XXXX

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:
Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2015. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Trans. Program. Lang. Syst.* V, N, Article XXXX (January 2015), 94 pages.
DOI: 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,

Author's current affiliations: T. Balabonski, LRI, Univ. Paris-Sud, Université Paris-Saclay and F. Pottier, INRIA and J. Protzenko, Microsoft Research.

DOI: http://dx.doi.org/10.1145/2837022

```
1 open woref
2
3 val f (x: frozen int, y: frozen int) : int =
4   get x + get y
5
6 val _ : int =
7   let r = new () in
8   set (r, 3);
9   f (r, r)
```

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

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 `woref`, whose implementation we show later on (§2.1).

At line 7, we create a write-once reference by calling `woref::new`. (Thanks to the declaration **open** `woref`, 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 $\Gamma \vdash$ `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.

```
1 open thread
2
3 val r = newref 0
4 val f (| r @ ref int) : () =
5   r := !r + 1
6 val () =
7   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 convention, `f @ (| r @ ref int) -> ()` means that `f` requires *and returns* the permission `r @ ref int`. When one wishes to indicate that a function requires some permission but does not return it, one must precede that permission with the keyword **`consumes`**.

On line 7 is a sequencing construct. The second call to `spawn` is type-checked using the permissions that are left over after the first `spawn`. A call `spawn f` requires two permissions: a (duplicable) permission that describes the function `f`, and the nonduplicable permission `r @ 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 spawns, we no longer have a permission for `r`. (We still have `f @ (| r @ ref int) -> ()`, as it is duplicable.) Therefore, the second `spawn` is ill-typed. The racy program of Fig. 2 is rejected.

This behavior should be contrasted with that of the earlier example. In Fig. 1, the permission `r @ frozen int`, which `get` requires, is duplicable. We can therefore obtain two copies of it and justify the call `f (r, r)`. We could also justify several concurrent calls to `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 @ (`**`consumes`**` a | s) -> b`. At the end of line 7, after the call to `lock::new`, we have given up `s`, which has been consumed by the call, and we have obtained `l @ 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 @ (`**`consumes`**` a | s) -> b` and `l @ 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 @ 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).

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

Fig. 3. The polymorphic higher-order function `hide` takes a function `f` of type `(`**`consumes`**` a | s) -> 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 `(`**`consumes`**` a) -> b`, which does not require `s`, hence can be invoked by multiple threads concurrently. The type variables `a` and `b` have kind **`type`** (this is the default kind). The square brackets denote universal quantification.

The anonymous function at line 8 does not require or return `s`. Yet, it needs `s` in order to invoke `f`. It obtains `s` by acquiring the lock, and gives it up by releasing the lock. Thus, `s` is available only to a thread that has entered the critical section. The side effect is now hidden, in the sense that the anonymous function has type `(consumes a) -> 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 @ () -> ()`, 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 != 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 **`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 `[a] list a -> 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 `length`, and `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 **`give`** and **`take`**, move from one metaphor to the other, thus marrying 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 @ ref int`, which means "`r` is a reference to an integer", mentions the variable `r`. The permission `l @ lock (r @ 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, §6), channels, ...), adoption and abandon (§2.5, §7), 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 λ-calculus (§4). In its typed version, it is an affine, polymorphic, value-dependent system, which enjoys type erasure: values exist at runtime, whereas types and permissions do not. Although this calculus does not have explicit side effects, we endow it with an abstract notion of machine state, and we organize the proof of type soundness in such a way that the statements of the main lemmas need not be altered as we introduce new forms of side effects. The next three layers are heap-allocated references (§5), locks (§6), and adoption and abandon (§7). These three layers are almost independent of one another. There is one dependency:

```
1 data mutable writable =
2   Writable { contents: () }
3
4 data frozen a =
5   Frozen   { contents: (a | duplicable a) }
6
7 val new () : writable =
8   Writable { contents = () }
9
10 val set [a] (consumes r: writable, x: a | duplicable a)
11     : (| r @ frozen a) =
12   r.contents <- x;
13   tag of r <- Frozen
14
15 val get [a] (r: frozen a) : a =
16   r.contents
```

Fig. 4. Implementation of write-once references

adoption and abandon is piggybacked on top of heap-allocated state. Yet, we are able to structure the meta-theory in such a way that there is very little interaction between these two features. Our definitions and proofs are machine-checked and are available online [Balabonski and Pottier 2014].

The paper ends with an overview of the features of Mezzo that we could not describe here (§8), a discussion of the implementation of Mezzo (§9), and a review of the related work (§10).

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`) and a regular field, also called `contents`[1], which has type `(a | duplicable a)`. This is a type of the form `t | p`: indeed, `a` is a type, while `duplicable a` is a permission. This means that the value stored at runtime in the `contents` field has type `a`, and is logically accompanied by a proof that the type `a` is duplicable.

Why do we impose the constraint `duplicable a` as part of the definition of the type `frozen a`? The reason is, a write-once reference is typically intended to be shared after it has been initialized. (If one did not wish to share it, then one could use a standard read/write, uniquely-owned reference.) Thus, its content is meant to be accessed by multiple readers. This is permitted by the type system only if the type `a` is duplicable. Technically, the constraint `duplicable a` 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 `frozen` does not contain the keyword `mutable`, so a block of type `frozen a` is immutable. Thus, it is safe to share read access to such a block. Furthermore, because we have imposed the constraint `duplicable a`, it is also safe to share the data structure of type `a` whose address is stored in the `contents` field. In other words, by inspection of the definition, the type-checker recognizes that the type `frozen a` is duplicable as a whole. This means that a write-once reference can be shared after it has been initialized.

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

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

The code of `set` is in two steps. First, the value `x` is written to the field `r.contents` (line 12). After this update, the memory block at address `r` is described by the permission `r @ 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 `writable`; neither is it an unfolding of `frozen a`. The memory block is in an intermediate state.

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

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

In general, the tag update instruction allows changing the type of a memory block to a completely unrelated type, with two restrictions: (i) the block must initially be

[1]Mezzo allows two user-defined types to have fields that go by the same name.

```
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

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

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

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

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

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

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

2.2. Lists

The example of write-once references has allowed us to discuss a number of concepts, including affine versus duplicable permissions, mutable versus immutable memory 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, `list a`, is defined in a standard way (Fig. 6). This definition does not use the keyword **`mutable`**. These are standard immutable lists, that is, lists with an immutable spine. The list elements may be mutable or immutable, depending on how the type parameter a is instantiated.

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:

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

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:

```
let ws = append (tail, ys) 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

```
111       let zs = Cons { head = head; tail = ws } in
112       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 @ list a` and `ys @ 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) -> list a`, where the **consumes** keyword bears on `xs` only. Recall that, by convention, the absence of the **consumes** keyword means that a permission is requested and returned. In other words, the above type is in fact syntactic sugar for the following, more verbose type:

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

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

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

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

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

Technically, this phenomenon may be summed up as follows. In a context where the type `t` is known to be duplicable, the function types `(consumes t) -> u` and `t -> u` are equivalent, that is, subtypes of one another. It would be premature to prove this claim at this point; let us simply say that one direction is obvious, while the other direction follows from the frame rule and the duplication rule (FRAMESUB and DUPLICATE, Fig. 26).

As a corollary, the universal type `[a] (consumes (list a, list a)) -> list a`, which is the type of append in Fig. 6, is strictly more general than the type `[a] (list a, list a | duplicable a) -> list a`, where the **consumes** keyword has been removed, but the type `a` of the list elements is required to be duplicable. In particular, this explains why append effectively does *not* consume its arguments when they have duplicable type.

List concatenation in tail-recursive style. The append function that we have discussed so far is a direct translation into Mezzo of the standard definition of list concatenation in ML. It has one major drawback: it is not tail-recursive, which means that it needs a linear amount of space on the stack, and may well run out of space if the operating system places a low limit on the size of the stack.

One 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 `xs @ Cons { head: a; tail: 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:

```
dst @ Cell { head: a; tail: () } *
xs @ Cons { head = head; tail = tail } *
head @ a *
tail @ list a *
ys @ list a
```

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

```
dst' @ Cell { head = head; tail: () }
```

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

The assignment of line 15 and the tag update of line 16 are reflected by updating the structural permission that describes the cell `dst`. Before the assignment, we have `dst @ Cell { head: a; tail: () }`. After the assignment `dst.tail <- dst'`, we have `dst @ Cell { head: a; tail = dst' }`. After the tag update instruction `tag **of** dst <- Cons`, we have we have `dst @ Cons { head: a; tail = dst' }`. It may be worth stressing that the last two structural permissions can be folded neither to `dst @ cell a` nor to `dst @ 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:

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

```
head @ a *
tail @ list a *
ys @ 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' } *
xs @ 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 Charguéraud [Charguéraud 2010, §3.3.2] and by Tuerk [2010].

2.3. A higher-order function

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

The implementation appears in Fig. 8. A stack is defined as a mutable reference to a list of elements. Here, we use traditional ML references, which are allocated with `newref`[2], 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 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

[2]This function is named `newref`, instead of `ref` in ML. Indeed, the type of references is called `ref` already,

```
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

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 find, which scans a list xs and returns the first element x (if there is one) that satisfies a user-provided predicate ok. Translit-

and Mezzo places types and values in a single namespace.

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

```
val find: [a] (xs: list a, ok: a -> bool) -> 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 `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 `find` the following type, where the list is consumed:

```
val find: [a] (
  consumes xs: list a,
  ok: a -> bool
) -> 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 `find`.

Another tentative solution is to give `find` the following type, where the type parameter `a` is required to be duplicable:

```
val find: [a] (
  xs: list a,
  ok: a -> bool
| duplicable a
) -> option a
```

Naturally, this does not solve the problem. This means that `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 `find` to return the desired element, we provide `find` with a function `f` that describes what we want to do with this element. The signature of `find` thus becomes:

```
val find: [a] (
  xs: list a,
  ok: a -> bool,
  f: a -> ()
) -> ()
```

Recall that, in Mezzo, a function argument that is *not* annotated with the keyword **consumes** is preserved: that is, the function requires and returns a permission for this argument. Thus, this version of `find` preserves `xs @ 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 @ list a`. If it did, the ownership problem would arise again!

A moment's thought reveals that the above signature for `find` is not as simple as it could be. The user is asked to provide *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 `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:

```
val exists: [a] (
  xs: list a,
  ok: a -> bool
) -> bool
```

We have changed the name of the function from `find` to `exists`, because it is now identical to the function known as `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 `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.

```
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 `find` and `exists`. Here, in an attempt to express `find` in Mezzo, we have ended up with `exists`, a higher-order function. We still do not have a satisfactory way of expressing `find` as a first-order function.

A 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 `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`.

```
1  data result (p: perm) a =
2    | NotFound { | p }
3    | Found    { found: a }
4
5  alias wand (pre: perm) (post: perm) =
6    {ammo: perm} (
7      (| consumes (pre * ammo)) -> (| post)
8    | ammo)
9
10 alias focused a (post: perm) =
11   (x: a, w: wand (x @ a) post)
12
13 val rec find [a] (consumes xs: list a, pred: a -> bool)
14 : result
15     (xs @ list a)
16     (focused a (xs @ list a))
17 = match xs with
18   | Nil ->
19       NotFound
20   | Cons { head; tail } ->
21       if pred head then begin
22         let w (| consumes (head @ a * tail @ list a))
23             : (| xs @ list a) = () in
24         Found { found = (head, w) }
25       end
26       else begin
27         let r = find (tail, pred) in
28         match r with
29         | NotFound ->
30             r
31         | Found { found = (x, w) } ->
32             let flex guess: perm in
33             pack w @ wand (x @ a) (xs @ list a)
34                  witness (head @ a * guess);
35             r
36         end
37       end
38   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.[3] 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 `find`.

At line 19, we have reached the end of the list. We return `NotFound`. By examining the return type of `find` (lines 14–16) as well as the definition of the algebraic data type `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 `Found` applied to the pair `(head, w)`, which should (therefore) have type `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:

```
w @ wand (head @ a) (xs @ list a)
```

The definition of `w` at lines 22–23 gives rise to the permission:

```
w @ (| consumes (head @ a * tail @ list a)) -> (| xs @ list a)
```

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

The type-checker must also verify that the definition of `w` at lines 22–23 is valid. This is indeed the case because `w` has access not only to `head @ a * tail @ list a`, but also to the duplicable permission `xs @ Cons { head = head; 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 `find` returns `NotFound`. Even though the code is terse, the reasoning is nontrivial. As we are in the `NotFound` branch, we have `tail @ list a`. Furthermore, we still hold `head @ a` and `xs @ Cons { head = head; 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 `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 @ a`, and a wand `w` that we are supposed to use when we are done with `x`. This wand has type `wand (x @ a) (tail @ list a)`. At lines 33–34, we argue that it also has type `wand (x @ a) (xs @ list a)`[4]. Combined with the structural permission `r @ Found { found = (x, w) }` obtained at line 28, this implies that `r` has type `result (xs @ list a) (focused a (xs @ 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 `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 `find` contains a redundant case analysis, which ideally should be unnecessary. Indeed, because **`let flex`** and **`pack`** have no runtime effect, the entire **`match`** construct at lines 28–36 is equivalent to just `r`. If we could replace this construct with just `r`, the code would be 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).

