racy}) \]

In conjunction with the subject reduction theorem, this implies that a well-typed program can never reach a racy configuration. Well-typed programs are data-race free.

6. Locks

We now extend the kernel calculus with dynamically-allocated locks. This extension is entirely independent of the previous one (§5). Naturally, references and locks are intended to be used in concert: as illustrated earlier (§1), the point of using a lock is precisely to allow a mutable data structure to be shared between several threads.

6.1. Syntax

We extend the syntax as per Fig. 30. Values now include lock addresses k, which are implemented as natural numbers. For simplicity, we allocate references and locks in two separate heaps, with independent address spaces.

Terms now include the three standard primitive operations on locks, namely allocating, acquiring, and releasing a lock.

Types now include the type lock P of a lock whose invariant is the permission P. The type lock P is duplicable (Fig. 33), regardless of P. This allows several threads to share (and compete for) a lock. Types now also include the type locked. This type is not duplicable. It serves as a proof that a lock is held and (hence) as a permission to release the lock.

6.2. Operational semantics

A lock status is U (unlocked) or L (locked). A lock heap kh maps a lock address to a lock status. We re-use the operations on heaps introduced earlier (§5.2). Mirroring the steps of §5.3, we take a machine state to be a tuple of several components, one of which is a lock heap: s ::= (..., kh,...). Again, we write ↓s (“get”) for the “lock heap” component of the machine state s and kh↑s (“set”) for the machine state obtained by updating the “lock heap” component of s with kh.

The reduction rules for locks appear in Fig. 31. newlock creates a new lock in the locked state (L). This may seem non-standard, and is opposite to surface Mezzo, where lock::new constructs an unlocked lock (§1). In principle, this convention is preferable, as...
because it offers more expressive power: see §6.3. acquire \( k \) needs the lock \( k \) to be unlocked \((U)\), and sets it to the locked state \((L)\). release \( k \) does exactly the opposite.

6.3. Assigning types to terms

The typing rules for the operations on locks appear in Fig. 32. They are analogous to (and inspired by) the axioms of concurrent separation logic with dynamically-allocated locks [O’Hearn 2007; Gotsman et al. 2007; Hobor et al. 2008]; see, in particular, Buisse et al.’s formulation [2011].

According to the rule **NewLock**, the expression ```newlock``` creates a new lock, say \( x \), and produces the permissions \( x \@ \text{lock} \ P \) and \( x \@ \text{locked} \). The permission \( x \@ \text{lock} \ P \) guarantees that \( x \) is a lock and records its invariant \( P \). The invariant can be arbitrarily chosen, but becomes fixed: it cannot be modified after the lock has been created. In surface Mezzo, the invariant is typically provided by the programmer via an explicit type annotation (see Fig. 3). The permission \( x \@ \text{locked} \) guarantees that the lock \( x \) is held and represents a permission to release it.

The type \( \exists x : \text{value} \cdot (=$x | (x \@ \text{lock} \ P) * (x @ \text{locked})$) may seem verbose. It is just an encoding of the intersection type \( \text{lock} \ P \wedge \text{locked} \). One could, if needed, use this sugar. In surface Mezzo, this type is written under the form \((x: \text{lock} \ p | x @ \text{locked})\), which seems fairly natural.

Because locks are created locked, creating a lock does not require any permission. In particular, one may create a lock of type \( \text{lock} \ P \) even in the absence of the permission \( P \). The opposite convention, whereby a new lock is created unlocked, can be simulated by composing ```newlock``` and acquire. This composition requires \( P \) and is therefore in principle less flexible.

According to **Acquire** and **Release**, the expressions acquire \( x \) and release \( x \) have the precondition \( x \@ \text{lock} \ P \), which guarantees that \( x \) is a valid lock with invariant \( P \). acquire \( x \) produces the permissions \( P \) and \( x @ \text{locked} \), whereas, symmetrically, release \( x \) requires and consumes these permissions. It should be intuitively clear that the type system prevents double release: indeed, because \( x @ \text{locked} \) is affine, release \( x \) cannot be invoked twice. Formally, a configuration where a thread attempts to release an unlocked lock cannot make progress (see Fig. 31) and is considered un-
acceptable (see §6.7). Hence, the type soundness theorem implies that a well-typed program cannot reach such a configuration.

The type system does not rule out deadlocks. Formally, a configuration where a thread attempts to acquire a locked lock is considered acceptable.

As noted earlier (§4.4), the interaction between polymorphism and hidden state is unsound. For this reason, newlock is not considered harmless, hence cannot appear under \texttt{FORALLINTRO}. This means that a new lock cannot receive the polymorphic type \([p : \text{perm}] \text{lock} p\). A lock can, however, receive an invariant that has a free variable: this was illustrated in Fig. 3, where the lock 1 has type \texttt{lock s} and \(s\) is a permission variable.

6.4. Resources
In a syntactic proof of type soundness for ML [Wright and Felleisen 1994], the store typing maps every memory location to a (closed) type (the type of its content). In the same manner, here, we wish to maintain a mapping of every lock address to a (closed) permission (its invariant). This mapping should be part of the resource that appears as the first parameter of a typing judgement. In order to do this, we must equip the type of all (closed) permissions with the structure of an MSA (§4.1). Furthermore, because we would like to justify the idea that the type \texttt{lock P} is duplicable, this MSA should be defined in such a way that every element is duplicable.

We achieve this by equipping the type of all permissions with the structure of a “discrete MSA”, deduced from the notion of discrete separation algebra by Dockins et al. [2009]. An element of the \textit{discrete MSA of permissions} is either \(\zeta\) or a permission \(P\). The MSA operations are defined as follows:

\[
P \ast \bar{\P} = P \quad \bar{\P} = P \quad \P \triangle \bar{\P} \quad \bar{\zeta} \triangle \zeta \quad \P \triangle \bar{\zeta} \quad P \text{ ok}
\]

In short, every element is duplicable, and is compatible only with itself. The “rely” relation is the identity: once fixed, a lock invariant can never change.

In addition to this, and independently of this, for every lock in existence, we wish to keep track of “who” (if anyone) has acquired this lock and (hence) has an exclusive right to release this lock. To do this, we use an \textit{MSA of exclusive access rights}. The elements of this MSA are \(\zeta\), \(N\), and \(X\), where \(N\) intuitively represents no access right and \(X\) represents an exclusive access right\(^9\). The MSA operations are defined as follows:

\[
\begin{align*}
N \ast X &= X \\
X \ast N &= X \\
N \ast N &= N \\
_\* \_ &= \zeta
\end{align*}
\]

In short, \(X\) represents an exclusive right to release the lock: \(X \ast X\) is \(\zeta\).

In summary, with every lock, we wish to associate a pair of two independent pieces of information: a lock invariant (\(\zeta\) or \(P\)) and an access right (\(\zeta\), \(N\), or \(X\)). We refer to such a pair as an \textit{instrumented lock status}. Because the product of two MSAs forms an MSA, instrumented lock statuses form an MSA, where (for instance) \((P, X) \ast (P, N)\) is \((P, X)\). That is, the lock invariant represents shared information, whereas the ownership of a locked lock is exclusive.

A \textit{lock resource} is an instrumented lock status heap. Lock resources form an MSA, whose construction is the same as the construction of heap resources in §5.6. As we

\(^9\)This can be viewed as a simplified version of the MSA of instrumented values of §5.6. This time, \(X\) does not carry any argument, and there is no element \(D\).
did there, we take a resource $R$ to be a tuple of a lock resource and possibly other components. Again, we write $\downarrow R$ for the “lock resource” component of the resource $R$.

A notion of agreement between a lock status and an instrumented lock status is defined by “$U$ and $(P,N)$ agree” and “$L$ and $(P,X)$ agree”. This is lifted to a notion of agreement between a lock heap and a lock resource, $kh$ and $R$ agree. In short, this relation means that, for every lock in existence, this lock is locked (according to $kh$) if and only if (according to $R$) “someone” holds the right to release this lock.

To summarize, if one extends the kernel with both references (§5) and locks, then a machine state $s$ is a pair of a value heap and a lock heap; a resource $R$ is a pair of a heap resource and a lock resource. The agreement relation $s$ and $R$ agree requires component-wise agreement.

6.5. Hidden state

The reader might expect the correspondence relation $s \sim R$ to be defined as agreement, $s$ and $R$ agree. After all, this is how we proceeded when we dealt with references (§5.6). However, there is something more subtle about locks. Locks introduce a form of hidden state: when a lock is released, its invariant $P$ disappears; when the lock is acquired again (possibly by some other thread), $P$ reappears, seemingly out of thin air. If we defined $s \sim R$ simply as $s$ and $R$ agree, we would be unable to prove subject reduction for acquire: we would not be able to exhibit a resource fragment that justifies $P$.

Intuitively, while the lock is unlocked, the resource fragment that justifies $P$ is not available to any thread. It is “owned by the lock”, in a certain sense, hence “hidden from the program”. The operations acquire and release perform transfers of ownership between a thread and a lock. We must somehow give a formal account of this phenomenon.

This leads us to refine our understanding of the correspondence $s \sim R$. This relation should not be taken to mean that $R$ represents the entire instrumented state; instead, it means that $R$ is the fragment of the instrumented state that is visible to the program, while the rest is hidden. To account for this idea, we define the relation $s \sim R$ as follows:

\[
\begin{align*}
\text{s and } R &\ast R' \text{ agree} \\
R' \emptyset &\mid \text{hidden invariants of } \downarrow (R \ast R') \\
\hline
s &\sim R
\end{align*}
\]

A machine state is a monolithic entity: it cannot be split. As a result, the premise $s$ and $R \ast R'$ agree implies that the resource $R \ast R'$ represents the entire instrumented state. We split this resource between a visible part $R_v$, which appears in the conclusion, and a hidden part $R'_h$. The second premise requires $R'_h$ to justify the conjunction of the invariants of all currently unlocked locks. The proof cases for acquire and release involve transferring the resource fragment that justifies the invariant $P$ between the hidden resource $R'_h$ and the visible resource $R_v$.

6.6. Assigning types to values

The typing rules $\text{LOCK}$ and $\text{LOCKED}$ (Fig. 34) assign types to lock addresses, thus giving meaning to the types $\text{locked} P$ and locked. Their premises look up the lock re-

---

10Define the hidden invariant of an instrumented lock status by equations hidden invariant of $(P,N) = P$ and hidden invariant of $(P,\_)$ = empty. That is, if the lock is currently unlocked, then its invariant $P$ is currently hidden; otherwise, nothing is currently hidden. Then, define the
\(v, l, T, P \ ::= \ldots\) (Everything)
\[
\text{give } v_1 \text{ to } v_2 \mid \text{take } v_1 \text{ from } v_2 \mid \text{fail} \mid \text{take! } v_1 \text{ from } v_2
\] (Terms: \(t\))
\[
\text{adoptable} \mid \text{unadopted} \mid \text{adopts } T
\] (Types: \(T\))

---

**Fig. 35.** Adoption and abandon: syntax

**Fig. 36.** Adoption and abandon: operational semantics

source \(\downarrow R\). According to \textbf{Lock}, a lock address \(k\) whose invariant (as recorded in \(\downarrow R\)) is \(P\) receives the type \text{lock} \(P\). A well-kindness premise (which, by convention, we have hidden) requires \(P\) to be closed. According to \textbf{Locked}, a lock address \(k\) whose access right (as recorded in \(\downarrow R\)) is \(X\) receives the type \text{locked}.

**6.7. Soundness**

A configuration is now deemed acceptable if every thread either (i) has reached a value; (ii) is waiting on a lock that is currently held; or (iii) is able to take a step. The statements of type soundness (including those of the main intermediate lemmas, described in §A) are unchanged. Well-typed programs cannot go wrong (i.e., they can reach only acceptable configurations) (§4.7) and are data-race free\(^{11}\) (§5.8).

