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 give a comprehensive tutorial overview of the language. Then, we present a modular formalization of Mezzo’s core type system, in the form of a concurrent λ-calculus, which we successively extend with references, locks, and adoption and abandon, a novel mechanism that marries Mezzo’s static ownership discipline with dynamic ownership tests. We prove that well-typed programs do not go wrong and are data-race free. Our definitions and proofs are machine-checked.

CCS Concepts:
- Theory of computation → Separation logic; Type structures; Operational semantics; Type theory; Software and its engineering → Imperative languages; Functional languages; Concurrent programming languages; Abstract data types; Polymorphism; Data types and structures; Recursion; Procedures, functions and subroutines; Syntax; Semantics;

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

ACM Reference Format:
D0I: http://dx.doi.org/10.1145/2837022

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. These tests do have costs, in terms of time, space,
1 open worref
2
3 val f (x: frozen int, y: frozen int) : int =
   get x + get y

4 5 6
7 val _ : int =
8 let r = new () in
9 set (r, 3);
10 f (r, r)

and robustness; in return, they offer far greater expressiveness than a simple-minded,
purely static discipline could hope to achieve.

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 two 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.

1.1. A few examples

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.

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 Γ ⊢ 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 the reference.
open

val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1
val () =
  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 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 \( \Gamma \vdash r : frozen int \).

At line 9, we apply the function \( f \) to the pair \((r, r)\). This function, defined at line 3, expects a pair \((x, y)\) and requires the right to use \( x \) and \( y \) at type \( frozen int \). (At this stage, the reader can understand the notation \( x : frozen int \) as a shorthand for “the argument is named \( x \)” and “this function requires the permission \( x @ frozen int \).” This notation is explained in detail in §3.) At the call site 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 \( f \). The call causes \( r \) to be read twice (through distinct aliases). This is permitted by the protocol.

A race. We now consider the tiny program in Fig. 2. This code exhibits a data race: two threads may concurrently access the reference \( r \), where one of the accesses is a write. In Mezzo, racy code is viewed as 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 (traditional) 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 nonduplicable permission. This restriction is made necessary by the fact that every function type is considered duplicable.) This is declared by the type annotation. Thus, at line 6, in conjunction with \( r @ ref int \), we have \( f @ (| r @ ref int) -> () \). This (duplicable) permission means that \( f \) is a function of no argument and no result (at runtime), which (at type-checking time) requires and returns the permission \( r @ ref int \).

To clarify this, let us 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 conven-
tion, \( f @ (| r @ \text{ref int}) \rightarrow () \) means that \( f \) requires and returns the permission \( r @ \text{ref int} \). When one wishes to indicate that a function requires some permission but does not return it, one must precede that permission with the keyword \text{consumes}.

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

This behavior should be contrasted with that of the earlier example. In Fig. 1, the permission \( r @ \text{frozen int} \), which \text{get} requires, is duplicable. We can therefore obtain two copies of it and justify the call \( f (r, r) \). We could also justify several concurrent calls to \text{get} \( r \).

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, hide (Fig. 3).

In Fig. 3, \( f \) is a parameter of hide. It has a visible side effect: it requires and returns a permission \( s \). When 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 \( f @ (\text{consumes} a | s) \rightarrow b \). At the end of line 7, after the call to \text{lock::new}, we have given up \( s \), which has been consumed by the call, and we have obtained \( l @ \text{lock} s \). 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 \( x @ a \). Nevertheless, the permissions \( f @ (\text{consumes} a | s) \rightarrow b \) and \( l @ \text{lock} s \) 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 \( l @ \text{lock} s \) 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 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.

```
open lock
val hide [a, b, s : perm] ( f : (consumes a | s) \rightarrow b | consumes s ) : (consumes a) \rightarrow b =
  let l : lock s = new () in
  fun (consumes x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

Fig. 3. The polymorphic higher-order function hide takes a function \( f \) of type \((\text{consumes} a | s) \rightarrow b \). This means that \( f \) needs access to a piece of state represented by the permission \( s \), 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 \text{type} (this is the default kind). The square brackets denote universal quantification.
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 `val f = hide f` before line 6. The type variables \( a \) and \( b \) in the type of `hide` are instantiated with \( (\)\), and the permission variable \( s \) is instantiated with \( r \@\ref\int \). This call consumes \( r \@\ref\int \) and produces \( f \@(\)\rightarrow(\)\), so the two `spawn` instructions are now type-correct. Indeed, the modified code is race-free.

1.2. A case for Mezzo

We believe that reasoning about unique ownership, or unique permission, is useful, and even necessary, for several reasons.

1. This allows the programming language designer to express and enforce protocols. As a result, several properties of the programming language can be proved, once and for all.
2. This allows the programming language user to express and enforce protocols. This helps to write secure, correct code, and to prove it.

The protocols imposed by the language designer restrict the use of the language’s primitive features, such as mutable state and locks. Examples of protocol descriptions may include: “accessing object \( x \) requires permission \( p \)”; “deallocating object \( x \) requires and consumes permission \( p \)”; “acquiring lock \( l \) produces permission \( p \)”; “releasing lock \( l \) requires and consumes permission \( p \)”; and so on. (Mezzo does not have manual memory deallocation; languages that do, and guarantee its safe use, include Cyclone [Swamy et al. 2006] and Rust [The Mozilla foundation 2014].) These protocols are designed so as to guarantee a small number of fundamental metatheoretic properties, such as memory safety (“only valid memory is ever accessed”) and data-race freedom (“no data race ever occurs”).

The protocols imposed by the user restrict the use of user-defined abstractions. Examples of protocols that a user may wish to enforce include: “a write-once reference must be fully initialized before it is used”; “a write-once reference may be initialized at most once”; “this continuation must be called at most once”; “either of these two continuations may be called, but not both; and it may be called at most once”; etc. Enforcing such protocols rules out a class of programming mistakes that cannot be detected by traditional type systems. Furthermore, ensuring that an abstraction uniquely owns its internal state is necessary in order to impose and maintain an invariant about this state. In other words, failure to ensure unique ownership of an abstraction’s internal state, also known as representation exposure [Detlefs et al. 1998], may lead to bugs and security flaws [Vitek and Bokowski 2001].

A huge number of tools exist that help detect bugs in software or prove their absence. Some have built-in support for reasoning about ownership or permissions: see, among others, jStar [Distefano and Parkinson 2008], VeriFast [Jacobs and Piessens 2008], VCC [Cohen et al. 2009], and Facebook Infer [Calcagno et al. 2015]. We believe that machine support for these concepts should ideally be built into the programming language and into its compiler, so that they may serve as a guide and as a safety net while the program is being designed and developed. Although several fairly large-scale experiments have been reported in the literature [Fähndrich et al. 2006; Gordon et al. 2012], as of today, no mainstream programming language imposes a static discipline based on these concepts.

If one attempted to design such a language, what would it be like? Could it be made simple, elegant, powerful? It is often said that Milner [1978] discovered a “sweet spot”, a striking compromise between simplicity and expressiveness, when he proposed the
foundations of ML. Is there a “sweet spot” out there with support for permissions and uniqueness? The Mezzo project explores these questions. While Mezzo cannot be the final answer, we believe that it is an interesting spot in the design space.

There are many different directions in which one might search for this sweet spot. Because we intend Mezzo to be a high-level language, we place most emphasis on simplicity, elegance, and expressiveness.

As the runtime model, we choose ML. More specifically, we re-use the OCaml garbage collector and runtime system: this is achieved by compiling Mezzo down to untyped OCaml. Even before we impose a static discipline, this guarantees that we need not worry about certain classes of runtime errors, including null pointer dereferences and accesses to deallocated memory. This choice has a certain cost in terms of efficiency (no manually managed memory; no stack-allocated objects; no unboxed or specialized data representations; no sub-word control of memory layout; etc.) but gives us greater freedom and greater hope of success in the design of a simple and powerful type and permission discipline. For instance, the fact that every value occupies one word of storage makes it easy to support polymorphism. Reasoning with algebraic data types (that is, tagged sums) is easier than working with disjunctions (that is, untagged sums), disequalities (such as \( x \neq \text{null} \)) and inductive predicates, as in the traditional separation-logic encoding of null-terminated lists. Reasoning about tail-recursive functions is easier than reasoning about \texttt{while} loops, as it obviates the need for thinking in terms of list (or tree) segments (§2.2).

In the design of the static type and permission discipline, we make several decisions with simplicity in mind. Let us mention a few salient points:

— We do not annotate types with owners: following separation logic, in Mezzo, “ownership is in the eye of the beholder”. That is, whoever is able to say “this is a hash table” in fact is the current owner of this hash table. This keeps types concise. More subtly, this means that a polymorphic function is polymorphic not only in the “shape” of its argument, but also in the “ownership regime” of its argument. For instance, the list length function, whose type is \([\alpha]\ \text{list}\ \alpha \rightarrow \text{int}\), can be applied either to a list of shareable elements or to a list of uniquely-owned elements, and does not care “who” owns the elements. In fact, the owner of the list elements must be the caller of \texttt{length}, and \texttt{length} itself becomes their owner while it is active: permission transfer is one of the key mechanisms that allows us to get away without naming owners.

— We draw a distinction between immutable, shareable data, on the one hand, and mutable, uniquely-owned data, on the other hand (and favor the use of the former), but do not attempt to incorporate more sophisticated ideas, such as per-field permissions, temporary immutable views of mutable data, or fractional permissions, as we wish to assess how far and how well one can fare without them.

— To deal with the situations where our static type and permission discipline is too coarse, we provide an “escape hatch”, known as adoption and abandon. This mechanism replaces static proof obligations, which would require complex compile time arguments, with dynamic ownership tests. Whereas Mezzo’s basic metaphor is “a thread owns an object”, the metaphor offered by adoption and abandon is “an object (the adopter) owns an object (the adoptee)”. The operations of “adoption” and “abandon”, known more simply as \texttt{give} and \texttt{take}, move from one metaphor to the other, thus marry them in a manner that we believe is quite natural and easy to understand.

Simplicity is not our sole guideline: the core features of the static discipline are chosen with expressiveness in mind as well.
Reasoning about unique ownership and state change requires distinguishing between duplicable and affine types and allowing strong updates.

Distinguishing between a value (say, a pointer) and the right to use this value (say, to dereference this pointer) requires introducing a notion of permission and letting permissions depend on values. For instance, the permission \( r \oplus \text{ref int} \), which means "\( r \) is a reference to an integer", mentions the variable \( r \). The permission \( l \oplus \text{lock}(r \oplus \text{ref int}) \), which means "the lock \( l \) protects the integer reference \( r \)", refers to the variables \( l \) and \( r \).

Writing modular code requires polymorphism and type abstraction.

None of these concepts is new. Our experience with Mezzo confirms (if necessary) that their combination is very powerful. It enables us to express the protocols that govern the use of concurrency primitives (locks (§1.1, §2.1), channels, . . .), adoption and abandon (§2.5, §2.6), nesting (§2.6), and so on, simply by ascribing an appropriate type to each operation.

One key question that one faces in the design of a permission-based programming language is: should the permission discipline come on top of a traditional type system? Or, on the contrary, should the language enjoy a single, unified type-and-permission discipline? The former approach allows type-checking and permission-checking to be carried out in two separate phases; this sounds technically attractive. The latter approach offers greater expressiveness (for instance, it supports gradual initialization of uniquely-owned objects), potentially greater conciseness (because types and permissions express layout and ownership at the same time), and a rather different mindset. In Mezzo, we explore the latter approach, which appears to have received relatively little attention in the literature.

1.3. 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. We give two variants of this example. One variant exploits adoption and abandon, a mechanism that defers some of the ownership tests to runtime. Another variant exploits nesting, a mechanism that serves the same purpose and does not require any runtime tests, but has more limited expressiveness.

The surface language that we expose to the user differs slightly from the core 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 (§2.6). 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 (§3.7), locks (§3.8), and adoption and abandon (§3.8). 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 (§3), a discussion of the implementation of Mezzo (§4), and a review of the related work (§5).

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.

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 well as the permission r @ writable, as affine (i.e., nonduplicable). 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)

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

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

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

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

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

Fig. 4. Implementation of write-once references
and a regular field, also called contents\(^1\), which has type \((a \mid \text{duplicable } a)\). This is a type of the form \(t \mid p\): indeed, \(a\) is a type, while \(\text{duplicable } a\) 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 \(\text{duplicable } a\) as part of the definition of the type \(\text{frozen } a\)? 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 \(\text{duplicable } a\) can 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.

The definition of \(\text{frozen}\) does not contain the keyword \(\text{mutable}\), so a block of type \(\text{frozen } a\) is immutable. Thus, it is safe to share read access to such a block. Furthermore, because we have imposed the constraint \(\text{duplicable } a\), 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 \(\text{frozen } a\) is duplicable as a whole. This means that a write-once reference can be shared after it has been initialized.

In the absence of \(\text{duplicable } a\) on line 5, the type parameter \(a\) would conservatively be considered affine (i.e., nonduplicable). Thus, the type \(\text{frozen } a\) would describe a shareable block containing a pointer to a nonshareable data structure of type \(a\). The type \(\text{frozen } a\) as a whole would be considered nonduplicable.

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

The code of \(\text{set}\) is in two steps. First, the value \(x\) is written to the field \(r\.\text{contents}\) (line 12). After this update, the memory block at address \(r\) is described by the permission \(r @ \text{Writable \{ contents: a \}}\). This is a structural permission: it describes the tag and the fields of the memory block. This permission is not an unfolding of \(\text{writable}\); neither is it an unfolding of \(\text{frozen } a\). The memory block is in an intermediate state.

Then, the tag of \(r\) is changed from \(\text{Writable}\) to \(\text{Frozen}\): this is a tag update (line 13). This particular tag update instruction is ghost code: it has no runtime effect, because both \(\text{Writable}\) and \(\text{Frozen}\) 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 \(r @ \text{Frozen \{ contents: a \}}\).

This permission can be combined with the permission \(\text{duplicable } a\) (which exists at this point, because \(\text{set}\) requires this permission from its caller) so as to yield \(r @ \text{Frozen \{ contents: (a | \text{duplicable } a) \}}\). This is the right-hand side of the definition of the type \(\text{frozen } a\). By folding it, one obtains \(r @ \text{frozen } a\). Thus, the permissions available at the end of the function \(\text{set}\) 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, hence uniquely owned; and (ii) the old and new types must have the same number of fields. This instruction is compiled down to either an update of the tag field,

\(^1\)Mezzo allows two user-defined types to have fields that go by the same name.
or nothing at all, as is the case above. (The distinction between these two cases depends on the mapping of tags to numbers. A programmer who does not wish to depend on this low-level detail may conservatively assume that a tag update is not a no-op.)

An interface for \texttt{woref}. Mezzo currently offers a simple notion of module, or unit. Each module has an implementation file (whose extension is \texttt{.mz}) and an interface file (whose extension is \texttt{.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 \texttt{woref} is shown in Fig. 5.

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

The type \texttt{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 \texttt{frozen \ a} is a duplicable type is published (line 3). In the absence of this declaration, \texttt{frozen \ a} would by default be regarded affine, so that sharing access to an initialized write-once reference would not be permitted. This \texttt{fact} declaration is implicitly universally quantified in the type variable \texttt{a}. One can think of it as a universally quantified permission, \texttt{[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 \texttt{new}, \texttt{get}, and \texttt{set}, without exposing their implementation. In the type of \texttt{set}, the first argument \texttt{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 \texttt{r: \ t} binds the variable \texttt{r} and at the same time requests the permission \texttt{r @ \ t}. In contrast, in the permission \texttt{r @ \ t}, the variable \texttt{r} occurs free. The second argument of \texttt{set}, \texttt{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 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