[4] 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 `wand (x @ a) (tail @ list a)`. This is an abbreviation for an existential type, which is automatically unpacked. Thus, `w` is now viewed as a function of type `(| `**`consumes`**` (x @ a * ammo)) -> (| tail @ list a)`, where `ammo` is a fresh abstract permission; and one copy of `ammo` is now available. Now, by the frame rule (FRAMESUB, Fig. 26), one can add a permission to the domain and codomain of a function. Here, by adding `head @ a` to the domain and codomain, we find that `w` also has type `(| `**`consumes`**` (x @ a * head @ a * ammo)) -> (| head @ a * tail @ list a)`. In the presence of the duplicable permission `xs @ Cons { head = head; tail = tail }`, the codomain can be further weakened: we find that `w` has type `(| `**`consumes`**` (x @ a * head @ a * ammo)) -> (| xs @ list a)`. This information about `w` and the fact that the permission `head @ a * ammo` is available allow us in principle to recognize the definition of a magic wand and to conclude that `w` has type `wand (x @ a) (xs @ list a)`, as desired. This is an existential introduction step, where the witness this time is `head @ a * ammo`. 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 **`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 "`head @ a * ammo`". Because the permission variable `ammo` 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 **`let flex`** construct on line 32. This construct introduces a flexible permission variable, called `guess`. This allows us to supply `head @ a * guess` as the witness on line 34. When examining the **`pack`** construct, the type-checker is able to guess that `guess` must be unified with `ammo`. In principle, we could simplify things slightly by allowing the programmer to supply "`head @ a * _`" as the witness on line 34. The wildcard would be considered syntactic sugar for a flexible variable, obviating the need for an explicit **`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 $x \neq y$ 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.

```
1  data mutable node a =
2    Node {
3      value    : a;
4      visited  : bool;
5      neighbors: list (node a);
6    }
7
8  val _ : node int =
9    let n = Node {
10     value     = 10;
11     visited   = false;
12     neighbors = ();
13   } in
14   let ns = Cons { head = n; tail = Nil } in
15   n.neighbors <- ns;
16   n
```

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 @ Node {
  value     : int;
  visited   : bool;
  neighbors = ns;
} *
ns @ Cons { head = n; tail: Nil }
```

This illustrates the fact that a cycle of statically known length can be described in terms of structural permissions and singleton types. The type error lies on line 16, where (due to the type annotation on line 8) the type-checker must verify that the above permission entails `n @ node int`. This permission subsumption step is invalid. This is the second limitation that was discussed earlier: since `n @ Node { ... }` is affine, it cannot be used to *separately* justify that n is a node and ns is a list of nodes. Furthermore, even if `n @ Node { ... }` was duplicable, this 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.