**7. ADOPTION AND ABANDON**

We extend the kernel calculus with adoption and abandon. This extension is entirely independent of locks (§6). It interacts with references (§5), because the concepts of adoption and abandon rely on the existing notions of memory location and memory block. More specifically, in addition to their ordinary role as the address of a memory block whose field(s) can be read and written, memory locations receive two new roles:

1. they serve as adopter addresses;
2. they serve as adoptee addresses, and (for this reason) every memory block receives an extra field, which points from adoptee to adopter.

In order to minimize the interaction between these three roles, we use several orthogonal types for memory locations. In particular, the type \text{ref}_{m} T (§5) retains its original meaning: it allows reading and (if permitted by the mode \(m\)) writing the ordinary field(s) of a memory block. In addition, three new types are introduced to describe and control the use of a memory location as an adopter and as an adoptee. This strategy relies on the fundamental fact that one may have several permissions at the same time for one given object.

**7.1. Syntax**

We extend the syntax as per Fig. 35. Two new instructions appear. The instruction give \(x_1\) to \(x_2\) transfers the ownership of the object \(x_1\) from the executing thread to

\(\text{hidden invariants}\) of a lock resource \(R\) as follows: \(\text{hidden invariants} \ of \ R\ is \ the \ (syntactic) \ conjunction,\ over \ all \ lock \ addresses \(k, \ of \ hidden \ invariant \ of \ R(k)\).

\(^{11}\)The definition of a race does not change with locks. In particular, two competing accesses to a lock are not considered as conflicting, since this is precisely what locks are used for.
the object \( x_2 \). (We use the word “object” as a synonym for “memory block”.) In other words, its effect is that \( x_2 \) adopts \( x_1 \). The instruction take \( x_1 \) from \( x_2 \) has the reverse meaning. It checks (at runtime) that \( x_1 \) is presently adopted by \( x_2 \). If this check is successful, then the ownership of \( x_1 \) is taken away from \( x_2 \) and transferred back to the executing thread. In other words, \( x_2 \) abandons \( x_1 \).

In order to describe the operational semantics of take, we need two auxiliary forms. fail arises as the reduct of an unsuccessful take instruction. take! \( x_1 \) from \( x_2 \) represents an intermediate state of the execution of take. It means that the dynamic check has been performed and has succeeded, but abandon has not actually taken place yet.

No new forms of values appear. The arguments expected by give and take are memory locations.

Three new types appear, namely adoptable, unadopted, and adopts \( T \). The first two describe a memory location in its adoptee role, while the latter describes a memory location in its adopter role.

The permission \( v @ adoptable \) guarantees that \( v \) is a memory location, hence is the address of a memory block, which must have an adopter field. This permission is duplicable. This is a key point: this means that an adoptable object can be aliased. At the cost of a pair of take and give instructions, such an object can be used via any alias.

The permission \( v @ unadopted \) is stronger than \( v @ adoptable \): it guarantees not only that \( v \) is a memory location, but also that \( v \)'s adopter field currently contains \texttt{null} (hence, \( v \) is presently not adopted). It is affine.

The permission \( v @ adopts \ T \) guarantees that \( v \) is a memory location and asserts that every adoptee of \( v \) (i.e., every object whose adopter field points to \( v \)) has type \( T \). It is affine.

The previous three paragraphs offer a descriptive interpretation of the three new permissions: what do these permissions guarantee about the current state? There is also a prescriptive interpretation: in what ways do these permissions allow altering the current state? The permission \( v @ adoptable \) allows reading \( v \)'s adopter field, and is required when one wishes to (attempt to) take \( v \) from its adopter. The permission \( v @ unadopted \) allows writing \( v \)'s adopter field, and is required when one wishes to give \( v \) to some adopter. \( v @ adopts \ T \) is a permission to use the address \( v \) as an adopter: it is required when giving to and taking from \( v \). Furthermore, it represents the collective ownership of all of the adoptees of \( v \) (at type \( T \)), and allows writing their adopter fields (which takes place when they are abandoned).

7.2. From Mezzo to Core Mezzo

There is a little gap between Mezzo, as used in our tutorial introduction to adoption and abandon (§2.5), and the theory presented here (§7). The theory relies on the three types described above, whereas Mezzo hides some of these types from the user.

This gap can be bridged via a desugaring process, which we explain very briefly and informally. The definition of the type graph \( a \) (Fig. 11) is annotated with the clause adopts node \( a \). This means that every graph can be used as an adopter of nodes; more precisely, the permission \( g @ graph \ a \) in Mezzo is desugared as \( g @ graph \ a \ast g @ adopts \ (node \ a) \) in Core Mezzo. Furthermore, in Mezzo, every node can be adopted: to account for this, the permission \( n @ node \ a \) in Mezzo is desugared as \( n @ node \ a \ast n @ unadopted \) in Core Mezzo. Finally, the type dynamic of Mezzo is known as adoptable in Core Mezzo.

Core Mezzo is more verbose, but also more expressive: for instance, a function that takes an argument of type adopts \( (node \ a) \) is applicable not just to a graph, but to any object that adopts nodes. In the future, we would like to make this expressive power available in Mezzo. A set of automatically-generated type abbreviations could be used to retain conciseness.
The structure of the heap. Instead of mapping memory locations to values (§5.2), a
need not extend the machine state with a new component. However, we must modify
Contrary to what happened when we extended the kernel with references and locks, we
7.3. Operational semantics
see, the type discipline guarantees that the value
ℓ
null
field is
account for the presence of adopter fields. When a new block is allocated, its adopter
instruction
take
ℓ
= null
field at address
ℓ
is overwritten with the value
ref
ℓ
′
value. We come back to this issue below (§7.9).
otherwise, the check fails, and the instruction reduces to
fail
ℓ
null
field at address
ℓ
is overwritten and receives the non-null value
ref
ℓ
′
value
ref
ℓ
null
field at address
ℓ
is discarded, which means, intuitively, that once a thread encounters
take
ℓ
null
field at address
ℓ
is guaranteed to succeed, even if the face of interference by
other threads; in the second branch, appropriate action can be taken. The introduction
branch,
take
ℓ
null
field at address
ℓ
is adopted by
ℓ
fails, this failure cannot be caught
otherwise, the check fails, and the instruction reduces to fail.
The fourth rule indicates that
take
ℓ
null
field at address
ℓ
is overwitten with the value
null
The type discipline guarantees that the value previously held in this field was
null.
This is not visible in the operational semantics. We intentionally omit the side
condition
p
null
this makes it clear that no runtime check is required.
The second and third reduction rules describe the runtime check performed by the
instruction
take
ℓ
null
field at address
ℓ
is equal to
ℓ
′
then the check succeeds, and the instruction reduces to
take!
ℓ
null
field at address
ℓ
Otherwise, the check fails, and the instruction reduces to fail.
The fourth rule indicates that
take!
ℓ
null
field at address
ℓ
is just a write instruction: the adopter field at address
ℓ
is adopted by
ℓ
′
null.
The type discipline guarantees that the value previously held in this field was
null.
In a concurrent setting, this is non-obvious: even though the current thread has just ascertained that the adopter field contains
ℓ
′
another thread could in principle have stepped in and written a different value. We come back to this issue below (§7.9).
The last rule in Fig. 36 is a standard reduction rule for
fail.
The evaluation context is discarded, which means, intuitively, that once a thread encounters
fail, it stops.
One could criticize the fact that, if
take
ℓ
null
fails, this failure cannot be caught
and handled. Mezzo provides an expression
adopts
y
x
which (statically) requires the
ownership of
y
and (at runtime) tests whether
x
is currently adopted by
y
producing a
Boolean result. This test can be followed by an ordinary
if
construct; in the first
branch,
take
x
null
y
guaranteed to succeed, even if the face of interference by
other threads; in the second branch, appropriate action can be taken. The introduction
of this construct means that valuable information can be stored in the adopter field: for instance, by using two distinct adopters, a graph traversal can use the adopter field to record which nodes have been visited.

Fig. 37. Adoption and abandon: typing rules for terms

7.3. Operational semantics
Contrary to what happened when we extended the kernel with references and locks, we
need not extend the machine state with a new component. However, we must modify
the structure of the heap. Instead of mapping memory locations to values (§5.2), a
heap
h
now maps memory locations to blocks, where a
block
p : v
is a pair of an
adopter pointer
p
and a value
v.
A
pointer
p
is either
null
or a memory location
ℓ.
The operational semantics of references (Fig. 26) must be slightly adjusted so as to
account for the presence of adopter fields. When a new block is allocated, its adopter
field is
null.
When a block is read or written, its adopter field is ignored and unaffected.
We omit the details.
The operational semantics of adoption and abandon appears in Fig. 36.
The first reduction rule indicates that
give
ℓ
v
to
ℓ
′
is just a write instruction: the
adopter field at address
ℓ
is overwitten and receives the non-null value
ℓ
′.
As we will
see, the type discipline guarantees that the value
p
null
field previously held in this field was
null.
This is not visible in the operational semantics. We intentionally omit the side
condition
p
null:
this makes it clear that no runtime check is required.

NewRefWithAdoption
\begin{align*}
R;K;v@T & \vdash \texttt{newref } v \triangleright 2x : \texttt{value} \triangleright (=x | (x \circ \texttt{ref}_m T) \circ (x \circ \texttt{adopts} \perp) \circ (x \circ \texttt{unadopted})) \\
\texttt{Give} & \quad R;K;(v_2 \circ \texttt{adopts} U) \circ (v_1 \circ \texttt{adopts} U) \circ (v_1 \circ \texttt{unadopted}) \vdash \texttt{give } v_1 \texttt{ to } v_2 : T | \\
& \quad (v_2 \circ \texttt{adopts} U) \\
\texttt{Take} & \quad R;K;(v_2 \circ \texttt{adopts} U) \circ (v_1 \circ \texttt{adopts} U) \circ (v_1 \circ \texttt{adopts} U) \vdash \texttt{take } v_1 \texttt{ from } v_2 : T | \\
& \quad (v_2 \circ \texttt{adopts} U) \circ (v_1 \circ \texttt{unadopted}) \\
\texttt{Take!} & \quad R;K; v \circ \ell \circ \texttt{adopts } U \quad \downarrow R;\ell \vdash \ell \texttt{ is adopted by } \ell' \quad \\
& \quad R;K;P \vdash \texttt{take! } \ell \texttt{ from } \ell' : T | (\ell' \circ \texttt{adopts } U) \circ (\ell \circ \texttt{adopts } U) \circ (\ell \circ \texttt{unadopted})
\end{align*}

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

7.4. Assigning types to terms

The typing rules for adoption and abandon appear in Fig. 37. There is a new version of the typing rule for memory allocation, as well as one typing rule for each of the constructs that were introduced in Fig. 35. Remember that $\top$ is a unit type and that $\bot$ is an uninhabited type.

The typing rule $\texttt{NewRefWithAdoption}$ replaces the rule $\texttt{NewRef}$ that was given earlier (Fig. 27). The previous rule states that a memory allocation instruction of the form “let $x = \texttt{newref} \, v \, \text{ in } \ldots$” consumes the permission $v \leq T$ and gives rise to the permission $x \leq \texttt{ref}_m \, T$. The new rule states that two additional permissions are produced, namely $x \leq \texttt{adopts} \, \bot$ and $x \leq \texttt{unadopted}$. This reflects the fact that the newly allocated memory location plays three distinct roles. The permission $x \leq \texttt{ref}_m \, T$ still states that $x$ denotes the address of a reference, which can be read and (if permitted by the mode $m$) written. The permission $x \leq \texttt{adopts} \, \bot$ states that $x$ can be used as an adopter (i.e., as the second argument of $\texttt{give}$ or $\texttt{take}$), and that all of its adoptees currently have type $\bot$ (i.e., it currently has no adoptees). The permission $x \leq \texttt{unadopted}$ states that $x$ is currently not adopted and also implies that it can be used as an adoptee (i.e., as the first argument of $\texttt{give}$ or $\texttt{take}$; this implication relies on the subsumption rule $\texttt{UnadoptedAdoptable}$, see §7.5). The permission $x \leq \texttt{ref}_m \, T$ is duplicable or affine, depending on the mode $m$, whereas $x \leq \texttt{adopts} \, \bot$ and $x \leq \texttt{unadopted}$ are affine.