```plaintext
1 abstract writable
2 abstract frozen a
3 fact duplicable (frozen a)
4 val new: () -> writable
5 val set: [a] (consumes r: writable, x: a | duplicable a)
6 -> (| r @ frozen a)
7 val get: [a] frozen a -> a
```

Fig. 5. Interface of write-once references
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
append, also shown in Fig. 6, which is the most natural definition.

The type of 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 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
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 Nil or Cons. This
justifies the 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. This is a 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 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 the structural permission
xs @ Cons { head: a; tail: list a }. This permission 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:

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

The first conjunct describes just the memory block at address xs. It indicates that
this block is tagged Cons, that its head field contains the value head, and that its tail
field contains the value tail. The types =head and =tail are singleton types [Smith
et al. 2000]: each of them is inhabited by just one value. In Mezzo and from here on in
the paper, we write xs @ Cons { head = head; tail = tail } as syntactic sugar for
xs @ Cons { head: =head; tail: =tail }.

In the following, the permission xs @ Cons { head = head; tail = tail } 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
have permission to use it at type a. The last conjunct describes just the value 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:

```plaintext
let ws = append (tail, ys) in
let zs = Cons { head = head; tail = ws } in
```

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 ->
    ys
  | 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 ->
    ys
  | Cons ->
    let dst = Cell { head = xs.head; tail = () } in
    appendAux (dst, xs.tail, ys);
    dst
end

Fig. 7. List concatenation in tail-recursive style
zs

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

```
head @ a *
ws @ list a
```

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

The memory allocation expression `Cons { head = head; 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:

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

At this point, since `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 @ Cons { head: a; tail: list a }
```

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

```
zs @ list a
```

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

**When is a list duplicable?** It is natural to ask: what is the status of the permission `xs @ 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 `list t` must not be duplicable either. In short, the fact that describes lists is:

```
fact duplicable a => duplicable (list a)
```

This fact is inferred by the type-checker by inspection of the definition of the type list. If one wished to export list as an abstract data type, this fact could be explicitly written down by the programmer in the interface of the 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 \(xs \odot list a\) and \(ys \odot list a\)? Could it, for instance, not consume the latter permission?

In order to answer this question, let us attempt to change the type of append to \([a] (consumes xs: list a, ys: list a) \rightarrow list a\), where the \textit{consumes} keyword bears on \(xs\) only. Recall that, by convention, the absence of the \textit{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:

\[
[a] (consumes xs: list a, consumes ys: list a) \\
\rightarrow (list a \mid ys @ list a)
\]

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 \textit{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 \texttt{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 \(xs @ list (ref int) \ast ys @ list (ref int)\) 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 \(xs @ list \texttt{int} \ast ys @ list \texttt{int}\)? There is no danger in sharing an integer value: the type \texttt{int} is duplicable. It would be a shame to lose the permissions \(xs @ list \texttt{int}\) and \(ys @ list \texttt{int}\). 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 \(xs @ list \texttt{int} \ast ys @ list \texttt{int} \ast zs @ list \texttt{int}\): 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 \((consumes t) \rightarrow u\) and \(t \rightarrow u\) 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 (\texttt{FRAME}\texttt{SUB} and \texttt{DUPLICATE}, Fig. ??).

As a corollary, the universal type \([a] (consumes \{list a, list a\}) \rightarrow list a\), which is the type of append in Fig. 6, is strictly more general than the type \([a] \{list a, list a \mid \texttt{duplicable} a\} \rightarrow list a\), where the \textit{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 could work around this problem by performing concatenation in two passes: that is, in OCaml, by composing List.rev and List.rev_append.