$\texttt{Give}$ describes the pre- and postcondition of the instruction $\texttt{give} \, v_1 \, v_2$. Initially, one must have permission to use $v_2$ as an adopter ($v_2 \leq \texttt{adopts} \, U$) and one must have proof that $v_1$ is currently not adopted ($v_1 \leq \texttt{unadopted}$). One must also have proof that $v_1$ has the same type $U$ as the current adoptees of $v_2$ ($v_1 \leq U$). If these requirements are met, then the instruction $\texttt{give} \, v_1 \, v_2$ is safe and, after the instruction, the fact that all adoptees of $v_2$ have type $U$ is preserved, so the permission $v_2 \leq \texttt{adopts} \, U$ remains available. The other two conjuncts of the precondition, on the other hand, are consumed: the ownership of $v_1$ (at types $U$ and unadopted) is effectively transferred from the executing thread to the object $v_2$. This ownership is now covered by the permission $v_2 \leq \texttt{adopts} \, U$, whose footprint grows.

$\texttt{Take}$ is the mirror image of $\texttt{Give}$. One needs permission to use $v_2$ as an adopter ($v_2 \leq \texttt{adopts} \, U$), and this permission is preserved. Furthermore, if the instruction $\texttt{take} \, v_1 \, v_2$ succeeds, then one learns that $v_1$ was among the adoptees of $v_2$, and has now been abandoned by $v_2$. Thus, $v_1$ must have type $U$ ($v_1 \leq U$) and is no longer adopted ($v_1 \leq \texttt{unadopted}$). The ownership of $v_1$ is taken away from $v_2$ and transferred back to the executing thread: the footprint of the permission $v_2 \leq \texttt{adopts} \, U$ shrinks. One apparent source of dissymmetry between $\texttt{Give}$ and $\texttt{Take}$ is that the latter requires proof that $v_1$ has an adopter field ($v_1 \leq \texttt{adoptable}$). This is required for soundness: the instruction $\texttt{take} \, v_1 \, v_2$ would be unsafe if the value $v_1$ turned out to be something other than a memory location (say, a $\lambda$-abstraction). This dissymmetry is only apparent, though. Because permission subsumption allows $v_1 \leq \texttt{adoptable}$ to arise out of $v_1 \leq \texttt{unadopted}$ (see further on), one can derive a variant of $\texttt{Give}$ where $v_1 \leq \texttt{adoptable}$ is part of the postcondition.
FAIL is standard. Since fail never terminates normally, its postcondition is false. In other words, it is deemed to have every type $T$.

TAKE! plays a role in the type preservation proof, but is not used to type-check source programs, since the take! construct is not available to the programmer. We explain it after we present the new typing rules for memory locations (§7.7).

7.5. Subsumption

The subsumption relation is extended with new rules for reasoning about adoption and abandon (Fig. 38).

The rule DUPADOPTABLE states that the type adoptable is duplicable. This is justified, intuitively, by the fact that an adopter field cannot be destroyed: if an adopter field at address $\ell$ exists now, then it exists forever, so it is forever safe to read it (as part of a take attempt).

The subsumption rule UNADOPTEDADOPTABLE states that, when one holds the affine permission $v @ unadopted$, one can obtain, in addition, the duplicable permission $v @ adoptable$. Indeed, $v @ unadopted$ means that there is an adopter field at address $v$ which currently contains a null pointer, whereas $v @ adoptable$ asserts only that there is an adopter field at address $v$. According to NEWREFWITHADOPTION (Fig. 37), a new memory location has type unadopted; hence, via this subsumption rule, a new memory location also has type adoptable, and since this is a duplicable type, this fact remains true forever.

The rule COADOPTS states that the type adopts $T$ is covariant in $T$. This is intuitively justified by the fact that, if every adoptee has type $T$, and $T$ is a subtype of $U$, then every adoptee has type $U$. According to NEWREFWITHADOPTION, a new memory location has type adopts $\bot$. Via this subsumption rule, one can (irreversibly) change this type to adopts $U$, for some type $U$ of one’s choosing, so as to allow this memory location to adopt objects of type $U$.

7.6. Resources

We have explained the intuitive meaning of the types adoptable, unadopted, and adopts $T$. They represent claims about the ownership of certain addresses as adoptee, as adopter, or both. They also represent claims about the value of certain adopter pointers. We must now define this meaning in a formal manner. This is done in two steps. First (§7.6), we extend resources with a new component, an adoption resource, and we update the definition of agreement between a machine state and a resource. Later on (§7.7), we introduce three new typing rules for memory locations, which serve as introduction rules for the types adoptable, unadopted, and adopts $T$, and whose premises contain assertions about the adoption resource.

An adoptee status is one of $\bot$, $N$, and $X p$, where $p$ is a pointer. (Recall that a pointer is either null or a memory location.) $X p$ means that we have exclusive ownership of this memory location as an adoptee, and we know that its adopter pointer is currently $p$. $N$ represents no ownership, and no information about the current value of the adopter pointer. Adoptee statuses form an MSA.

An adopter status is one of $\bot$, $N$, and $X$. (These are the same as the lock statuses of §6.4.) $X$ means that we have exclusive ownership of this memory location as an adopter, whereas $N$ represents no ownership. Adopter statuses form an MSA.

An adoption status is a pair of an adoptee status and an adopter status. For every memory location $\ell$, the roles “$\ell$ as an adoptee” and “$\ell$ as an adopter” are logically independent, which is why we use a pair, whose components can be looked up and updated independently of one another. Adoption statuses form an MSA.
An adoption resource is an adoption status heap. Adoption resources form an MSA.

We will shortly restrict our attention to a subset of “round” adoption resources, which also forms an MSA.

We now introduce several predicates that will be used (§7.7) to give meaning to the types adoptable, unadopted, and adopts T. These predicates are as follows. Here, R ranges over adoption resources.

1. \( R \vdash \ell \) is adoptable holds iff \( \ell \) is in the domain of \( R \), i.e., \( \ell \) is a valid memory location. Because every memory block has an adopter field, this condition is sufficient to ensure that there is an adopter field at address \( \ell \).

2. \( R \vdash \ell \) is unadopted holds iff \( R \) maps \( \ell \) to a pair of the form (X null, _). That is, \( R \) owns \( \ell \) as an adoptee and the adopter pointer at \( \ell \) is currently null.

3. \( R \vdash \ell' \) is an adopter holds iff \( R \) maps \( \ell' \) to a pair of the form (_, X). That is, \( R \) owns \( \ell' \) as an adopter.

4. \( R \vdash \ell \) is adopted by \( \ell' \) holds iff \( R \) maps \( \ell \) to a pair of the form (X \( \ell' \), _). That is, \( R \) owns \( \ell \) as an adoptee, and there is an edge from \( \ell \) to \( \ell' \), which one can think of as “owned by \( R' \)” as well.

5. \( R \vdash \ell \) are the adoptees of \( \ell' \) holds iff the following two conditions are met:
   - \( R \vdash \ell' \) is an adopter holds; and
   - the list \( \ell \) contains all of the addresses \( \ell \) such that \( R \vdash \ell \) is adopted by \( \ell' \) holds, and it contains each such address just once. That is, \( \ell \) is a list of all adoptees of \( \ell' \) according to \( R \), and \( R \) owns every member of the list \( \ell \) as an adoptee, and \( R \) owns every edge from a member of \( \ell \) to \( \ell' \).

Intuitively, these predicates represent knowledge about and ownership of certain fragments of the adoption graph. In particular, \( R \vdash \ell \) is unadopted represents the ownership of an adoptee vertex at address \( \ell \), together with the knowledge that this vertex has no outgoing edge. \( R \vdash \ell \) are the adoptees of \( \ell' \) represents the ownership of a star (in the sense of graph theory) whose center is \( \ell' \), i.e. the ownership of the adopter vertex \( \ell' \), of the adoptee vertices \( \ell \), of the edges from \( \ell \) to \( \ell' \), and the knowledge that this star is complete, i.e., there are no other edges entering \( \ell' \).

We would like these predicates to be affine (i.e., preserved when one moves from \( R \) to \( R \ast R' \)) and stable (i.e., preserved when one moves from \( R \) to \( R' \), where \( R \vartriangleleft R' \) holds). One can, in fact, prove that they are stable. (This exploits the definition of “rely” for the heap MSA, §5.6.) There is, however, one difficulty with affinity: the last predicate, \( R \vdash \ell \) are the adoptees of \( \ell' \), is not affine. That is, the following implication is invalid:

\[
\frac{R_1 \vdash \ell \text{ are the adoptees of } \ell' \quad R_1 \ast R_2 \text{ ok}}{R_1 \ast R_2 \vdash \ell \text{ are the adoptees of } \ell'}
\]

This implication is violated if there exists \( \ell \) such that \( R_2 \vdash \ell \) is adopted by \( \ell' \), that is, there is an edge from the adoptee vertex \( \ell \), owned by \( R_2 \), to the adopter vertex \( \ell' \), owned by \( R_1 \). In that case, the list of all adoptees of \( \ell' \) according to \( R_1 \ast R_2 \) is not just \( \ell \); it includes \( \ell \) as well.

In graphical terms, the problem arises because the adoption graph has been split between \( R_1 \) and \( R_2 \) and we have allowed a star, whose center is \( \ell' \), to be split. Thus, what seems to be a complete star from the point of view of \( R_1 \) is only a fragment of a star from the point of view of \( R_1 \ast R_2 \). In order to avoid this problem, when we split a resource, we should promise to never split a star.

Put another way, the problem arises because \( R_2 \) has a dangling adopter edge: \( R_2 \) owns this edge (i.e., it owns \( \ell \) as an adoptee and owns the edge from \( \ell \) to \( \ell' \)) but it does not own its destination (\( \ell' \) as an adopter is owned by \( R_1 \), hence not owned by \( R_2 \)).
In order to avoid this problem, when we split a resource, we should promise to never create a dangling adopter edge.

Let us say that an adoption resource $R$ is \textit{round} iff it does not exhibit a dangling adopter edge, i.e., $R \vdash \ell$ is \textit{adopted} by $\ell'$ implies $R \vdash \ell'$ is an adopter. It is not difficult to prove that roundness is preserved by the three MSA operations, namely $\ast$, $\cdot$, and $\triangleleft$. This implies that the subset of the round adoption resources forms an MSA.

Thus, we restrict our attention, everywhere, to round adoption resources. This entails some proof obligations: whenever we split a resource, we must prove that the fragments are round. The benefit is that we can now prove that the five predicates defined above are affine and stable. In particular, if $R \vdash \ell$ are the adoptees of $\ell'$, then $\ell$ is a list of all adoptees of $\ell'$, not just with respect to the partial adoption graph represented by $R$, but also with respect to the (implicit) global adoption graph.

Agreement between a heap and an adoption resource is defined in a straightforward way. A block $\langle p \mid v \rangle$ agrees with the adoptee status $X$ $p$. A block $\langle p \mid v \rangle$ agrees with the adopter status $X$. A block agrees with an adoption status (i.e., a pair of an adoptee status and an adopter status) iff it agrees with each of its components. Agreement between a heap and an adoption resource is then defined pointwise. Thus, if $R$ is an adoption resource, $h$ and $R$ agree means that $R$ has $X$’s everywhere and represents the global adoption graph.

Agreement between a heap and a heap resource ($\S 5.6$) must be slightly adapted, because we have altered the structure of heaps by adding an adopter pointer in every memory block. We do so by setting that “$\langle p \mid v \rangle$ and $m$ $v$ agree”, i.e., we simply ignore the adopter pointer. The definition of heap resources is not modified: this helps disturb the existing proof cases as little as possible.

To sum up, once we combine references ($\S 5$), locks ($\S 6$), and adoption and abandon, a machine state $s$ has two components (namely, a heap and a lock heap), whereas a resource $R$ has three components (namely, a heap resource, a lock resource, and an adoption resource). Agreement between a machine state and a resource, $s$ and $R$ agree, requires agreement between the heap and the heap resource, between the lock heap and the lock resource, and between the heap and the adoption resource. On top of that, the correspondence relation $s \sim R$, which accounts for hidden state, remains unchanged ($\S 6.5$).

### 7.7. Assigning types to values

The typing rules \textsc{Adoptable}, \textsc{Unadopted}, and \textsc{Adopts} (Fig. 39) assign types to memory locations, thus giving meaning to the types adoptable, unadopted, and adopts $U$. They come in addition to the typing rule \textsc{Ref} (Fig. 29), which is unmodified.

The premises of these typing rules look up the resource $R$ via the predicates defined above. In the case of \textsc{Adopts}, there are two premises. The first premise means that $\ell$ is a complete list of the adoptees of $\ell'$, that we own each of the addresses $\ell$ as an adoptee, and that we own the address $\ell$ as an adopter. The second premise means that every adoptee, separately, has type $U$; formally, we write $\ell \@ U$ for the iterated conjunction of the permissions $\ell \@ U$, where $\ell$ ranges over the list $\ell$. These two premises must hold separately: the resource $R_1 \ast R_2$ that appears in the conclusion is split between the premises. Intuitively, the type adopts $U$ represents a conjunction of two separate
claims: (i) the ownership of a fragment of the adoption graph, comprising all edges from $\vec{\ell}$ to $\ell'$, or in other words, all edges whose destination is $\ell'$; and (ii) for every member $\ell$ of $\vec{\ell}$, the ownership of the memory location $\ell$, to the extent dictated by the type $U$.

7.8. Soundness
A term $t$ is an answer iff it is either a value or fail. A configuration is now deemed acceptable if every thread either (i) has reached an answer; or (ii) is waiting on a lock that is currently held; or (iii) is able to take a step. The statements of type soundness (including those of the main intermediate lemmas, described in §A) are unchanged. Well-typed programs do not go wrong (§4.7) and are data-race free (§5.8).

Because fail is considered an answer, what we have proved is that “well-typed programs cannot go wrong, but they can fail at a take instruction”.

7.9. Adoption and abandon in a concurrent setting
While the soundness results we just stated apply to the full formalization, including adoption and abandon and concurrency, it is worth giving some details on how these two aspects interact.

The give instruction writes an adopter field, while the take instruction reads an adopter field, performs a pointer comparison, and (if the comparison succeeds) writes this field. Hence these instructions may introduce a race condition on an adopter field, if two threads simultaneously attempt to execute a give and a give, a give and a take, or a take and a take, for the same adoptee $\ell$.

Because the type unadopted (which, on the side of the adoptee, enables a give instruction) is affine, the case of two conflicting give instructions cannot occur in well-typed programs. The type adoptable however (which, on the side of the adoptee, enables a take attempt) is duplicable, hence the other two cases are not avoided by the typing discipline of Mezzo.

This does not invalidate our claim that “well-typed programs are data-race free”, since formally we do not modify the definition of a data race that was given earlier (§5.8). Thus, two conflicting accesses to an ordinary field are regarded as a data race, while two conflicting accesses to an adopter field are not. With this definition, the theorem still holds.

By ignoring adopter fields in the definition of a data race, are we cheating? Arguably, no. Our type soundness theorem guarantees that, whichever interleaving is chosen, “a well-typed program does not go wrong”. Since the operational semantics of take is in two steps, this includes interleavings where the execution of a take instruction is interrupted by another thread. In short, under the assumption of a sequentially consistent memory model, take is sound, even though it is not implemented by an atomic compare-and-set instruction.

Moreover, we believe that, in a well-typed program, the various interleavings of give and take, competing on a single adoptee $\ell$, must lead to the same result. Indeed, the permission $r_2 \oplus$ adopts $U$ that is required by both give and take instructions is affine. Thus, two such competing instructions must target different adopters. This remark allows convincing oneself that the outcome of the dynamic test performed by a take instruction cannot be altered by a competing give or take instruction.

7.10. Design discussion
One might wonder why the type dynamic, or adoptable, is so uninformative: it gives no clue as to the type of the adoptee or the identity of the adopter. Would it be possible to parameterize it so as to carry either information? The short answer is negative. The type adoptable is duplicable, so the information that it conveys should be stable (i.e., forever valid). However, through a combination of strong updates and give and take
instructions, the type of an adoptee may change with time (e.g., a graph node may move from the type node \( \cdot \) to the type node \( \text{int} \)), and the identity of its adopter may change as well (e.g., a node may be adopted by some graph \( g_1 \) at one point in time and later adopted by some other graph \( g_2 \)). Thus, in theory, it does not make sense for adoptable to carry more information.

That said, by using type abbreviations and abstract types, the programmer may define restricted patterns of use of adoption and abandon, and in return, obtain more informative types. She may, for instance, define a pair of parameterized types adoptable_at \( a \) and more informative types. She may, for instance, define a pair of parameterized types adoptable_by \( y \) and unadopted_at \( a \), accompanied with suitable variants of the \text{give} and \text{take} operations, such that an object of one of these types is guaranteed to ever be adopted only if (when) it has type \( a \). (We omit the details.) She may even define a pair of types adoptable_by \( y \) and unadopted_by \( y \), where the parameter \( y \) has kind value, such that an object of one of these types is guaranteed to ever be adopted only by the adopter \( y \). These idioms could be defined as part of the standard library.

There are a number of ways in which adoption and abandon could be optimized or enhanced. Let us briefly mention two potential improvements:

— To avoid paying the cost of one adopter field in every object, one should introduce a distinction between “slim” and “fat” objects (as suggested by footnote 3 on page 23).
— To permanently delegate \( y_1 \)'s adopter role to some other object \( y_2 \), one could add an instruction \text{merge} \( y_1 \) into \( y_2 \). The effect of this instruction would be that all adoptees of \( y_1 \) immediately become adopted by \( y_2 \), and \text{take} \( x \) \text{from} \( y_1 \) and \text{give} \( x \) to \( y_1 \) thereafter become synonymous with \text{take} \( x \) \text{from} \( y_2 \) and \text{give} \( x \) to \( y_2 \). Its implementation would rely on a union-find data structure, and would cost one extra field per (adopter) object.

8. EXTENSIONS

Let us briefly outline how the formal definition of Core Mezzo should be extended so as to reduce the gap with Mezzo. Perhaps the most important feature of Mezzo that is currently missing is algebraic data types. (In fact, this feature was present in an earlier version of the machine-checked proof, but has not yet been ported to the current version.) This feature can be introduced in several steps, as follows:

(1) Allow memory blocks to have multiple fields, designated by an integer offset. The primitive type \text{ref}_m \{ T \} is replaced with a record type, say \text{ref}_m \{ \vec{T} \}.

(2) Equip memory blocks with a tag, represented as an integer. The record type \text{iref}_m \{ \vec{T} \} is replaced with a structural type \( i \in m \{ \vec{T} \} \), where \( i \) is a tag. This type is analogous to the structural type Cons \{ \text{head}: a; \text{tail}: \text{list} \ a \} found in Mezzo. Introduce the tag update instruction, which allows changing the tag of a memory block.

(3) Introduce a union type, of the form \( T_0 \cup \ldots \cup T_{n-1} \), with the condition that the \( i \)-th summand, \( T_i \), must exhibit the tag \( i \). (That is, \( T_i \) should be a subtype of \( i \in m \{ \vec{T} \} \), for some \( m \) and \( \vec{T} \).) Add the injection axiom \( T_i \leq T_0 \cup \ldots \cup T_{n-1} \). Introduce a switch construct, whose typing rule refines \( T_0 \cup \ldots \cup T_{n-1} \) to \( T_i \) in the \( i \)-th branch.

(4) (This item is independent of the previous three.) Add parameterized iso-recursive type definitions, of the form \( \tau \overset{x}{=} T \), which are (un)folded via subsumption. For instance, the nominal type \text{list} \ x \ could be declared isomorphic to the union type \( 0 \ D \{ \} \cup 1 \ D \{ x ; \text{list} \ x \} \).

In Mezzo, the type \text{ref} of mutable references is defined as part of the core library: \text{data mutable ref} \ a = \text{Ref} \{ \text{contents}: a \}. Thus, \text{ref} \ t \ is a nominal type, and is interconvertible with the structural type \text{Ref} \{ \text{contents}: t \}. The latter corresponds in spirit to the primitive type \text{ref}_x \ T \ of §5. Similarly, in Mezzo, one can define a type of immutable references, as follows: \text{data iref} \ a = \text{IRef} \{ \text{contents}: a \}. The type
The Design and Formalization of Mezzo, a Permission-Based Programming Language

IRef { contents: t } then corresponds to the primitive type ref_D T. Freezing, which is written tag of r <-> IRef in Mezzo, is represented by the instruction `ghost` in §5.

The implementation of write-once references (§2.1) is a closely related example of use of these concepts.

A few more features, perhaps of lesser interest, would have to be formalized if one wished to close the gap between Core Mezzo and Mezzo:

— Immutable tuples.
— Recursive functions.
— Named (as opposed to numbered) data constructors and fields. In Mezzo, the names of data constructors have lexical scope. Field names are treated in a different way: they are overloaded, and the resolution of overloading is type-directed.
— Compilation units, with implementation (.mz) and interface (.mzi) files.

Mezzo does not have exceptions. If one wished to extend Mezzo with exceptions, then, in order to preserve type soundness, every function type would have to be annotated with a list of the exceptions that may be raised, and, for each such exception, the permission that is returned in case this exception is raised. From the language designer's point of view, this would duplicate a lot of the machinery that exists for dealing with sum types. So, our stance, for the moment, is that it seems preferable to simulate exceptions, either by returning sums or by taking multiple continuations as arguments. More experience with the language is needed in order to evaluate whether this position is tenable in practice.

9. RELATED WORK

The literature offers a wealth of type systems and program logics that are intended to help write correct programs in the presence of mutable, heap-allocated state. We review some of them and contrast their design principles with those of Mezzo.

Many of these systems rely on a concept of “ownership”, or “permission” to perform certain actions. However, there are many ways of approaching this concept and its implications on the programming discipline. What is an owner? A principal, a thread, an object in memory? What kind of privileges are associated with ownership? Does the discipline restrict who can read, who can write, who can point to an object? What global invariant does the discipline enforce?

9.1. Annotating types with owners

Ownership Types and its descendants [Clarke et al. 2013] are type systems where types are explicitly annotated (or implicitly associated) with owners. These systems enforce topological restrictions on the heap (i.e., they restrict which paths may exist in the heap) and may in addition enforce a form of “encapsulation” (i.e., limit the operations that can be performed through certain references).

In Clarke et al.’s original paper [1998], every object has (at most) one owner, and an owner is an object. An “owner-as-dominator” principle is enforced: every path from a root to an object x must go through x’s owner.

In Clarke et al.’s later paper [2001], there is a partially ordered set of ownership “contexts”, where a context could be a class, a package, etc. (One may also think of a context as a “region”.) An object x has both an “owner context”, which restricts which objects may point to x, and a “representation context”, which restricts which objects x may point to. The system enforces a “containment invariant”: a pointer from x to y may exist only if rep(x) <= owner(y) holds.

Another member of the family, Universe Types [Dietl and Peter 2005], imposes an “owner-as-modifier” principle. There, arbitrary paths are allowed to exist in the heap, but only those that go through x’s owner can be used to modify x. This approach is
motivated by the desire to better support program verification, as it allows the owner
to impose an object invariant.

The literature on Ownership Types is too vast for us to survey here; the reader is
referred to the comprehensive overview by Clarke et al. [2013].