If instead one insists on performing concatenation in one pass and in constant stack space, then one must write append in destination-passing style [Larus 1989]. Roughly speaking, the list xs must be 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.

In ML, unfortunately, this style requires breaking the type discipline. To wit, the authors of the OCaml library “Batteries included” [2014] implement concatenation (and other operations on lists) in this style by using an unsafe type cast. There are two (related) reasons why destination-passing style cannot be well-typed in ML. 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 in Mezzo a tail-recursive version of append, in destination-passing style. The code appears in Fig. 7. We recall that, even though this approach uses mutation internally, the goal is to concatenate two immutable lists so as to obtain an immutable list.

The reviewers pointed out that, in the presence of a generational garbage collector, updating a mutable field is significantly more costly than a single write instruction. As a result, append in destination-passing style may well be slower than the composition of List.rev and List.rev_append. Nevertheless, we believe that it is a good example of the power of Mezzo's type discipline. In particular, the manner in which Mezzo allows traversing lists without explicitly reasoning about list segments applies to other data structures as well (e.g., mutable lists; mutable trees).

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 turned by a tag update instruction (line 12) into an immutable block whose tag is Cons. (As in §2.1, this instruction has no runtime effect, because Cell and Cons are both represented by the number 1 at runtime. This is the reason why we have declared the apparently useless data constructor Dummy on line 2. This is not essential, though: in the absence of Dummy, the tag update instruction would have an actual runtime effect, and appendAux would still be tail-recursive.)

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). We explain below why this code is well-typed.

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 [2005a], 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 an inductive auxiliary lemma? 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 xs is \( \text{xs} @ \text{Cons} \{ \text{head}: \text{a}; \text{tail}: \text{list a} \} \). As in the earlier version of append (Fig. 6), the type-checker automatically decomposes it into a conjunction. Here, this requires introducing fresh auxiliary 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:

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

On line 14, we read \( \text{xs}\.\text{head} \). 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:

\[
\text{dst}' @ \text{Cell} \{ \text{head} = \text{head}; \text{tail}: () \}
\]

Although this does not play a key role here, it is worth noting that these permissions imply that the fields \( \text{xs}\.\text{head} \) and \( \text{dst}'\.\text{head} \) contain the same value, namely head. Besides, we have one (affine) permission for this value, \( \text{head} @ \text{a} \). So, the type-checker “knows” that \( \text{xs}\.\text{head} \) and \( \text{dst}'\.\text{head} \) are interchangeable, and that either of them (but not both in parallel) can be used as a value of type \( \text{a} \). Thanks to this precise knowledge, we do not need a “borrowing” convention [Naden et al. 2012] so as to decide which of \( \text{xs}\.\text{head} \) or \( \text{dst}'\.\text{head} \) has type \( \text{a} \). 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 the cell dst. Before the assignment, we have \( \text{dst} @ \text{Cell} \{ \text{head}: \text{a}; \text{tail}: () \} \). After the assignment \( \text{dst}.\text{tail} <- \text{dst}' \), we have \( \text{dst} @ \text{Cell} \{ \text{head}: \text{a}; \text{tail} = \text{dst}' \} \). After the tag update instruction \( \text{tag of } \text{dst} <- \text{Cons} \), we have we have \( \text{dst} @ \text{Cons} \{ \text{head}: \text{a}; \text{tail} = \text{dst}' \} \). It may be worth stressing that the last two structural permissions can be folded neither to \( \text{dst} @ \text{cell a nor to dst} @ \text{list a} \). There is no obligation for a structural permission to coincide at all times with the unfolding of some algebraic data type. A structural permission that is not an unfolding of some algebraic data type typically represents an intermediate state in a sequence of transitions.

At the beginning of line 17, just before the recursive call, the current permission is:

\[
\text{dst} @ \text{Cons} \{ \text{head}: \text{a}; \text{tail} = \text{dst}' \} *
\]
xs @ Cons { head = head; tail = tail } *
head @ a *
tail @ list a *
y @ list a *
dst' @ Cell { head = head; tail = () }

The call consumes the last four permissions and produces a new permission for dst'. Immediately after the call, the current permission is thus:

dst @ Cons { head = a; tail = dst' } *
x @ Cons { head = head; tail = tail } *
dst' @ list a

We have reached the end of the code. There remains to verify that the postcondition of appendAux is satisfied. By combining the first and last permissions above, the type-checker obtains dst @ Cons { head: a; tail: list a }. At this point, the types of the head and tail fields match the definition of the type list a (Fig. 6, line 3), so this permission can be folded back to dst @ list a. Thus, the postcondition is indeed satisfied: dst is now a valid list.

The fact that the structural permission dst @ Cons { ... } 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. 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 Chaguéraud [Chaguéraud 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. Here, we use traditional ML references, which are allocated with newref, assigned with :=, and dereferenced with !.

The function new creates a new stack. The function push inserts a list of elements into an existing stack. It relies on list concatenation (§2.2). The higher-order function work abstracts a typical pattern of use of a stack as a work list: as long as the stack is nonempty, 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 s is the stack; the parameter f is a user-provided

[2]This function is named newref, instead of ref in ML. Indeed, the type of references is called ref already, and Mezzo places types and values in a single namespace.
alias stack a = ref (list a)

val new [a] (consumes xs: list a) : stack a =
  newref xs

val push [a] (consumes xs: list a, s: stack a) : () =
  s := append (xs, !s)

val rec work [a, p : perm] (s: stack a,
  f: (consumes a | s @ stack a * p) -> ()
  | p) : () =
  match !s with
  | Cons { head; tail } ->
    s := tail;
    f head;
    work (s, f)
  | Nil ->
    ()
  end

Fig. 8. A minimal implementation of stacks, with a higher-order iteration function

function that is in charge of processing one element. This function has access to the permission \( s @ stack a \), which means that it is allowed to update the stack. The code is polymorphic in the type \( a \) of the elements. It is also polymorphic in a permission \( p \) that is threaded through the whole computation: if \( f \) requires and preserves \( p \), then \( work \) also requires and preserves \( p \). One can think of the conjunction \( s @ stack a * p \) as the loop invariant. The pattern of abstracting over a permission \( p \) 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 \( xs @ list t \), one must provide a permission \( x @ t \) for every element \( x \) of the list \( xs \).

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

If the type \( t \) is duplicable, then the permission \( x @ t \) does not have an ownership reading. One can duplicate this permission, give away one copy to the container when \( x \) is inserted into it, and keep one copy so that \( x \) 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 \( \text{find} \), which scans a list \( xs \) and returns the first element \( x \) (if there is one) that satisfies a user-provided predicate \( \text{ok} \). Transliti-
Generating the type of this function from ML to Mezzo, one might hope that this function admits the following type:

\[
\text{val find: } [a] (xs: \text{list} \ a, \ ok: a \rightarrow \text{bool}) \rightarrow \text{option} \ a
\]

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

One can give \(\text{find}\) the following type, where the list is consumed:

\[
\text{val find: } [a] (\text{consumes} \ xs: \text{list} \ a, \ ok: a \rightarrow \text{bool}) \rightarrow \text{option} \ a
\]

Naturally, in most situations, this type is too restrictive. We do not expect to lose the permission to use the entire list after just one call to \(\text{find}\).

Another tentative solution is to give \(\text{find}\) the following type, where the type parameter \(a\) is required to be duplicable:

\[
\text{val find: } [a] (xs: \text{list} \ a, \ ok: a \rightarrow \text{bool} | \text{duplicable} \ a) \rightarrow \text{option} \ a
\]

Naturally, this does not solve the problem. This means that \(\text{find}\) is supported only in the easy case where the elements are shareable. Certainly, 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 indirect style.** A simple approach is to give up control. Instead of asking \(\text{find}\) to return the desired element, we provide \(\text{find}\) with a function \(f\) that describes what we want to do with this element. The signature of \(\text{find}\) thus becomes:

\[
\text{val find: } [a] (xs: \text{list} \ a, \ ok: a \rightarrow \text{bool}, \ f: a \rightarrow ()) \rightarrow ()
\]

Recall that, in Mezzo, a function argument that is *not* annotated with the keyword \text{consumes} is preserved: that is, the function requires and returns a permission for this argument. Thus, this version of \(\text{find}\) preserves \(xs \@ \text{list} \ a\). The function \(f\), which the user supplies, preserves \(x \@ a\), where \(x\) is a list element. 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!

A moment’s thought reveals that the above signature for \(\text{find}\) is not as simple as it could be. The user is asked to provide \textit{two} functions, \(ok\) and \(f\). The function \(ok\) is supposed to recognize the desired element, while the function \(f\) is supposed to do something with it. Without loss of generality, we may combine these two functions into one,
val rec exists [a, p : perm] (xs: list a, ok: (a | p) -> bool | p): bool = match xs with Nil -> False | Cons -> ok xs.head || exists (xs.tail, ok) end

Fig. 9. Definition of list::exists

whose type is a -> bool. This function is expected to do something with the element, if desired, and return a Boolean result that indicates whether the search should stop or continue. In addition to this, we may as well change the result type of \texttt{find} to bool, so as to report whether the search was stopped or went all the way to the end of the list. The signature becomes:

\[
\text{val exists: [a] (xs: list a, ok: a -> bool) -> bool}
\]

We have changed the name of the function from \texttt{find} to \texttt{exists}, because it is now identical to the function known as \texttt{List.exists} in OCaml's standard library.