9.2. Annotating types with permissions

In a permission system, types are annotated not with owners, but with permissions. The permission carried by a pointer tells how this pointer may be used (e.g., for reading and writing, only for reading, or not at all) and how other pointers to the same object (if they exist) might be used by others.

In Boyland’s work [2003], a permission is a fraction \( q \) in the interval \((0, 1]\). The unit fraction 1 means “we” have read-write access, whereas “others” have no access at all. Any fraction less than 1 means “everyone” has read-only access. Permissions can be split and recombined, which allows a heap fragment to transition from the state “read-write, exclusive” to the state “(temporarily) read-only, shared” and back.

Javari [Tschantz and Ernst 2005] extends Java (where, by default, “everyone” has read-write access) with the permission \( \text{readonly} \), which means that “we” have read-only access, while “others” may have read and/or write access.

Plural [Bierhoff and Aldrich 2007; Bierhoff et al. 2009; Bierhoff et al. 2011], which takes the form of a mostly automatic intra-procedural analysis, includes Boyland’s fractions, under the names unique and immutable \((q)\). It also includes the permissions full, which means “we” have read-write access, whereas “others” have read-only access; and pure, which is the dual of full. Most importantly, Plural introduces the idea that a permission can carry typestate information (e.g., an iterator object may be in one of two states, “available” or “finished”) and that a unique permission allows a strong update (i.e., a typestate change).

In principle, permission systems need not impose any topological restrictions on the heap. Their distinguishing feature is that permissions have a clear interpretation in terms of “rely” and “guarantee”. A permission dictates what actions “we” can perform on an object and what assumptions “we” can make about its current state. Dually, it dictates what “others” may assume or do. This duality between what one may assume and what one may do seems particularly pleasant, as it ties “policy” and “mechanism” together in a compelling way. By “policy”, we mean the properties of objects that one wishes to enforce (e.g., Dietl and Müller’s object invariants [2005], or Bierhoff and Aldrich’s typestates [2007]). By “mechanism”, we mean the details of which operations through which references must be allowed or forbidden. Ultimately, one may argue, policy matters more than mechanism: as put in a provocative manner by Fähndrich, “we couldn’t care less about aliasing” [Clarke et al. 2004].

9.3. Replacing types with permissions

The systems mentioned so far are refinements (i.e., restrictions) of a traditional type discipline. Separation logic [Reynolds 2002] departs from this approach. Like many other program logics, it does not rely on a type system: in principle, it can be used equally well to reason about typed or untyped programs. It takes the idea of permission systems to an extreme, where the permission is everything and there isn’t a need for types any more.

Separation logic obeys a principle that we dub “owner-as-asserter”. (In O’Hearn’s words, “ownership is in the eye of the asserter” [2007]. This principle has also been referred to as “owner-as-privilege” [Clarke et al. 2004].) As usual in a program logic, objects are described by assertions. The novelty lies in the fact that a separation logic assertion is a permission: it not only represents knowledge about the current state of an object, but also implies that “we” may act in certain ways on this object, and
that “others” may not. Thus, “to assert is to own”. The assumption that “x is a linked list” means that “we” may read and write the cells that form this list, and “others” may not. Whereas the systems mentioned previously (§9.1, §9.2) combine traditional structural descriptions (i.e., types) with owner or permission annotations, separation logic assertions are at once structural descriptions and claims of ownership.

Mezzo follows this principle. A permission in Mezzo is fundamentally an assertion in the sense of separation logic. (Technically, the permission interpretation judgement $R; K \vdash P$ in §4.4 is defined in essentially the same manner as the interpretation of assertions in separation logic.) Let us give three key motivations for this design decision.

One motivation is that this makes the system conceptually less redundant. Indeed, a traditional type assumption, such as “x has type list”, can be viewed as a duplicable permission in the sense of Mezzo, stating that “everyone may assume that x points to a linked list, and everyone must preserve this fact”. From a pragmatic standpoint, eliminating the redundancy between (traditional) types and permissions leads to a more concise system. In Mezzo, types are not annotated with owners (§9.1) or with access permissions (§9.2). Polymorphic code is just type-polymorphic; it does not have to be both type- and annotation-polymorphic. That said, there are situations in Mezzo where something that resembles “ownership annotations” appears anyway: this is the case, as a matter of fact, when working with static regions (§9.6).

The second motivation for this design decision is that it allows the “type” of an object to change. In a traditional type system, types are fixed: because every object is potentially shared, its type cannot be altered. In a system where permissions are layered on top of types (§9.2), a strong update can alter the information carried by a unique permission. For instance, in Bierhoff and Aldrich’s system [2007], the “typestate” of an object can be changed by its owner. However, the object’s underlying type still cannot be modified. In separation logic or in Mezzo, where information about the “type” of an object is carried by the permission, a strong update can alter this information. This enables gradual initialization (§2.2), memory re-use, and certain forms of typestate tracking [Guéneau et al. 2013].

The last motivation is that “owner-as-asserter” seems a very natural approach from the standpoint of program verification. More specifically, a program logic for Mezzo could conceivably take advantage of its type and permission discipline. Permissions would be annotated with logical assertions, expressed in a standard logic. A simple mechanical procedure would extract a collection of proof obligations out of a well-typed program. These obligations would be passed on to a standard theorem prover. The prover would not need to reason about separation, because this reasoning would have been carried out already by the Mezzo type-checker.

In contrast with ownership type systems (§9.1), Mezzo does not purposely impose any topological restrictions on the heap. Nevertheless, it provides strong “end-to-end” safety guarantees: well-typed programs do not go wrong (which, in the presence of type-changing updates, is a non-trivial result) and are data-race free.

### 9.4. Containers, ownership, and ownership transfer

Does a container “own” its elements? Arguably, there are situations where one wishes to view the elements as owned by the container, and situations where one doesn’t.

In principle, ownership type systems (§9.1) can easily describe both situations. It is just a matter of annotating the type of the elements with an appropriate owner.

In Mezzo, the type of a container is typically parameterized with the type a of its elements. One could say that a container always “owns” its elements, but only to the extent described by the type a. If the parameter a is instantiated with an affine type, such as ref int, then the container effectively has unique access to the elements. If the parameter a is instantiated with a duplicable type, such as list int or dynamic,
then the elements may be shared. One could say, in the latter case, that the container “does not own” its elements.

The functions that insert an element into a container, or extract an element out of a container, typically receive the following concise types:

\[
\begin{align*}
\text{val put: } & \ [a] \ (\text{container } a, \ \text{key}, \ \text{consumes } a) \rightarrow () \\
\text{val get: } & \ [a] \ (\text{container } a, \ \text{key}) \rightarrow a
\end{align*}
\]

We have written container a for the type of the container, whose elements have type a, and key for an unspecified type of keys. These types express in a fairly natural way the transfer of ownership that takes place when an element is inserted or extracted. A call \(\text{put}(c, k, x)\) consumes the permission \(x @ t\), if \(c\) has type container \(t\). A call \(\text{let } x = \text{get}(c, k) \text{ in } \ldots\) produces the permission \(x @ t\).

In contrast, the early ownership type systems (§9.1) do not keep track of uniqueness, hence do not allow strong updates or ownership transfer. Later proposals [Clarke and Wrigstad 2003; Müller and Rudich 2007] include a notion of “external uniqueness” which supports ownership transfer.

One problem with containers that “own” their elements is that they typically do not allow “consulting” or “borrowing” an element, i.e., getting access to it, without taking it out of the container. We have discussed this issue in §2.4. In Mezzo, one works around this problem by using a container that “does not own” its elements, that is, by using a container at a duplicable element type.

There are several ways by which a mutable object may be assigned a duplicable type. (1) Adoption and abandon allows assigning the duplicable type \texttt{dynamic} (also known as adoptable, see §7) to an object. Access to the object, together with information about its “real” type, is obtained via a take instruction. (2) Regions (§9.6) allow assigning the duplicable type \texttt{rref rho t} to a mutable cell that inhabits the region \(\rho\). (3) Pairing an object \(x\) together with a lock that protects the permission \(x @ a\) results in a package of duplicable type protected \(a\), which is defined as an abbreviation for \((x: \texttt{unknown}, \ \text{lock (}x @ a\text{)})\), where \texttt{unknown} is a surface notation for \(\top\), that is a duplicable type that gives no useful permission. Access is obtained by acquiring the lock. An example of this final possibility is a simple definition of communication channels in Mezzo, which can be done using a (mutable) queue protected by a lock [Protzenko 2014].

9.5. Linearity, singleton types, and capabilities

Wadler [1990] notes that an object of linear type can be updated in place. Furthermore, he proposes that a value that represents “the file system” should be assigned a linear type. In the same manner, Clean uses uniqueness types to ensure that the “world” is never duplicated [Smetsers et al. 1994; Achten and Plasmeijer 1995]. The early linear type systems are often very restrictive, though. Because reading a reference creates a copy of its content, a reference whose content has linear type cannot be read in the usual manner. A destructive read operation, or a swap operation, must typically be used instead.

Perhaps the main reason why linearity (or affinity, or uniqueness) matters is that the type of a unique object can change with time, through a “strong update”. Smith et al.’s Alias Types [2000] is perhaps the first system where this idea is clearly pointed out and exploited.

Alias Types makes another major contribution, which is to recognize that it does not really matter that there exist at most one pointer to an object. It is fine for multiple pointers to exist, as long as there is a unique (static) capability to dereference any of these pointers. Thus, Smith et al. introduce the distinction between values, which are
In order to keep track of which capabilities give access to which objects, and (at the same time) to keep track of the known equations between pointers, Alias Types uses singleton types. A pointer \( p \) receives a singleton type \( \text{ptr} \ell \), where \( \ell \) is a type-level name for a memory location. A pointer \( q \) that happens to also have type \( \text{ptr} \ell \) is statically known to be equal to \( p \). A capability \( \{ \ell \mapsto \tau \} \) guarantees that the memory location represented by \( \ell \) currently contains data of type \( \tau \). This capability allows reading and writing, through \( p \) or \( q \), and a strong update can change this capability to \( \{ \ell \mapsto \tau' \} \), for some new type \( \tau' \).

Analogous ideas appear in the contemporary paper by Walker et al. [2000] on region-based memory management. In one paper, there is one capability per object; in the other, there is one capability per region. Vault [DeLine and Fähndrich 2001; Fähndrich and DeLine 2002] has capabilities for both single objects and regions, together with a “focusing” mechanism for temporarily singling out an inhabitant of a region.

Reynolds notes early on [2002] that separation logic is closely related to Alias Types. He sketches a translation from a fragment of Alias Types into separation logic, whereby a capability \( \{ \ell \mapsto \tau \} \) is translated to a “points-to” assertion.

Quite obviously, Mezzo is strongly inspired by both Alias Types and separation logic. However, it is meant to be a type discipline that helps structure high-level programs, as opposed to a tool for low-level reasoning. This guides some of our design choices. For instance, Mezzo has tagged sums (that is, algebraic data types), whereas separation logic usually has untagged union and null pointers.

One contribution of Mezzo with respect to Alias Types is a simpler notion of singleton type. As Mezzo is value-dependent, a type can refer directly to a value. Thus, every variable \( p \) has the singleton type \( =p \). There is no need for introducing a type-level name \( \ell \) in addition to the name \( p \). From a pragmatic point of view, this economy is quite important. From a theoretical point of view, this is also a simplification: whereas the type \( \text{ptr} \ell \) of Alias Types is at the same time a pointer type and a singleton type, Mezzo has two distinct (orthogonal) type constructors, namely \( \text{ref a} \) and \( =p \), for these concepts. In that sense, Mezzo is analogous to separation logic, where an assertion can refer directly to a value, and where “points-to” and equality are orthogonal concepts.

Although capabilities originally did not have first-class status, this was considered by Walker and Morrisett [2000]. L\(^3\) [Ahmed et al. 2007] and Alms [Tov and Pucella 2011] are also linear or affine \( \lambda \)-calculi where capabilities have first-class status. In fact, in these systems, capabilities are viewed as ordinary values, which one hopes the compiler can erase. In contrast, Mezzo guarantees that permissions are erased: they do not appear in its operational semantics (§4). Furthermore, the flow of permissions in a Mezzo program is (to a large extent) implicit; it is reconstructed by the type-checker.

In the tradition of linear logic, many linear or affine type systems in the literature adopt the convention that types are by default not duplicable. An explicit modality, written “!” , must be used to indicate that a value is duplicable. This is the case, for instance, in \( L^3 \) [Ahmed et al. 2007], and in Pottier’s earlier work [2013], which was inspired in part by dual intuitionistic linear logic [Barber 1996]. From a theoretical point of view, this approach may seem pleasant. In practice, however, it seems too verbose to be tolerable. Several authors have suggested recording duplicability information at the level of kinds [Charguéraud and Pottier 2008; Mazurak et al. 2010; Tov and Pucella 2011]. In Mezzo, in contrast, this information is implicit in the syntax of types. The type \( \text{int} \), for instance, is inherently duplicable. The type \( \text{ref int} \) is inherently non-duplicable. The tuple type \( (t, u) \) is duplicable if and only if \( t \) and \( u \) are duplicable; and so on. The type-checker “knows” these rules and applies them transparently. (As noted in §2.2, the type-checker infers the rules that govern user-defined types, such as
list t.) Formally, **duplicable** t is viewed as a permission, and the rules listed above are permission subsumption rules. This approach seems lightweight, both in theory and in practice. As illustrated by the types of get and set in Fig. 5, it is easy to express both quantification with respect to an arbitrary type and quantification with respect to a duplicable type. Mezzo's duplicable permissions are analogous to the necessary assertions found in CaReSL [Turon et al. 2013].

By convention, in Mezzo, every function type is duplicable. This means functions are easy to use: they can be freely shared, and can be called as many times as one wishes, provided one is able to supply an argument of appropriate type. The flip side is that this can make functions difficult to construct: in particular, a function is not allowed to capture an affine permission that exists at its definition site (see **FUNCTION** in §4.4). This is not a problem, though, as the affine type $T \rightarrow U$ of functions that can be called at most once can be defined as an abbreviation for $\exists p : \text{perm}.(((T \mid p) \rightarrow U) \mid p)$. (This is Core Mezzo syntax.) This can be intuitively understood as a package of a shotgun and one cartridge, represented by the abstract permission $p$. The shotgun can be fired at most once, because it consumes $p$, which is not known to be duplicable. In fact, Mezzo allows encoding not only the type $T \rightarrow U$ of one-shot functions, but also variations on this theme. For instance, an affine type of functions that can be called many times can be defined as $\exists p : \text{perm}.(((T \mid p) \rightarrow (U \mid p)) \mid p)$. As another example, double-barrelled CPS style, where one is handed a pair of a success continuation and a failure continuation and one is allowed to invoke at most one of them, can be easily and faithfully encoded in Mezzo.

### 9.6. Regions, nesting, adoption and abandon

From a type-theoretical point of view, a region is a type-level name for a set of values, typically a set of memory locations. A region may also exist at runtime, in which case it is typically an area where objects can be dynamically allocated one by one, and can be deallocated all at once. In Tofte and Talpin's original work [1994; 1997], regions exist at type-checking time and at runtime. This is the case also in Walker et al.'s paper [2000], in Vault [DeLine and Fähndrich 2001; Fähndrich and DeLine 2002], and in Cyclone [Swamy et al. 2006]. In some of these works, so as to avoid confusion, the type-level entity is referred to as a **region**, whereas the runtime entity is referred to as a **region handle**. Some languages where regions exist only at type-checking time include Haskell, whose ST monad is indexed with a region, and Rust [The Mozilla Foundation 2014], whose "borrowed pointer" type &T is indexed with a "lifetime", another name for a region. In both cases, regions are used to ensure that a piece of memory is not accessed outside of a certain lexical scope.

Nesting [Boyland 2010] is a mechanism by which an arbitrary permission $p$ can be "nested" within an object $y$. Then, whoever has (exclusive) access to $y$ implicitly also has access to $p$. The act of nesting $p$ within $y$ consumes the permission $p$ and produces a new permission, known as a "nesting witness", written $p \prec y$. Because nesting cannot be undone, it is safe to view nesting witnesses as duplicable. A nesting witness $p \prec y$, combined with an (arbitrary) exclusive permission for $y$, allows gaining temporary access to $p$, as follows. When one wishes to obtain $p$, one gives up the exclusive permission $y @ \ldots$. Symmetrically, when one is finished working with $p$, one gives up $p$ and recovers $y @ \ldots$. These mechanisms can be viewed as a generalization of Fähndrich and DeLine's adoption and focus [Fähndrich and DeLine 2002]. They are purely static: at runtime, nothing happens. Although, in this presentation, the object $y$ exists at runtime, this is

---

12In fact, Boyland allows per-field nesting: one may nest $p$ in $y.f$. He also has fractional permissions, which interact with nesting in an interesting way. We omit these refinements from our discussion.
abstract region

abstract rref (rho : value) a

fact duplicable (rref rho a)

val newregion: () -> region
val newrref: [rho: value, a] (consumes x: a | rho @ region) -> rref rho a
val get: [rho: value, a] (r: rref rho a | duplicable a | rho @ region) -> a
val set: [rho: value, a] (r: rref rho a, consumes x: a | rho @ region) -> ()

Fig. 40. A signature for regions, which can be implemented on top of nesting

not essential. What is required is that the permission y @ ..., which is used to obtain access to the permissions nested within y, be affine.

Mezzo can be extended with nesting. We have axiomatized this extension (whose details are available online [Pottier and Protzenko 2014]) and have written a few library modules that rely on it. We have not proved type soundness for this extension. We believe that it should be possible to do, based on the kernel presented in this paper.

On top of nesting, one can implement (static) regions. For instance, in Mezzo with nesting, one can define a library module that offers the signature in Fig. 40. The function call let rho = newregion() in ... produces a new region rho. The affine permission rho @ region serves as a unique right to access the cells allocated in the region rho. The duplicable type rref rho a describes a reference (a memory cell) that inhabits the region rho. This abstract type is defined internally as (x: unknown | nests rho (x @ a))\textsuperscript{14}. The operations newrref, get, and set respectively allocate, read, and write a reference which (conceptually) inhabits the region rho. Each of these operations requires (and returns) the permission rho @ region. The operation get is restricted to the case where the type a is duplicable: this is another instance of the “borrowing problem”, which we have discussed at length (§2.4).

This encoding of regions shows that Mezzo (with nesting) subsumes Haskell’s ST monad. In Haskell, monads are used to encapsulate many sorts of effects, including mutable state [Peyton Jones and Wadler 1993]. While the IO monad gives unrestricted access to mutable references in the style of ML, the ST monad offers more controlled access. The type of a reference reflects which region it inhabits. The type of a computation reflects which region it affects: a computation that affects region rho and produces a result of type a has type ST rho a. In Mezzo, this would be encoded as a function of type ( | rho @ region) -> a, i.e. a function that requires and returns the permission rho @ region. One benefit of this discipline, in Haskell, is that a computation that uses only local state can be deemed pure; this is encoded in the type of the primitive operation runST. In Mezzo, an analogue of runST would be implemented in a straightforward way, as follows:

val runST [a] (f: [rho: value] ( | rho @ region) -> a) : a =
  let rho = newregion() in
This implementation of runST is just an exercise, though: in practice, a user would typically use newregion() directly, instead of calling runST. We note in passing that in Mezzo it is easy to work with multiple regions simultaneously, whereas we believe that this would be awkward in Haskell.

In retrospect, Haskell's primitive state monads can be viewed as a way of ensuring that state is treated in a linear manner, even though Haskell's type system does not have linear types. In Mezzo, they can be programmed up, by relying on the primitive notion of an affine permission.

Mezzo's adoption and abandon is inspired by adoption and focus [Fähndrich and DeLine 2002] and by nesting [Boyland 2010]. The common purpose of all three mechanisms is to have just one permission for a group of objects (a region), together with a way of recovering a permission for an individual member of the group (a region inhabitant), when necessary.

Adoption in the sense of Fähndrich and DeLine [2002] and nesting are purely static mechanisms. They are irreversible: membership in a region, or nesting, cannot be undone. Access to an inhabitant can be gained only temporarily. Simultaneous access to two inhabitants (if supported) requires proving that they are distinct. In contrast, adoption in the sense of Mezzo can be undone: the take instruction revokes an adopter-adoptee relationship (after checking, at runtime, that this relationship exists). If desired, a member can leave a group forever. Obtaining simultaneous access to two or more members is a simple matter of using several take instructions. The fact that these objects are distinct is then checked at runtime.

For many practical purposes, adoption and abandon is a more flexible mechanism than nesting. One price to pay is the runtime cost of give and take, as well as the introduction of potential runtime failures. Another weakness of adoption and abandon is that, in its simplest form, it does not support transferring all adoptees of adopter \( \gamma_1 \) to adopter \( \gamma_2 \) in constant time (this was discussed at the end of §7), whereas in the case of purely static regions, one can imagine a ghost instruction that merges one region into another.

One can view adoption and abandon as a region discipline where regions have a runtime representation. However, in contrast with the traditional notions of runtime regions, whose aim is to support mass deallocation, the runtime data structures maintained by adoption and abandon serve to keep track of certain ownership relationships at runtime. In this sense, adoption and abandon seems related to the variants of Ownership Types where ownership can be tested at runtime [Clarke et al. 2013, §4.3].

9.7. Concurrency and locks

The unique ownership discipline imposed by separation logic provides data race freedom by default. On top of this basic discipline, it is necessary to offer a mechanism by which several threads can synchronize and exchange permissions. In Mezzo, following concurrent separation logic [O’Hearn 2007] and its successors [Gotsman et al. 2007; Hobor et al. 2008; Buisse et al. 2011], this role is played by locks. Other mechanisms, such as communication channels, can be implemented in Mezzo on top of locks. It is worth noting that, in these logics and in Mezzo, it is possible for an object to be protected by different locks, or not protected at all (i.e., owned by a single thread), at different moments in its lifetime. Indeed, the only requirement that must be obeyed is that each lock have a fixed invariant.

Like second-order separation logic, which forms the core of CaReSL [Turon et al. 2013], Mezzo supports quantification over permissions. This is exploited in the specification of locks: the type \( \text{lock } p \) is parameterized over a permission \( p \). Our specification
of locks is very close to the one found in iCAP [Svendsen and Birkedal 2014], an extension of higher-order separation logic. In Mezzo, of course, locks must be axiomatized, because Mezzo rejects racy programs. In iCAP, in contrast, locks can be implemented. iCAP supports reasoning about racy programs, under the assumption of a sequentially consistent memory model.

Our duplicable permissions are analogous to the necessary assertions of CaReSL. Our higher-order function hide (§1), which encodes a typical usage pattern of a lock, is essentially identical to Turon et al.’s mkSync [2013, §3.2].

The introduction of locks into Mezzo changes the meaning of Mezzo’s function type in quite a radical way. In the absence of locks, a function that modifies a piece of mutable state must request a suitable permission (and, usually, returns this permission). As a result, every side effect performed by a function must be advertised in its type. In the presence of locks, however, this is no longer the case. As illustrated by the definition of hide at the beginning of this paper (Fig. 3), a closure of type () -> () may capture the address l of a lock. By acquiring this lock, it obtains a permission (the lock invariant), which may allow it to perform a side effect. We refer to this feature – the fact that not every side effect is advertised in a function type – as hidden state. It is a good feature in that it promotes certain kinds of modularity, and a bad one in that it makes reasoning about programs more difficult and destroys some potential type-based compiler optimizations.