One soon finds out that this type is not expressive enough, as it does not provide any permission to ok beside \(x @ a\), where \(x\) is the list element that \(ok\) receives as an argument. This means that \(ok\) cannot perform any side effect, except possibly on \(x\). In order to relax this restriction, one must parameterize \texttt{exists} over a permission \(p\), which is transmitted to (and preserved by) \(ok\). This is a typical idiom for higher-order functions in Mezzo, which already appeared in the work function from Fig. 8.

\[
\text{val exists: [a, p: perm] (xs: list a, ok: (a | p) -> bool | p): bool}
\]

The definition of this function is shown in Fig. 9.

This approach works, to some extent, but is awkward. Working with a higher-order function is unnatural and rigid: elements must be borrowed from the container and returned to the container in a syntactically well-parenthesized manner; and one can borrow at most one element at a time. Furthermore, this style is verbose, especially in light of the fact that, in Mezzo, anonymous functions must be explicitly type-annotated.

In short, there is a reason why OCaml's standard library offers two distinct functions \texttt{find} and \texttt{exists}. Here, in an attempt to express \texttt{find} in Mezzo, we have ended up with \texttt{exists}, a higher-order function. We still do not have a satisfactory way of expressing \texttt{find} as a first-order function.

A \textit{solution in direct style.} The root of the problem lies in the fact that the permissions \(xs @ list a\) and \(x @ a\) cannot coexist. Thus, the function \texttt{find}, if written in a standard style, must consume \(xs @ list a\) 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 \(xs @ list a\).
data result (p: perm) a =
| NotFound { | p }
| Found { found: a }

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) : result
= match xs with
  | Nil -> NotFound
  | Cons { head; tail } ->
    if pred head then begin
      let w (|
        consumes (head @ a * tail @ list a))
        : (| xs @ list a) = () in
      Found { found = (head, w) }
    end
  else begin
    let r = find (tail, pred) in
    match r with
      | NotFound -> r
      | Found { found = (x, w) } ->
        let flex guess: perm in
        pack w @ wand (x @ a) (xs @ list a)
        witness (head @ a * guess);
        r
    end
  end

Fig. 10. Borrowing an element from a container in direct style

Fig. 10 shows a version of find that follows this idea. The function find requires the permission xs @ list a, which it consumes (line 13). If no suitable element exists in the list, then it returns a unit value, together with the permission xs @ list a (line 15). If one exists, then it returns a focused element (line 16). This alternative is expressed on line 14 via the algebraic data type result. According to the definition of this type (lines 1–3), an object of type result p a either is tagged NotFound, has zero fields, and carries the permission p; or is tagged Found and has one field of type a.

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 10–11) is a pair of an element x, which has type a, and a function w, a “magic wand” 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 \(xs\). A magic wand is affine: it can be used just once.

Mezzo does not currently have magic wand as a primitive notion. Instead, we define a magic wand (lines 5–8) as a (runtime) function of no argument and no result, which consumes a permission \(pre\) and produces a permission \(post\). A magic wand typically has some internal state, which, conjoined with \(pre\), gives rise to \(post\). (This is why the magic wand can be used just once.) We represent this internal state as an existentially quantified permission \(ammo\). Within the existential quantifier \(\{ammo: perm\}\), one finds a package of (1) a function that consumes \(pre * ammo\) and (2) one copy of \(ammo\). Because \(ammo\) is affine, a magic wand can indeed be used at most once: it is a one-shot function. The name “\(ammo\)” suggests the image of a gun that needs a special type of ammunition and is supplied with just one cartridge of that type. The reason why we need this encoding of one-shot functions, involving an explicit \(ammo\), is that we view every function as duplicable and (therefore) forbid a function from capturing a nonduplicable permission that exists at its definition site.

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

At line 19, we have reached the end of the list. We return \(\text{NotFound}\). By examining the return type of \(\text{find}\) (lines 14–16) as well as the definition of the algebraic data type \(\text{result}\) (lines 1–3), the type-checker determines that the permission \(xs @ list a\) must be returned to the caller. We have this permission, so all is well.

At line 24, the element head is the one we are looking for. We return \(\text{Found}\) applied to the pair \((\text{head}, w)\), which should (therefore) have type \(\text{focused } a (xs @ list a)\). To check that this is indeed the case, the type-checker must verify that we are able to produce the permission:

\[
\text{w @ wand (head @ a) (xs @ list a)}
\]

The definition of \(\text{w}\) at lines 22–23 gives rise to the permission:

\[
\text{w @ (| consumes (head @ a * tail @ list a)) -> (| xs @ list a)}
\]

The type-checker verifies that this permission, combined with \(\text{tail @ list a}\), entails the desired permission, shown previously. This subsumption step involves an existential quantifier introduction, taking \(\text{tail @ list a}\) as the witness for \(ammo\).

The type-checker must also verify that the definition of \(\text{w}\) at lines 22–23 is valid. This is indeed the case because \(w\) has access not only to \(\text{head @ a * tail @ list a}\), but also to the duplicable permission \(xs @ \text{Cons}\{\text{head = head}; \text{tail = tail}\}\). (In Mezzo, a function has access to every duplicable permission that exists at its definition site.) By combining these three permissions, one obtains \(xs @ list a\), as desired.

At line 30, the desired element has not been found further down: the recursive call to \(\text{find}\) returns \(\text{NotFound}\). Even though the code is terse, the reasoning is nontrivial. As we are in the \(\text{NotFound}\) branch, we have \(\text{tail @ list a}\). Furthermore, we still hold \(\text{head @ a}\) and \(\text{xs @ Cons}\{\text{head = head}; \text{tail = tail}\}\), which were framed out during the call. The type-checker recombines these permissions and verifies that we have \(xs @ list a\), as demanded in this case by the postcondition of \(\text{find}\).

---

\(^3\)Magic wands, also known as “separating implication” [Reynolds 2002], can be used to encode data structures with a hole, or data structure segments [Maeda et al. 2011]. They have also been used in the description of iterator protocols [Krishnaswami et al. 2009; Haack and Hurlin 2009].
At line 31 is the last and most intricate case. The desired element x has been found further down the list. The recursive call returns the value x, the permission \( x \circ a \), and a wand \( w \) that we are supposed to use when we are done with x. This wand has type \( (x \circ a) \rightarrow (\text{tail } \circ \text{list } a) \). At lines 33–34, we argue that it also has type \( (xs \circ \text{list } a) \). Combined with the structural permission \( r \circ \text{Found} \{ \text{found} = (x, w) \} \) obtained at line 28, this implies that \( r \) has type result \( (xs \circ \text{list } a) \). As promised on lines 14–16.

**Limits of this approach.** The strength of this approach is that it allows the user to work in direct 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 \texttt{find} function, is good. 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 nontrivial type annotations, in spite of the fact that the type-checker already performs quite a lot of work for us, including automatic elimination and (sometimes) automatic 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 \texttt{find} contains a redundant case analysis, which ideally should be unnecessary. Indeed, because \texttt{let flex} and \texttt{pack} have no runtime effect, the entire \texttt{match} construct at lines 28–36 is equivalent to just \( r \). If we could replace this construct with just \( r \), the code would be more much more transparent, and it would become clear that the recursive call is a tail call. At present, the Mezzo compiler could 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 a “ghost” function. The system would check that this function has no runtime effect (including nontermination). This would clarify the program and improve efficiency (by allowing ghost code to be erased).

---

\footnote{For the reader who would like to understand this code fragment in detail, let us say a little more. On line 31, the type-checker automatically expands the type of \( w \), namely \( (x \circ a) \rightarrow (\text{tail } \circ \text{list } a) \). This is an abbreviation for an existential type, which is automatically unpacked. Thus, \( w \) is now viewed as a function of type \( ((\text{consumes } (x \circ a) \rightarrow (\text{tail } \circ \text{list } a)) \rightarrow ((\text{head } \circ a) \rightarrow (\text{xs } \circ \text{list } a)) \rightarrow (\text{xs } \circ \text{list } a)) \). In the presence of the duplicable permission \( xs \circ \text{Cons} \{ \text{head} = \text{head}; \text{tail} = \text{tail} \} \), the codomain can be further weakened: we find that \( w \) has type \( ((\text{consumes } (x \circ a) \rightarrow (\text{head } \circ a) \rightarrow (\text{xs } \circ \text{list } a)) \rightarrow (\text{xs } \circ \text{list } a)) \). This information about \( w \) and the fact that the permission \( \text{head } \circ a \circ \text{amm} \) is available allow us in principle to recognize the definition of a magic wand and to conclude that \( w \) has type \( (x \circ a) \circ (xs \circ \text{list } a) \), as desired. This is an existential introduction step, where the witness this time is \( \text{head } \circ a \circ \text{amm} \). Unfortunately, as of today, the Mezzo type-checker is unable to automatically infer this witness. This should not be surprising, as the knowledge of the witness guides the subsumption steps that we have just described.) Instead, the programmer must explicitly provide this information, via the construct \texttt{pack w @ <existential permission> witness <witness type or permission>} (lines 33–34). This leads to another difficulty, though. The programmer cannot write that the witness is “\( \text{head } \circ a \circ \text{amm} \)”. Because the permission variable \( \text{amm} \) has been automatically introduced by the type-checker, there is no way for the programmer to refer to it. We solve this problem via the \texttt{let flex} construct on line 32. This construct introduces a flexible permission variable, called \texttt{guess}. This allows us to supply \( \text{head } \circ a \circ \text{guess } \) as the witness on line 34. When examining the \texttt{pack} construct, the type-checker is able to guess that \texttt{guess} must be unified with \( \text{amm} \). In principle, we could simplify things slightly by allowing the programmer to supply “\( \text{head } \circ a \circ \_ \)” as the witness on line 34. The wildcard would be considered syntactic sugar for a flexible variable, obviating the need for an explicit \texttt{let flex}. This has not yet been implemented.}
However, extending Mezzo with ghost code, while guaranteeing its termination, could be nontrivial. We do not wish to restrict references to base types, as done by Filliâtre et al. [2014] in order to prohibit recursion through the store. Perhaps, instead, we could adopt the elaborate “call permissions” used by Jacobs et al. [2015] in VeriFast. Or, perhaps, it is sufficient to forbid unfolding a recursive type inside ghost code. We leave these questions for the future.