In a sequential setting, hidden state can be introduced via the anti-frame rule [Pottier 2008; Schwinghammer et al. 2010; Pottier 2013]. In a concurrent setting, this rule is unsound, so it is abandoned, and hidden state is typically introduced via locks. This is a good thing anyway, because the anti-frame rule seems quite difficult to explain, both to theorists and to end users. This complexity is perhaps due to the fact that the anti-frame rule is more ambitious, in a sense, than the rules that govern locks in Mezzo. In the case of locks, re-entrancy is ruled out via a runtime mechanism (which, unfortunately, may give rise to deadlocks); whereas in the case of the anti-frame rule, re-entrancy is ruled out via a purely static criterion (so there is no runtime cost and no risk of deadlock).

The idea of using a type discipline to enforce the correct usage of locks can be traced back to Flanagan and Abadi [1999]. There, every lock receives a type-level name, and every reference cell receives a type that mentions this name. The code is type-checked under a “current permission”, which is the set of the currently-held locks. This discipline can be simulated in Mezzo (if desired) by letting a lock protect a region (in the sense of §9.6). Thus, by acquiring the lock, one obtains a permission for the region, which in turn allows one to access the region inhabitants. If one prefers to use adoption and abandon rather than a static region, one can let a lock protect an adopter. Acquiring the lock yields a permission for the adopter, which (via a take instruction) gives access to the adoptees.

Boyapati, Lee and Rinard [2002] present an approach to safe locking that is based on Ownership Types. As in Flanagan and Abadi’s work, the system keeps track of which lock protects which object. It is quite expressive: it recognizes that synchronization is unnecessary when the object is immutable or is accessible to a single thread. The ownership of a unique object can be exchanged between two threads.

Chalice [Leino and Müller 2009; Leino et al. 2010] reasons about locks in a manner that seems roughly similar to concurrent separation logic. A monitor (an object that also serves as a lock) is equipped with an invariant, that is, a permission that is gained by acquiring the lock and is given up when releasing the lock.

Fractional permissions [Boyland 2003; Bornat et al. 2005; Boyland 2010] are supported by several tools, including VeriFast [Jacobs and Piessens 2008] and Chalice, which strives to hide them from the user, when possible [Heule et al. 2013]. For
greater simplicity, Mezzo does not have fractional permissions; it distinguishes only between immutable and mutable data. In Mezzo, a mutable data structure can become immutable and shareable, but not the other way around. Furthermore, in Mezzo, one cannot create a temporary read-only view of a mutable data structure. Extending Mezzo with fractional permissions should allow removing these restrictions; we have not studied this extension. Regions, as in Cyclone [Swamy et al. 2006] and Rust [The Mozilla foundation 2014], offer another way of creating temporary read-only views, which does not require accounting.

Gordon et al. [Gordon et al. 2012] ensure data-race freedom in a simple extension of C#. Their system, which is descended in part from Tschantz and Ernst’s Javari [2005], qualifies types with permissions in the set immutable, isolated, writable, or readable. The first two roughly correspond to our immutable and mutable modes, whereas the last two have no Mezzo analogue. Shared (writable) references allow legacy sequential code to be considered well-typed. Quite remarkably, the system requires neither permission accounting nor an alias analysis. This makes the system very simple, but comes at a cost in expressiveness: mutable global variables, as well as shared objects protected by locks, are disallowed.

Several of the works cited above [Flanagan and Abadi 1999; Boyapati et al. 2002; Leino et al. 2010] enforce deadlock freedom. They impose a total order on locks, which in the more expressive systems can be determined (and possibly evolve) at runtime. These systems keep track of which locks are held and, when another lock is acquired, check that the order is respected. In Mezzo, for the sake of simplicity, this is not done; Mezzo does not guarantee deadlock freedom. It should be possible in principle to extend Mezzo with these ideas; we have not studied this extension. It is tempting to view the information that a certain lock is held as a permission. (Indeed, in Mezzo today, 1 @ locked is a permission. It means that the lock 1 is held and acts as a permission to release it.) However, one must be careful. In a system that aims to guarantee deadlock freedom, the information that a lock 1 is held does not only allow us to release 1; it also forbids us from acquiring a lock that is not provably ordered above 1. Thus, this permission is not affine: it is not sound to forget (either forever or, via the frame rule, temporarily) that a lock is held. In Leino et al.’s terminology [2010], it should be thought of not as a permission, but as an obligation, or a “negative credit”. In an extension of Mezzo with these concepts, one would perhaps remove the rule that “every permission is affine”, and introduce an explicit predicate affine p so as to be able to quantify over affine types and permissions, when desired. The type-checker would ensure that the operations of dropping a permission, framing out a permission, and hiding a permission (by protecting it with a lock) are applied only to affine permissions.

9.8. Proof techniques and modularity

The soundness of concurrent separation logic was first established by Brookes [2004], based on a denotational semantics that interprets programs as sets of traces. Vafeiadis [2011] proposes a different proof, based on a standard (i.e., uninstrumented) operational semantics. He defines the meaning of a judgement in terms of the operational semantics; then, he proceeds to prove that every deduction rule is sound with respect to this interpretation. Several more recent and more advanced logics, such as iCAP [Svendsen and Birkedal 2014], follow a similar route, where the meaning of a judgement is defined in terms of a “model” whose construction (on top of a standard operational semantics) is quite elaborate but uses by-now well-understood techniques. Our proof of soundness for Mezzo is also based on a standard operational semantics, but follows Wright and Felleisen’s “syntactic” approach [1994]. Instead of defining up front the meaning of a judgement, we view the set of deduction rules as a definition
of the judgement, and proceed to prove that this judgement is an invariant (i.e., it is preserved by reduction) and is safe (i.e., every well-typed configuration is acceptable).

There has been some debate as to which approach is preferable. Vafeiadis [2011], for instance, writes that the syntactic approach is “rather fragile”, because “if a new construct were to be added to the language, the soundness of the existing rules would have to be reproved”. This criticism is valid. However, we believe that a similar criticism can be made of the “semantic” approach, where the construction of the model and the interpretation of triples can be viewed as a rather elaborate and monolithic summary of all the features that the logic supports. The introduction of a single new feature, such as locks, the higher-order frame rule [Schwinghammer et al. 2009], or the anti-frame rule [Schwinghammer et al. 2010], can require a modification of the model and/or of the interpretation of triples, which in turn requires checking every proof again.

In the end, we believe neither approach is fundamentally superior to the other. Furthermore, we argue, if one relies on a mechanized proof assistant, the need to re-check existing proofs is not really a problem. What matters is that the proof terms, or proof scripts, be written in such a way that they remain valid, or are easy to fix, after some definitions are altered. This pragmatic understanding of robustness has been relatively under-studied, it seems.

We have emphasized the modular organization of the meta-theory of Mezzo. When one extends the kernel in a new direction (references; locks), one must of course extend existing inductive definitions with new cases and extend the state with new components. However, one does not need to alter existing rules, or to alter the statements of the main type soundness lemmas. Of course, one sometimes must add new cases to existing proofs—only sometimes, though, as it is often possible to express an Ltac “recipe” that magically takes care of the new cases [Chlipala 2013, chapter 16].

The manner in which this modularity is reflected in our Coq formalization reveals pragmatic compromises. We use monolithic inductive types. Delaware et al. [2013] have shown how to break inductive definitions into fragments that can be modularly combined. This involves a certain notational and conceptual overhead, as well as a possible loss of flexibility, so we have not followed this route. A moderate use of type classes allows us to access or update one component of the state without knowing what other components might exist. A similar feature is one of the key strengths of the MSOS notation [Mosses 2004]. As often as possible, we write statements that concern just one component of the state, and in the few occasions where it seems necessary to explicitly work with all of them at once, we strive to write Ltac code in a style that is insensitive to the number and nature of these components. It has been our experience that each extension (references; locks) required very few undesirable amendments to the existing code base.

Although the formalization of Mezzo was carried out independently, and in part grew out of earlier work by Pottier [2013], it is in several ways closely related to the Views framework [Dinsdale-Young et al. 2013]. In both cases, an abstract calculus is equipped with a notion of machine state; a commutative semigroup of views, or resources; and a projection, or correspondence, between the two levels. This abstract system is proven sound, and is later instantiated and extended to accommodate features such as references, locks, and more. The Views framework is meant to form a simple, abstract, re-usable kernel on top of which more elaborate logics, such as CAP [Dinsdale-Young et al. 2010], can be defined and proved sound.

From Pottier’s previous work [2013], we borrow some ideas, such as the axiomatization of monotonic separation algebras. It is closely related to Dockins et al.’s separation algebras [2009] and to Views. It differs in that it has explicit provision for reasoning about duplicability (via the function “core”, which maps $R$ to $\hat{R}$) and about interference (via the “rely” relation $R_1 \prec R_2$). Compared with Pottier’s previous mechanized
proof [2013], the absence of regions and the absence of an instrumented operational semantics represent significant technical simplifications.

10. CONCLUSION

In our tutorial introduction to Mezzo (§1, §2) we have strived to illustrate how a hypothetical Mezzo programmer thinks and works. We believe that Mezzo makes it relatively easy to work with with list- or tree-shaped mutable data structures, with immutable data structures of arbitrary shape, and with (possibly higher-order) functions. We believe that Mezzo’s static discipline helps the programmer reason about the transfers of ownership that take place when a function is called or returns and when a lock is acquired or released. Thanks to this discipline, certain mistakes caused by undesired aliasing are ruled out, and data race freedom is guaranteed. We have also illustrated the difficulties that arise when one wishes to borrow a (non-duplicable) element from its container (§2.4) and when one wishes to build mutable data structures that involve arbitrary aliasing patterns (§2.5). One typically works around these difficulties by organizing objects in groups and by keeping track of just one permission for an entire group. This is done by using either nesting (a purely static mechanism, described in §9.6; not part of our formalization of Mezzo) or adoption and abandon (a dynamic mechanism; part of our formalization, and a contribution of this paper).

We have presented a modular formalization of Mezzo, organized as a kernel, on top of which sit three (almost) independent extensions. The kernel (§4) can be described as a concurrent call-by-value λ-calculus, equipped with an affine, polymorphic, value-dependent type-and-permission system. The extensions are:

— strong (i.e., affine, uniquely-owned) mutable references (§5);
— dynamically-allocated, shareable locks, which offer a form of hidden state (§6);
— adoption and abandon (§7).

This paper is accompanied with a Coq proof [Balabonski and Pottier 2014]. It is about fourteen thousand (non-blank, non-comment) lines of code. Out of this, a de Bruijn index library and a monotonic separation algebra library, both of which are reusable, occupy about 2Kloc each. The remaining 10Kloc are split between the kernel (roughly 4Kloc) and its three extensions (roughly 6Kloc). These are rough figures only, as the kernel and its extensions are not clearly separated in the final artifact.

REFERENCES


Adam Chlipala. 2013. Certified Programming and Dependent Types. MIT Press.


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.


The Design and Formalization of Mezzo, a Permission-Based Programming Language


Philip Wadler. 1990. Linear types can change the world! In Programming Concepts and Methods, M. Broy and C. Jones (Eds.). North Holland.


A. PROOF OF TYPE SOUNDNESS

We outline the main steps along the way that leads to the statement of type soundness (Theorem A.13).

A.1. Hygiene properties of well-kindness

Well-kindness is preserved by weakening (i.e., the insertion of a new, unused variable) and by substitution. Furthermore, the typing judgements respect well-kindness, in the sense that (if seeded with a well-kindred precondition) they hold of a well-kindred term and a well-kindred type. We omit these statements.
A.2. Hygiene properties of well-typedness

The typing judgement is preserved by weakening. (We omit this statement.) It is also preserved by kind-preserving substitution at kind value, type, and perm. (These are the kinds at which quantification is permitted.) Below, we state this lemma for the main typing judgement. There are analogous statements about the auxiliary judgements, namely the interpretation of permissions, subsumption, etc., which we omit.