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 subtrees. 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 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 immutable list: let rec xs = 0 :: xs.) 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 substructure, one must work around the restrictions described above. This requires extra machinery.
data mutable node a =
  Node {
    value : a;
    visited : bool;
    neighbors: list (node a);
  }

val _ : node int =
  let n = Node {
    value = 10;
    visited = false;
    neighbors = ();
  } in
  let ns = Cons { head = n; tail = Nil } in
  n.neighbors <- ns;

Fig. 11. A failed attempt to construct a cyclic graph

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. 11, 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:

\[ n \in \text{Node}\{\text{value: int; visited: bool; neighbors: ns}\} \ast \]
\[ \text{ns} \in \text{Cons}\{\text{head: n; tail: Nil}\} \]

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 \[ n \in \text{node int}\]. This permission subsumption step is invalid. This is the second limitation that was discussed earlier: since \[ n \in \text{Node}\{\text{...}\} \] is affine, it cannot be used to separately justify that n is a node and ns is a list of nodes. Furthermore, even if \[ n \in \text{Node}\{\text{...}\} \] was duplicable, this permission subsumption step would still be invalid. This is the first limitation discussed earlier: since the algebraic data type node is interpreted inductively, a node cannot participate in its own construction.
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;

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
  )

Fig. 12. Graphs, a cyclic graph, and depth-first search, using adoption and abandon

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 subtrees.

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. Because ownership is a hierarchy, this implies that the graph must be a hierarchy, that is, a tree.

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. 12 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 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.

Pointers and ownership are now decoupled. The existence of a pointer (at type dynamic) from a node to a successor does not imply ownership. Conversely, ownership does not imply the existence of a pointer: as we will see, the graph object owns all nodes, even though it does not necessarily have a pointer (or even a path) to them.

**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. 11. 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 (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 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”. The permission g @ graph int should be interpreted intuitively as a proof of ownership of the object g (which has type graph int) and of its adoptees (each of which has type 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 (§??), these permissions are explicitly distinguished.

Although g @ 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 `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 `abandon` instruction, `take n 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 `take n from g` requires the permissions n @ dynamic, which proves that n is the address of a valid block, and g @ graph int, which proves that g is an adopter and indicates that its adoptees have type node int. It preserves these permissions and (if successful) produces n @ 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.** An adopter owns its adoptees. Conceptually, one can imagine that an adopter maintains a list of its adoptees. More precisely, if g has been declared to adopt objects of type t, one can imagine that g has a field adoptees of type list t. An instruction `give n to g` inserts the (new) element n into the list g.adoptees. An instruction `take n from g` checks that n appears in this list (if not, the instruction fails) and removes it from the list.

In fact, this simple view of adoption and abandon could be implemented in Mezzo as a library. An adopter y would be a reference to a list of adoptees. Calling `give (x,y)` would consume x @ t and insert x into the list. The type dynamic would be defined as the “top” type {a}a, so the permission x @ dynamic would be duplicable and always available. Calling `take (x,y)` would search for x in the list (based on its address). Once found, the element x would be removed from the list of adoptees, and the permission x @ t would be returned to the caller.

This implementation would work, but would be expensive, as the cost of take would be linear in the number of adoptees. We propose an alternative representation, which allows performing `give` and `take` 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 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. One could save some space by letting the programmer decide which objects need this field (§??).

The runtime effect of the instruction `give n to g` is to write the address g to the field n.adopter. The static discipline guarantees that this field exists and that its value, prior to adoption, is null. The runtime effect of the instruction `take n from g` is to check that the field n.adopter contains the address g and to write null into this field. If this check fails, the execution of the program is aborted. We also offer an expression form, g adopts n, which tests whether n.adopter is g and produces a Boolean result. It is not described in this paper.

The reader may be worried that this mechanism introduces a data race on the adopter field. We explain in §?? that, thanks to the type discipline, the only possible race is between two `take` instructions on a single adoptee, i.e., between `take x from y` and `take x from z`, where y and z are distinct. There, we argue informally that this race is benign: intuitively, neither of these instructions can affect the outcome of the other.

**Illustration.** We illustrate the use of adoption and abandon with the example of depth-first search (Fig. 12, 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 \texttt{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 \texttt{dynamic} (line 28). Thus, the type-checker does not know a priori that \( n \) is a node. The \texttt{take} instruction (line 30) recovers this information. It is justified by the permissions \( n @ \, \texttt{dynamic} \) and \( g @ \, \texttt{graph int} \) and (if successful) produces \( n @ \, \texttt{node int} \). 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 \texttt{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 \texttt{give} would lead to a runtime failure at a later \texttt{take} instruction, typically on line 30. In order to diminish the likelihood of this particular mistake, we propose \texttt{taking n from g begin ... end} as syntactic sugar for a well-parenthesized use of \texttt{take} and \texttt{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 multiple pointers to a mutable object, 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 remains undetected and can lead to incorrect runtime behavior or security flaws [Vitek and Bokowski 2001; Tschantz and Ernst 2005].

A limitation of adoption and abandon is that \texttt{give} and \texttt{take} require exclusive ownership of the adopter. Thus, although this mechanism allows “sharing” in the sense of establishing and exploiting multiple pointers to a mutable object or data structure, it does not allow sharing mutable data between several threads. For this purpose, we use locks (§1). It is typical to use the two mechanisms together: a lock controls access to an adopter, which in turn gives access to a group of adoptees. An upside of this limitation is that, even in a concurrent setting, \texttt{give} and \texttt{take} can be implemented cheaply, using normal read and write instructions. Although a data race exists, it is benign (§??).

2.6. Nesting, an alternative to adoption and abandon

A drawback of adoption and abandon is that they incur a runtime cost in time and space. In a sense, this is justified, as they are very powerful. In particular, they allow regaining permanent ownership of an adoptee (by \texttt{take}-ing it away from its adopter), and they allow \texttt{take}-ing two distinct adoptees simultaneously from the same adopter. A purely static mechanism typically cannot support these features, because that would require statically keeping track of which objects have been taken away and exhibiting sophisticated nonaliasing proofs.

If one is willing to give up on these two features, though, it is possible to devise static mechanisms for aliasing exclusively-owned, mutable data, at no runtime cost.
Nesting [Boyland 2010] is one such mechanism. In the following, we show how (a simplified version of) nesting can be axiomatized in Mezzo.

An axiomatization of nesting. We axiomatize nesting by providing a module, nest, whose interface (shown in Fig. 13) offers a small number of types and operations. Each of these operations is a no-op: at runtime, it does nothing and returns a unit value. In other words, nesting is a purely static mechanism. It is axiomatized, as opposed to defined: that is, its implementation uses unsafe type casts. We have not proved type soundness for Mezzo with nesting. We believe that it should be possible to do so, as an extension of the type soundness proof presented in this paper.

The first item in Fig. 13 is \texttt{nests} (line 1). It is an abstract permission, that is, an abstract type of kind \texttt{perm}. It is parameterized with a value \texttt{y} and a permission \texttt{p}. The permission \texttt{nests y p} means that \texttt{p} has been “nested” in \texttt{y}, or delegated to \texttt{y}. In other words, whoever has (exclusive) ownership of the object \texttt{y} also (implicitly) possesses the permission \texttt{p}.

Nesting is similar to adoption and abandon. In both mechanisms, a permission is delegated to an object \texttt{y}. In nesting, an arbitrary permission \texttt{p} can be delegated. In adoption and abandon, it must be a permission of the form \texttt{x @ u}, where \texttt{x} is the adoptee and \texttt{u} is the agreed-upon type of \texttt{y}’s adoptees.

In nesting, the permission \texttt{nests y p} serves as a static witness that \texttt{p} has been nested in \texttt{y}, whereas in adoption and abandon, there is no such witness. All we have is \texttt{x @ dynamic}, which allows us to perform a dynamic ownership test, via the instruction \texttt{take x from y}.

The permission \texttt{nests y p} is duplicable (line 2). In other words, the fact that \texttt{p} has been nested in \texttt{y} can be advertised without restriction. This is sound because such a fact, once true, remains true forever: nesting cannot be undone. Similarly, in adoption and abandon, the type \texttt{dynamic} is duplicable. This serves a common purpose, namely to allow sharing a piece of information about the nestee, or adoptee.

In adoption and abandon, a pair of dual operations, \texttt{give} and \texttt{take}, allow delegating the ownership of some object \texttt{x} to an adopter \texttt{y} and taking it back. Nesting offers a set of operations that serve a similar purpose. Their types are crafted in such a way that one cannot simultaneously take two permissions away from a single object \texttt{y}. Two operations, \texttt{nest} and \texttt{defocus}, correspond to \texttt{give}. One operation, \texttt{focus}, corresponds to \texttt{take}. We stress, once more, that these operations have no runtime effect.

A new nesting relationship is established by a call to \texttt{nest} (line 4). Such a call takes the permission \texttt{p} away from the caller and transfers it to the object \texttt{y}, or in other words,
to whoever owns \( y \). Thus, the caller loses \( p \) and gains the (duplicable) nesting witness \( y @ p \). Perhaps surprisingly, no permission for \( y \) is required.

If and when one wishes to regain \( p \), one can do so by invoking focus (line 9). This operation requires proof that \( p \) has been nested in \( y \): this is encoded by requiring \( y @ p \). Furthermore, exclusive ownership of \( y \) is needed. This is expressed by requiring \( y @ a \), for an arbitrary exclusive type \( a \). Naturally, one must not allow this operation to be performed twice in sequence: that would allow the user to obtain two copies of \( p \), which may be nonduplicable. In order to disallow this, focus consumes the exclusive permission \( y @ a \). In its stead, it produces the permission \( y @ a \), which is not exclusive (hence, does not allow focusing on \( y \) again).

The permission \( y @ a \) is, in essence, a magic wand (in Boyland’s words, a linear implication). It means that, once the user is done working with \( p \), she may give it up and recover \( y @ a \). This is done via a call to defocus (line 12).

In a typical call to nest, focus, or defocus, the parameters \( y \) and \( p \) must be explicitly instantiated by the programmer, as they cannot be inferred. This is illustrated further on in our re-implementation of graphs using nesting.

Our version of nesting is not as powerful as Boyland’s. For instance, Boyland allows per-field nesting: one may nest \( p \) in \( y.f \). Furthermore, his theory includes fractional permissions, which interact with nesting in subtle ways. Nevertheless, nesting in Mezzo has potentially interesting applications. In the following, we re-formulate the example of graphs (§2.5) using nesting.

**Implementing graphs with nesting.** The code in Fig. 14 defines graphs, constructs a cyclic graph, and defines depth-first search, based on nesting. It should be compared with the code in Fig. 12, which is based on adoption and abandon.

In both approaches, in order to allow aliasing, we wish to conceptually place all of the graph nodes in a group and to have just one permission for the entire group, as opposed to one permission per node.

In the present case, we do this by nesting the permission for every node in a single object. This could be the graph \( g \) itself. Here, we prefer to use a separate (dummy) object \( r \), which we then store in a field of \( g \). This object \( r \) serves no purpose besides nesting every node. We refer to it as a “region”.

We begin by defining the algebraic data type \( \text{region} \) (line 3). Like the unit type \( () \), it has just one data constructor, of arity zero. Unlike the unit type, it is declared \texttt{mutable} , which implies that a region has a unique owner and that the permission \( r @ \text{region} \) is exclusive. This allows us to use \( r \) in focus and defocus operations.

For every graph node \( x \), we intend to nest the permission for \( x \) in the region \( r \). In other words, we intend every node to become an inhabitant of this region. For greater clarity, a type of “region inhabitants” is made explicit via a type abbreviation (line 6). (The type \texttt{unknown} is surface syntax for \( \top \), that is, a duplicable type that carries no information.) Thus, by definition, the permission \( x @ \text{inhabitant} \ r \) is synonymous with nests \( r \ (x @ a) \).

We are now ready to define the type node (line 9). We parameterize it not only over the type \( a \) of the content field, as before (Fig. 12), but also over a region \( r \). Indeed, this time, we intend to statically keep track of which region the nodes inhabit. We declare

---

5Like \texttt{duplicable a}, \texttt{exclusive a} is an assertion about the type \( a \), and can be viewed as a permission. It is not formalized in this paper. Intuitively, \texttt{exclusive a} holds if and only if (1) the permission \( y @ a \) implies exclusive ownership of the object that exists at address \( y \) in the heap and (2) \( y \) has not been focused. It is not synonymous with \texttt{affine a}, which in Mezzo would be true of every type \( a \).

6The operation \texttt{nest} corresponds roughly to transformation 7 in Boyland’s Theorem 6.4 [2010]. The operations \texttt{focus} and \texttt{defocus} correspond roughly to the second item and fourth items, counting from the end, in Boyland’s Theorem 6.2.
open nest

data mutable region =
  Region

alias inhabitant (r : value) a =
  (x: unknown | nests r (x @ a))

data mutable node (r: value) a =
  Node {
    visited : bool;
    content : a;
    neighbors: list (inhabitant r (node r a))
  }

alias inode (r: value) a =
  inhabitant r (node r a)

alias graph a =
  (r: region, roots: list (inode r a))

val g : graph int =
  let r = Region in
  let n = Node {
    visited = false;
    content = 10;
    neighbors = nil;
  } in
  nest [r, (n @ node r int)] ();
  focus [r] ();
  n.neighbors <- ns;
  defocus [r] ();
  (r, ns)

val dfs [a] (g: graph a, f: a -> ()): () =
  let (r, roots) = g in
  let s : stack (inode r a) = stack::new roots in
  stack::work [inode r a] (s, fun (n: inode r a
    | r @ region * s @ stack (inode r a)) : () =
      focus [r] ();
      if not n.visited then begin
        n.visited <- true;
        f n.content;
        let ns = n.neighbors in
        stack::push [inode r a] (ns, s);
      end;
      defocus [r] ()
    )

Fig. 14. Alternative implementation of graphs using nesting
that the neighbors field holds a list of nodes in the region \( r \) (line 13). It is important to note that, although node \( r \ a \) is affine, inhabitant \( r \) (node \( r \ a \)) is duplicable. This stems from the fact that \( \text{nests} \ r \ p \) is duplicable even if \( p \) is affine. Thus, the type of the neighbors field indicates that the neighbors are nodes, and indicates which region they inhabit, but does not claim that "each node owns its successors". The ownership of all nodes, collectively, lies with the unique permission \( r \ @ \text{region} \).

We define \( \text{inode} \ r \ a \) as an abbreviation for inhabitant \( r \) (node \( r \ a \)) (line 16), and define a graph as a pair of a region \( r \) and a list \( \text{roots} \) of nodes that inhabit \( r \) (line 19). The ability for the components of a tuple to refer to one another is exploited here.

As before, we construct a cyclic graph \( g \), composed of just one node that is its own successor (lines 22–34). Whereas our earlier construction based on adoption and abandon (Fig. 12, lines 13–24) involved just one \textit{give} instruction, placed after the assignment \( \text{n.neighbors} \leftarrow \text{ns} \), here we must use \textit{nest} before the assignment (line 30) and use \textit{focus} and \textit{defocus} afterwards (lines 31 and 33). The reason why we can get away in Fig. 12 with just one \textit{give} instruction is that the permission \( \text{ns} @ \text{list} \) already exists before we give \( n \) to \( g \). (Every node has type \texttt{dynamic} at every time, regardless of which \textit{give} and \textit{take} instructions have been executed.) In Fig. 14, in contrast, we have a chicken-and-egg problem of sorts. In order to argue that the singleton list \( \text{ns} \) has type \( \text{list} \ (\text{inode} \ r \ a) \), we need the node \( n \) to inhabit the region \( r \). So, we must first nest \( n \) in \( r \). To do this, however, we must first prove that \( n \) is a well-formed node, that is, we must exhibit \( n @ \text{node} \ r \ \text{int} \). And, to do this, we need \( \text{n.neighbors} \) to have type \( \text{list} \ (\text{inode} \ r \ a) \). We work around this circularity by initializing \( \text{n.neighbors} \) with the empty list. This allows us to nest \( n \) in \( r \), which is good, but unfortunately takes the permission \( n @ \text{node} \ r \ \text{int} \) away from us. We temporarily recover this permission via \textit{focus} and \textit{defocus}, which allows us to justify the assignment \( \text{n.neighbors} \leftarrow \text{ns} \).

The call to \textit{nest} (line 30) uses an explicit type application. Indeed, the type-checker cannot guess which permission we wish to nest in which object. The calls to \textit{focus} and \textit{defocus} (lines 31 and 33) also use explicit type applications. There, it is sufficient to instantiate \( r \). The type-checker is able to guess that \( p \) must be instantiated with \( n @ \text{node} \ r \ \text{int} \), as there is no other choice.

The code for depth-first search (lines 36–49) is very similar to its counterpart in Fig. 12. Instead of a pair of \textit{give} and \textit{take} operations, we use a pair of \textit{focus} and \textit{defocus} operations (lines 41 and 48).

Discussion. In light of this example, nesting may appear preferable to adoption and abandon. Indeed, it has no runtime cost. Furthermore, it gives rise to more precise types: \( \text{inode} \ r \ a \) is arguably a more satisfactory piece of information than \texttt{dynamic}. However, adoption and abandon is more flexible. In particular, only adoption and abandon allows permanently detaching a node from a graph, perhaps in order to attach it to some other graph, or perhaps in order to permanently reclaim unique ownership of the value stored in its content field. Also, only adoption and abandon allows taking two elements at the same time. Finally, because every mutable object has type \texttt{dynamic} even before it is adopted, adoption and abandon makes it easy to build cyclic data structures. Nesting, in comparison, requires an object to have its final (fully initialized) type before it is nested, which may make it awkward or impossible to build a cyclic data structure.

The fact that nesting can be axiomatized in Mezzo seems a testimony to the expressive power of Mezzo’s basic type discipline. The distinction between duplicable and affine types (and permissions), and the ability of types (and permissions) to refer to values, are powerful features, which may well allow a variety of ownership disciplines to be axiomatized.
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 (§??–§??) 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 (§??–§??). 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 traditional quantifiers. The second difference is that the surface syntax adopts the convention that functions by default do not consume their argument, and offers a `consumes` keyword to indicate that (part of) the argument is in fact consumed. In contrast, the core syntax does not have a `consumes` keyword; it adopts the convention that functions do consume their argument, and repeat the nonconsumed 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 §??. 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-kindedness 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 `find` (§2.4, Fig. 10).

\[
[a] \ (\text{consumes } xs: \text{list } a) \rightarrow \\
(x : a, \text{wand } (x @ a) (xs @ \text{list } a))
\]

The name introduction construct \( xs: \text{list } a \) 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 \( xs @ \text{list } a \)) is bound by the name introduction.