**Lemma A.1 (Substitution).** Let $\kappa$ be value, type, or perm. Typing is preserved by the substitution of a syntactic element $u$ of kind $\kappa$ for a variable of kind $\kappa$.

$$
R; K, x: \kappa; P \vdash t: T \\
R; K; [u/x]P \vdash [u/x]t: [u/x]T
$$

Note that, when one replaces a variable of kind value with a value $v$, this value is not required to be well-typed. This is a natural consequence of the fact that our kind environments do not contain any type assumptions. This should be contrasted with the substitution lemma of (say) simply-typed $\lambda$-calculus. A more conventional lemma can be recovered by using the above lemma in conjunction with the rule cut (Fig. 21):

**Lemma A.2 (Substitution/Cut).** Typing is preserved by the substitution of a value $v$ of type $U$ for a variable $x$ that was assumed to have type $U$.

$$
\frac{R_1; K \vdash v @ U \\
R_2; K, x: value; x @ U \vdash t: T}{R_1 \star R_2; K; empty \vdash [v/x]t: T}
$$

A.3. Resources and well-typedness

The following three properties are established independently of one another. They state that the typing judgement “respects” the main three constituents of the monotonic separation algebra, namely composition of resources $\star$, core $\hat{\cdot}$, and rely $\triangleleft$. Again, we formulate each of these three statements only about the typing judgement; there are analogous statements about the other judgements.

The typing judgement is affine: it never hurts to have more resources than necessary.

**Lemma A.3 (Affinity).** Well-typedness is preserved by the addition of unnecessary resources.

$$
\frac{R_1; K \vdash t: T \\
R_1 \star R_2 \text{ok}}{R_1 \star R_2; K; P \vdash t: T}
$$

The syntactic notion of duplicable types and permissions, as defined by the metalevel predicate $\theta$ is duplicable (§4.4), is sound with respect to the semantic idea of a duplicable resource. The lemma states that if a duplicable permission is justified by some resource (say, $R$), then it is justified by some duplicable resource (in fact, it is justified by $\hat{R}$).

**Lemma A.4 (Duplication).** Duplicable permissions can be justified by duplicable resources.

$$
\frac{R; K \vdash P \quad R \text{ ok} \quad P \text{ is duplicable}}{R; K \vdash P}
$$

As an immediate corollary of the above lemma, if $R \text{ ok}$ and $P$ is duplicable hold, then $R; K \vdash P$ implies $R; K \vdash P \star P$. This shows that the subsumption rule Duplicate (Fig. 23) is sound.

The actions of a thread cannot cause an inactive thread to become ill-typed. In other words, well-typedness is stable in the face of permitted interference, as defined by the “rely” relation $\triangleleft$ (§4.1).
Lemma A.5 (Stability). Typing is preserved under an evolution of the resource along the relation $<$.

$$
R_1; K; P \vdash t : T \quad R_1 \text{ ok} \quad R_1 \prec R_2 \\
R_2; K; P \vdash t : T
$$

A.4. Classification and decomposition

We prove a classification lemma and a decomposition lemma for each type constructor. These lemmas extract information out of a canonical typing judgement. By way of example, we present the classification and decomposition lemmas for functions; similar lemmas must be stated for each of the other type constructors.

**Lemma A.6 (Classification).** Among the values, only $\lambda$-abstractions admit a function type.

$$
R; K \parallel v \in T \rightarrow U \\
\exists x, \exists t, v = \lambda x.t
$$

These statements must allow for a non-empty environment $K$. Indeed, as $\text{FORALLINTRO}$ (Fig. 21) is not restricted to values, we must be able to reason about “reduction under $\lambda$”, that is, reduction in a non-empty environment.

In light of this remark, the classification lemma may seem surprising: since $K$ can contain a binding of the form $x' : \text{value}$, couldn’t the value $v$ be a variable $x'$, in which case the conclusion would not hold? It turns out that the premise rules out this case: in a canonical type derivation, a variable cannot receive a function type.

**Lemma A.7 (Decomposition).** If $\lambda x.t$ has type $T \rightarrow U$, then $t$ has type $U$ under the precondition $x \in T$.

$$
R; K \parallel \lambda x.t \in T \rightarrow U \quad R \text{ ok} \\
R; K, x : \text{value}; x \in T \vdash t : U
$$

The above decomposition lemma is slightly stronger than one might expect in view of the typing rule $\text{FUNCTION}$ (Fig. 21). Indeed, the lemma does not mention the fact that the function body may need a duplicable permission $P$. We are able to establish this strong statement because the permission $P$, if there is one, can be hidden by application of Lemma A.4 and of the typing rule $\text{CUT}$.

A.5. Soundness of subsumption and canonicalization

Permission subsumption, which we have inductively defined (Fig. 23), is sound with respect to the semantic notion of entailment that arises out of the interpretation of permissions. All of the previous lemmas (except Lemma A.5, Stability, which is used only in the proof of Subject reduction) are used in this proof.

**Lemma A.8 (Soundness of Subsumption).** Permission subsumption is sound:

$$
K \vdash P \leq Q \quad R; K \parallel P \quad R \text{ ok} \\
R; K \parallel Q
$$

and so is subtyping:

$$
K \vdash T \leq U \quad R; K \parallel v \in T \quad R \text{ ok} \\
R; K \parallel v \in U
$$

This result immediately implies that an arbitrary type derivation for a value can be turned into one that does not use the subsumption rules $\text{SUBLEFT}$ and $\text{SUBRIGHT}$.
outside of a \( \lambda \)-abstraction. Furthermore, it is possible to eliminate every use of \textbf{EXISTS}\textbf{ELIM} outside of a \( \lambda \)-abstraction. This is done by substituting the concrete witness for the abstract variable, using the substitution lemma (Lemma A.1). In summary, an arbitrary derivation about a value \( v \), whose precondition is empty, can be turned into a canonical derivation:

**Lemma A.9 (Canonicalization).** If a value \( v \) admits the type \( T \) under an empty precondition, then there is a canonical derivation of this fact.

\[
\begin{align*}
R; K; \text{empty} & \vdash v : T & R \text{ ok} \\
\end{align*}
\]

Canonicalization is exploited just once, in the proof of subject reduction for a \( \beta \)-redex. There, the redex is of the form \((\lambda x.t) v\). A priori, we have an arbitrary type derivation for the value \( v \). But, in order to prove that the reduct \([v/x]t\) satisfies the desired typing judgement, we must apply Lemma A.2, which requires a canonical derivation for \( v \).

### A.6. Subject reduction and progress

The proof of the subject reduction lemma is by induction over (a measure of the height of) the type derivation and by induction over the reduction step. This requires the statement to be written under a suitable form. One reasonably readable form is as follows. (There is a more complex form, which allows for a non-empty environment \( K \), and explodes the hypothesis about \( t_1 \) into multiple hypotheses. We omit it.)

**Lemma A.10 (Subject reduction, preliminary form).** Let the configuration \( s_1 / t_1 \) have kind \( \text{term} \) under an empty environment. (Thus, the term \( t_1 \) is closed and represents a single thread.) Assume \( s_1 / t_1 \) reduces in one step to \( s_2 / t_2 \). Assume the machine state \( s_1 \) corresponds to the resource \( R_1 \star R'_1 \), where \( R_1 \) allows arguing that \( t_1 \) has type \( T \). Then, the machine state \( s_2 \) corresponds to some resource of the form \( R_2 \star R'_2 \), where \( R_2 \) allows arguing that \( t_2 \) has type \( T \); and the interference that has been imposed to the environment, from \( R'_1 \) to \( R'_2 \), is permitted.

\[
\begin{align*}
\begin{cases}
\begin{align*}
s_1 / t_1 & \longrightarrow s_2 / t_2 \\
R_1; \emptyset; \text{empty} & \vdash t_1 : T \\
\end{align*}
\end{cases}
\end{align*}
\]

\[
\exists R_2 R'_2 \begin{cases}
\begin{align*}
s_2 & \sim R_2 \star R'_2 \\
R_2; \emptyset; \text{empty} & \vdash t_2 : T \\
R'_1 & \triangleright R'_2
\end{align*}
\end{cases}
\]

This result allows deriving the following, much more compact corollary, which is phrased in terms of the typing judgement for configurations (Fig. 24):

**Lemma A.11 (Subject reduction).** Reduction preserves well-typedness.

\[
\begin{align*}
c_1 \longrightarrow c_2 & \vdash c_1 \\
\end{align*}
\]

A configuration \( s / t \) is deemed acceptable if and only if every thread in the thread soup \( t \) either:

— has finished and produced a value; or
— is waiting on a lock that is currently held; or
— is able (with respect to the machine state \( s \)) to take a step.

In other words, a configuration is acceptable if no thread has gone wrong.
The Design and Formalization of Mezzo, a Permission-Based Programming Language

The type soundness result states that well-typed programs do not go wrong. Note that the type system does not rule out deadlocks or livelocks; it is possible for a thread to wait indefinitely for a lock.

**Theorem A.13 (Type Soundness).** Let \( t \) be a well-typed source program; that is, assume \( \text{void;} \emptyset; \text{empty} \vdash t : T \). Then, by executing the configuration initial / \( t \), one can reach only acceptable configurations.

**B. Derived Rules**

None of the typing rules resemble the “axiom” rule of simply-typed \( \lambda \)-calculus, which states that \( x \) has type \( T \) under the assumption that \( x \) has type \( T \). Indeed, **Singleton** (Fig. 21) only allows proving that \( x \) has type \( \equiv x \). Fortunately, an axiom rule, **Value**, presented in Fig. 41, can be derived using **Singleton**, **Frame**, and subsumption.

We have defined \( \text{let } x = u \text{ in } t \) as sugar for \( (\lambda x.t)u \). Using the rules **Function**, **Application**, **Cut**, **Frame**, and subsumption, it is possible to derive a typing rule for this construct. The rule **LetFrame**, shown in Fig. 41, uses part of the precondition (namely \( P \)) to prove that the term \( t \) has type \( T \), while the rest (namely \( Q \)), together with the new hypothesis \( x@T \), is used to type-check the term \( u \).

Finally, using **LetFrame**, it is straightforward to obtain a new type-checking rule for function application, **NormalApp** (Fig. 41). This rule has two premises and splits the current permission between them. This is in contrast to **Application** (Fig. 21), which has only one premise, and requires the operator to be a value \( v \). Here, the operator can be a term \( t \), but an explicit sequencing construct must again be used.

**C. Encoding the Simply-Typed \( \lambda \)-Calculus**

Using the above derived rules, it is easy to encode the simply-typed \( \lambda \)-calculus into Core Mezzo. The encoding of terms is as follows:

\[
\begin{align*}
[x] &= x \\
[\lambda x.t] &= \lambda x.[t] \\
[t u] &= \text{let } x = [t] \text{ in } x [u]
\end{align*}
\]

Because the left-hand side of an application must be a value, an explicit sequencing construct is introduced.

The types of the simply-typed \( \lambda \)-calculus are given by the grammar \( T ::= \top | T \rightarrow T \). The encoding \( [T] \) of a type \( T \) is \( T \) itself. A type environment \( E \) of the simply-typed \( \lambda \)-calculus is encoded in two distinct ways: as a kind environment, \( [E] \), and as a per-
mission, $\langle E \rangle$. They are defined as follows:

$$
[x_1 : T_1, \ldots, x_n : T_n] = x_1 : \text{value}, \ldots, x_n : \text{value}
$$

$$
\langle x_1 : T_1, \ldots, x_n : T_n \rangle = x_1 @ T_1 * \ldots * x_n @ T_n
$$

Every $[T]$ is a duplicable type, and (as a result) every $\langle E \rangle$ is a duplicable permission. This property is necessary for the encoding to work: since the simply-typed $\lambda$-calculus is not an affine calculus, a variable can be used more than once.

The encoding is type-preserving:

**Lemma C.1 (Encoding).** If $E \vdash t : T$ holds in the simply-typed $\lambda$-calculus, then $\text{void;} [E]; \langle E \rangle \vdash \llbracket t \rrbracket : \llbracket T \rrbracket$ holds.