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

\[
[a] \ [xs : \text{value}] \\
(\text{consumes } (=xs | xs @ \text{list } a)) \rightarrow \\
\{x : \text{value}\}
\]

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article XXXX, Publication date: January 2015.
\((=x \mid x @ a), \text{wand} (x @ a) (xs @ \text{list } a)\)

The name \(xs\) is now universally quantified (at kind 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 value) above the codomain. Thus, its scope is the codomain.

The name introduction \(xs: \text{list } a\) is now replaced with \((=xs \mid xs @ \text{list } a)\). This is a conjunction of a (singleton) type and a permission. This means that the function \(\text{find}\) 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 \(\text{find } ys\) is well-typed, provided the caller is able to provide the permission \(ys @ \text{list } a\). Similarly, the name introduction \(x: a\) is replaced with \((=x \mid x @ a)\).

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 \(\langle x: t, y: u \rangle\), both of the variables \(x\) and \(y\) are in scope in both of the types \(t\) and \(u\). The high flexibility of this name introduction construct was illustrated in Fig. 14, where the type graph can name itself and refer to itself.

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

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

In this example, because the argument is entirely consumed, the translation is trivial. All we have to do is erase the \texttt{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 \texttt{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 \texttt{consumes}. Consider, for instance, the type of a function that merges two sets, updating its first argument and destroying its second argument:

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

The domain of this function type is a pair, whose second component is marked with the keyword \texttt{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 \(x\):

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

The symbol \(\top\) is an abbreviation for \(\exists (y : \text{value}) =y\), which is a duplicable but noninformative 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-kindness

The combined surface and core syntaxes of Mezzo are presented in Fig. 15. 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 consumes $T$ appear only in the surface syntax. All of the other constructs are shared between the two levels.

The well-kindness 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 nonstandard, 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-kindness rules, one must at certain well-defined points go down and collect the names that are introduced by some name introduction form, so as to extend the environment with assumptions about these names.

The auxiliary function $\text{names}(T)$, which collects the names introduced by the type $T$, is defined in Fig. 16. 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. 18) relies on an auxiliary judgement,
\(\Gamma \vdash \#T : \kappa\), which by definition means \(\Gamma; \text{names}(T) \vdash T : \kappa\) (Fig. 17). Intuitively, \# is a “beginning-of-scope” mark: it means that, at this point, the names collected by the auxiliary function names are in scope. We stress that the \# symbol is not part of the syntax of types, and is not exposed to the user. It is purely a technical means of formulating our rules in a convenient manner.

There are additional restrictions that the well-kindness rules should impose: for instance, the `consumes` keyword should appear only in the left-hand side of an arrow, and should not appear under another `consumes` keyword. This can be expressed by extending the well-kindness judgement with a Boolean parameter, which indicates whether `consumes` is allowed or disallowed. In order to reduce clutter, we omit this aspect.
it constructs distinct pre- and postconditions, namely order to express the meaning of the to the argument of the function; this is imposed by the singleton type ⇝ face arrow copy of T simply encode a recursive traversal. The definition appears in Fig. 21. Only one rule is shown, as the other rules (omitted) other words, “a copy of T" represents the ownership of the argument, including the components marked with consumes, whereas the permission x @ T” represents the ownership of the argument, deprived of these components.

Fig. 21. Types and permissions: second translation phase (only one rule shown)

3.3. Translation

We now define the translation of (well-kinded) 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 surface function type into its core counterpart, and at the same time eliminate the consumes construct.

Phase 1. The first phase is described by the translation judgement T ▶ T’, whose definition (Fig. 20) relies on the auxiliary judgement #T ▶ T’ (Fig. 19).

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-NAMEINTRO, which translates a name introduction form to a conjunction of a singleton type =x and a permission x @ 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-kindedness is preserved by the first translation phase:

— Γ ⊢ T : κ and T ▶ T’ imply Γ ⊢ T’ : κ.
— Γ ⊢ #T : κ and #T ▶ T’ imply Γ ⊢ T’ : κ.

Claim 3.2. If T ▶ T’ or #T ▶ T’ holds, then T’ does not contain a name introduction construct.

Phase 2. The second phase is described by the translation judgement T ▷ T’, whose definition appears in Fig. 21. 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 a surface arrow ¬→ into a core arrow →. Second, it introduces a fresh name, x, which refers to the argument of the function; this is imposed by the singleton type =x. Finally, in order to express the meaning of the consumes keywords that may appear in the type T’, it constructs distinct pre- and postconditions, namely x @ T_1' and x @ T_2'. These permissions respectively represent the properties of x that the function requires (prior to the call) and ensures (after the call).

The type T_1'^in is defined as [T/consumes T|T_1']. By this informal notation, we mean “a copy of T_1 where every subterm of the form consumes T is replaced with just T”, or in other words, “a copy of T_1 where every consumes keyword is erased”.

The type T_2'^out is defined as [?/consumes T|T_2']. By this informal notation, we mean “a copy of T_2 where every subterm of the form consumes T is replaced with ∃(x : κ) x, where the kind κ is either type or perm, as appropriate”. We note that ∃(x : κ) x is a “top” type or permission: it does not provide any useful information.

Thus, the permission x @ T_1'^in represents the ownership of the argument, including the components marked with consumes, whereas the permission x @ T_2'^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 $\Gamma \vdash T : \kappa$ and $T \triangleright T'$ hold, then $T'$ contains no surface arrow $\rightsquigarrow$ and no consumes keyword.

4. THE IMPLEMENTATION OF MEZZO

The current implementation of Mezzo is made up of about 12,500 (nonblank, noncomment) lines of OCaml code, along with 3,000 (nonblank, noncomment) lines of Mezzo code for the standard library. It features:

— a type-checker, which is the main contribution;
— an interpreter, which is used mainly in the online version of Mezzo [Protzenko 2014c], where programs are type-checked and run in a browser, via an OCaml-to-JavaScript compilation scheme;
— an OCaml backend, which translates Mezzo into untyped OCaml, that is, OCaml with unsafe casts (Obj.magic).

The compiler can be integrated within OCaml's compilation toolchain. A sample project shows how to drive the compilation of a Mezzo program using OCamlbuild [Protzenko 2014b].

4.1. Problems addressed by the type-checker

The type-checker performs a flow-sensitive forward analysis of the code. At each program point, it computes a (persistent) representation of the currently available set of permissions. Permissions are consumed and added as the type-checker steps through the program. This process can be described by a set of “algorithmic” typing rules [Protzenko 2014a, Chapter 10], where the frame rule is no longer a stand-alone rule, but instead is built into every axiom.

The type-checker faces several difficult problems, which we briefly explain below. The first four points (entailment; frame inference; inference of polymorphic instantiations; intersection types) in fact describe four facets of a single, complex problem. The last one (join) describes a separate problem.

Entailment. At a function call site, the type-checker must prove that the current permission $P$ justifies the call, that is, $P$ entails the precondition $Q$ of the function. This may involve applying subsumption rules so as to obtain the desired permission. For instance, if $P$ contains the conjunct $\text{xs} @ \text{Cons} \{ \text{head}: a; \text{tail}: \text{list}\ a \}$, then, in order to justify the call $\text{length}\ \text{xs}$, the type-checker must first convert this permission to $\text{xs} @ \text{list}\ a$. Entailment is used also at the end of a function's body, where the type-checker must verify that the current permission entails the function's postcondition (which is provided by the programmer as part of the function's header).

The entailment problem in Mezzo is roughly analogous to the entailment problem in separation logic. If formulae are restricted to (dis)equalities and spatial conjunctions of points-to assertions and list segments, then the latter problem is decidable [Berdine et al. 2004; Navarro Pérez and Rybalchenko 2011; Piskac et al. 2013] and tractable [Cook et al. 2011]. However, the hardness results obtained by Antonopoulos et al. [2014] indicate that entailment becomes intractable as soon as one combines existential quantification and unbounded data structures.

Another informal argument why the entailment problem in Mezzo is probably hard is that Mezzo contains System F, whose subtyping problem is undecidable [Chrzastcz 1998]. In other words, quantifiers and function types alone give rise to undecidability. This situation is further complicated in Mezzo by the nontrivial subsumption axioms.
that involve function types, such as `FRAME_SUB` and `HIDE DUPPLICABLE PRECONDITION` in Fig. ??.

Frame inference. At a function call site, the type-checker must not only prove that the current permission $P$ entails the precondition $Q$, but also find the strongest possible permission $R$ such that $P$ entails $Q * R$. The permission $R$ is the “remainder”, or the “frame”. It is considered automatically preserved by the call. This problem, which subsumes the entailment problem, is known as the frame inference problem [Berdine et al. 2005b]. It can be viewed as an entailment problem with one flexible permission variable (namely, $R$) on the right-hand side.

Inference of polymorphic instantiations. At a function call site, if the callee is a polymorphic function, the type-checker must not only solve a frame inference problem, but also (and at the same time) find a suitable instantiation of the universal quantifiers. This can be viewed as an entailment problem with flexible variables on the right-hand side.

In fact, due to the contravariance of function types, flexible variables can occur also in the left-hand side of an entailment problem. In general, the type-checker faces problems of the form $P \vdash Q$ where both $P$ and $Q$ contain flexible variables.

These problems do not in general have unique or “best” solutions. For instance, a call to a function of type $[p: \text{perm}] (\downarrow \text{consumes } p) \rightarrow ()$ could consume no permission at all, or the entire current permission, or anything in between, depending on how one chooses to instantiate the permission variable $p$. In this case, instantiating $p$ with `empty` is the best solution. As another example, in a call to a function of type $[p: \text{perm}] (\downarrow p) \rightarrow ()$, any instantiation of $p$ is as good as any other, since $p$ is not consumed. In general, there are problems that admit several incomparable solutions. The current type-checker picks one solution, based on a set of heuristics. If this solution is not satisfactory, then the programmer must provide an annotation that indicates how the polymorphic function should be instantiated. This can be done via an explicit type application, as in Fig. 22, or via an annotation that indicates which permissions one expects to possess after the call, as in Fig. 23.

Intersection types. Mezzo is able to encode intersection types: in the presence of the permissions $f @ t_1 \rightarrow u_1$ and $f @ t_2 \rightarrow u_2$, the function $f$ has type $t_1 \rightarrow u_1$ and $t_2 \rightarrow u_2$ at the same time. Thus, a function call $f \ x$ is well-typed if the current permission entails $x @ t_1$ or $x @ t_2$. Such a situation requires the type-checker to explore both avenues. This phenomenon has appeared in practice in our case study on iterators [Guéneau et al. 2013], where a single function is used to encode several magic wands.

Join. At a join point in the control-flow graph, the type-checker must construct the least upper bound of two (or more) permissions. For instance, after an `if` expression, if the current permissions at the end of the `then` and `else` branches are respectively

```
1 val x = newref 0
2 val l: lock (x @ ref int) = lock::new ()
```

Fig. 23. A more idiomatic type annotation
The Design and Formalization of Mezzo, a Permission-Based Programming Language

val x =  
  if ... then  
    (newref 0, newref 0)  
  else begin  
    let y = newref 0 in  
    (y, y)  
  end

Fig. 24. An ambiguity at a join point

val f (): (ref int, unknown) =  
  if ... then  
    (newref 0, newref 0)  
  else begin  
    let y = newref 0 in  
    (y, y)  
  end

Fig. 25. An ambiguity at a join point, resolved by an explicit annotation

If the user is dissatisfied with the choice made by the type-checker, she can explicitly provide \( P \), in which case the join problem disappears altogether and is replaced with a collection of entailment problems. In particular, when a \texttt{match} construct appears in terminal position in a function, the join point after the \texttt{match} coincides with the end of the function’s body, so the postcondition of the function (which must always be explicitly provided) serves as an explicit \( P \). This is the case in Fig. 25, where the return type of \( f \) is explicitly provided by the user.
In light of these difficulties, it should come as no surprise that our current algorithms are incomplete. More precisely, our frame inference and join algorithms sometimes make arbitrary choices: an annotation must be provided so as to guide them. Even in the presence of annotations, our entailment algorithm may fail to verify a valid entailment and may fail to terminate. This is an admittedly rather unsatisfactory state of affairs; future work in this area is needed.

4.2. Proof search and backtracking

When confronted with a frame inference problem: “from the permission \( P \), can one extract \( P' \) and, if so, what is the remainder \( R' \)?”, the type-checker may have to explore several avenues, backtracking when one of them fails. It may be the case that several subsumption rules can be applied (and do not commute), or that a single subsumption rule can be applied in several ways. For instance, assume \( P \) is as follows:

\[
\begin{align*}
x @ & \text{Cons} \{ \text{head} = h; \text{tail} = t \} * \\
h @ & (=h_1, =h_2) * \\
h_1 @ & \text{int} * \\
h_2 @ & \text{int} * \\
t @ & \text{list} (\text{int, int})
\end{align*}
\]

Assume further that \( P' \) is \( x @ \text{list} \ a \), where the type variable \( a \) is flexible, i.e., a suitable instantiation of \( a \) must be guessed. Such a frame inference problem arises, for instance, at a call to \( \text{length} \ x \).

Because \( x \) is obviously a Cons cell, the type-checker first strengthens the goal, which becomes \( x @ \text{Cons} \{ \text{head}: a; \text{tail}: \text{list} \ a \} \). There is no loss of generality in this step.

Then, the type-checker compares the structural permission that describes \( x \), namely \( x @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \), with the goal, and deduces that the goal can be reduced to the conjunction \( h @ a * t @ \text{list} \ a \). Again, this step involves no loss of generality.

The type-checker now attempts to solve the two sub-goals \( h @ a \) and \( t @ \text{list} \ a \), one after the other, in an arbitrary order. This is where trouble begins: because these sub-goals share the flexible variable \( a \), they are not independent of one another.

Let us assume that the type-checker attempts to solve \( h @ a \) first. That is, it must extract \( h @ a \) out of the current permission, for some choice of \( a \). Perhaps surprisingly, several choices of \( a \) are possible:

- Instantiating \( a \) with the singleton type \( =h \) solves the first sub-goal, because \( h @ =h \) holds trivially. However, this choice leads to a failure in the second sub-goal, because \( t @ \text{list} \ (\neg h) \) does not hold: not all elements of the list are equal to its head \( h \).
- Instantiating \( a \) with \( (=h_1, =h_2) \) similarly solves the first sub-goal and leads to a failure in the second sub-goal.
- Instantiating \( a \) with \( (\text{int, int}) \) is the right choice, as it allows both sub-goals to succeed. More generally, the type-checker could decide that “\( a \) should be a pair type” and encode this by introducing two new flexible variables \( a1 \) and \( a2 \), instantiating \( a \) with the pair type \( (a1, a2) \), and resuming the search, which then recursively attempts to obtain \( h1 @ a1 \) and \( h2 @ a2 \). This leads to instantiating \( a1 \) with \( \text{int} \) and \( a2 \) with \( \text{int} \), among several other possibilities.
- In fact, there are infinitely many successful ways of instantiating \( a \). For instance, one may instantiate with \( (\text{int, int} | \text{empty}) \), with \( (\text{int, int} | \text{empty} * \text{empty}) \), and so on. Although these examples are contrived, the fact that the search tree has an infinite branching factor is a major issue.
In this example, instantiating a with (int, int) is the simplest choice that eventually leads to a success. In order to come up with this conclusion, however, backtracking appears to be necessary.

We have experimented with heuristics that attempt to favor the “right” choice up front (e.g., “instantiate a type variable with a singleton type only as a last resort”). We believe that they are useful insofar as they increase efficiency and they reduce the number of situations where the type-checker commits an incorrect choice and an explicit type annotation must be provided. They do not in general eliminate the need for backtracking.

In order to limit the number of branches that the type-checker explores, we only consider a subset of the valid instantiation choices. In particular, we never consider instantiating a flexible variable with a conjunction, such as p1 * p2 or a1 | p2. If a universally quantified variable must be instantiated with a type or permission of this form, then an explicit type application is required. In Fig. 14, for instance, the calls to the polymorphic functions stack::new, stack::work, and stack::push (lines 38, 39, and 46) must be explicitly annotated: the type-checker is not able to infer that the appropriate instantiation is inode r a. Indeed, this type is an abbreviation for (x: unknown | nests r (x @ node r a)), a complex type that involves a conjunction and an existential quantification.

Over the 3,000 lines of Mezzo code in the standard library, one finds only 73 explicit type applications (as in Fig. 22) and 28 explicit type annotations (as in Fig. 23). This amounts to roughly one annotation every 30 lines of code. We believe this to be an acceptable burden, especially considering that some of the type annotations serve as documentation.

We limit the scope of backtracking to one invocation of the frame inference algorithm. If, upon type-checking a function call, we find several ways of successfully type-checking the function call, then we make an arbitrary choice and stick to it when type-checking the remainder of the code (as opposed to exploring every choice until one is found that allows the remainder of the code to be type-checked). This strategy seems less costly and more predictable than unlimited backtracking.

4.3. Implementation details
The type-checker’s implementation can be described by a set of “algorithmic” typing rules [Protzenko 2014a, Chapter 10]. Unlike the typing rules shown in the present paper, where every variable is “rigid”, these rules support “flexible” variables, which represent deferred (as-yet-undetermined) instantiations of universal quantifiers. The frame inference algorithm (also known as “subtraction”) and the join algorithm are also described in Protzenko’s dissertation [Protzenko 2014a, Chapters 11 and 12].

5. 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 (nonduplicable) element from its container (§2.4) and when one wishes to build mutable data structures that involve arbitrary aliasing patterns. 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 adoption and abandon, a dynamic mechanism (§2.5), part of our formalization (§??), and a contribution of this paper; or nesting, a purely static mechanism (§2.6), not part of our formalization.

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

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

This paper is accompanied with a Coq proof [Balabonski and Pottier 2014]. It is about 14,000 (nonblank, noncomment) 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.

We have listed earlier (§1.2) the main goals that motivated and guided the design of Mezzo. Which of these goals have been met? We believe that Mezzo is remarkably simple, concise, expressive, often more so than competing proposals. Its modular and machine-checked meta-theory not only guarantees that it can be trusted, but also explains its design and hopefully can serve as a guide in future endeavors. That said, not everything is perfect. For instance, even though every function signature is explicitly provided by the programmer, type-checking remains a hard problem, which involves a good deal of inference, and which we have not fully solved (§4). Another issue is that, even though Mezzo is implemented on top of the OCaml runtime system, safe interoperability with OCaml is currently missing: it is not clear how to safely translate a Mezzo type to an OCaml type, or vice-versa. We reflect on these issues (among others) in a conference paper [Pottier and Protzenko 2015]. Mezzo is but a step along the way. There remains ample room for further research.

REFERENCES


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article XXXX, Publication date: January 2015.
The Design and Formalization of Mezzo, a Permission-Based Programming Language


François Pottier and Jonathan Protzenko. 2015. A few lessons from the Mezzo project. In Summit on Advances in Programming Languages (SNAPL).