```
1  data mutable node a =
2    Node {
3      content  : a;
4      visited  : bool;
5      neighbors: list dynamic;
6    }
7
8  data mutable graph a =
9    Graph {
10     roots    : list dynamic;
11   } adopts node a
12
13 val g : graph int =
14   let n = Node {
15     content   = 10;
16     visited   = false;
17     neighbors = ();
18   } in
19   let ns = Cons { head = n; tail = Nil } in
20   n.neighbors <- ns;
21   assert n @ node int * ns @ list dynamic;
22   let g : graph int = Graph { roots = ns } in
23   give n to g;
24   g
25
26 val dfs [a] (g: graph a, f: a -> ()) : () =
27   let s = stack::new g.roots in
28   stack::work (s, fun (n: dynamic
29                      | g @ graph a * s @ stack dynamic) : () =
30     take n from g;
31     if not n.visited then begin
32       n.visited <- true;
33       f n.content;
34       stack::push (n.neighbors, s)
35     end;
36     give n to g
37   )
```

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 a` is that the `neighbors` field now has type `list` **`dynamic`** (line 5).

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

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

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

The ***give*** *and* ***take*** *instructions.* The runtime effect of the *adoption* instruction **`give`** `n` **`to`** `g` (line 23) is that the node `n` becomes a new adoptee of the graph `g`. At the beginning of this line, the permissions `n @ node int` and `g @ graph int` are available. Together, they justify the instruction **`give`** `n` **`to`** `g`. (The type-checker verifies that the type of `g` has an **`adopts`** clause and that the type of `n` is consistent with this clause.) After the **`give`** instruction, at the end of line 23, the permission `n @ node int` has been consumed, while `g @ graph int` remains. A *transfer of ownership* has taken place: whereas the node `n` was "owned by this thread", so to speak, it is now "owned by `g`". 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 (§7), 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 (§7.10).

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 §7.9 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 `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 `dfs` initializes the stack (line 27) and enters a loop, encoded as a call to the higher-order function `stack::work`. At each iteration, an element `n` is taken out of the stack; it has type **`dynamic`** (line 28). Thus, the type-checker does not know a priori that `n` is a node. The **`take`** instruction (line 30) recovers this information. It is justified by the permissions `n @` **`dynamic`** and `g @ graph int` and (if successful) produces `n @ 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 **`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 **`give`** would lead to a runtime failure at a later **`take`** instruction, typically on line 30. In order to diminish the likelihood of this particular mistake, we propose **`taking`** `n` **`from`** `g` **`begin ... end`** as syntactic sugar for a well-parenthesized use of **`take`** and **`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 **`give`** and **`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, **`give`** and **`take`** can be implemented cheaply, using normal read and write instructions. Although a data race exists, it is benign (§7.9).

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 **`take`**-ing it away from its adopter), and they allow **`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.

```
1 abstract nests (y : value) (p : perm) : perm
2 fact duplicable (nests y p)
3
4 val nest: [y : value, p : perm]
5   (| consumes p) -> (| nests y p)
6
7 abstract punched (a : type) (p : perm) : type
8
9 val focus: [y : value, p : perm, a] exclusive a =>
10   (| nests y p * consumes y @ a) -> (| p * y @ punched a p)
11
12 val defocus: [y : value, p : perm, a] exclusive a =>
13   (| consumes (p * y @ punched a p)) -> (| y @ a)
```

Fig. 13. Axiomatization of nesting in Mezzo

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 `nests` (line 1). It is an abstract permission, that is, an abstract type of kind **perm**. It is parameterized with a value `y` and a permission `p`. The permission `nests y p` means that `p` has been "nested" in `y`, or delegated to `y`. In other words, whoever has (exclusive) ownership of the object `y` also (implicitly) possesses the permission `p`.

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

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

The permission `nests y p` is duplicable (line 2). In other words, the fact that `p` has been nested in `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 **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, **give** and **take**, allow delegating the ownership of some object `x` to an adopter `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 `y`. Two operations, `nest` and `defocus`, correspond to **give**. One operation, `focus`, corresponds to **take**. We stress, once more, that these operations have no runtime effect.

A new nesting relationship is established by a call to `nest` (line 4). Such a call takes the permission `p` away from the caller and transfers it to the object `y`, or in other words, to whoever owns `y`. Thus, the caller loses `p` and gains the (duplicable) nesting witness `nests 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 `nests y p`. Furthermore, exclusive ownership of `y` is needed. This is expressed by requiring `y @ a`, for an arbitrary exclusive type `a`.[5] 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 @ punched a p`, which is not exclusive (hence, does not allow `focus`ing on `y` again).

The permission `y @ punched a p` 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[6]. 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 `region` (line 3). Like the unit type `()`, it has just one data constructor, of arity zero. Unlike the `unit` type, it is declared **`mutable`**, which implies that a region has a unique owner and that the permission `r @ 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 **`unknown`** is surface syntax for $\top$, that is, a duplicable type that carries no information.) Thus, by definition, the permission `x @ inhabitant r a` 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

[5]Like **`duplicable`** `a`, **`exclusive`** `a` is an assertion about the type `a`, and can be viewed as a permission. It is not formalized in this paper. Intuitively, **`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 `focus`ed. It is not synonymous with **`affine`** `a`, which in Mezzo would be true of every type `a`.

[6]The operation `nest` corresponds roughly to transformation 7 in Boyland's Theorem 6.4 [2010]. The operations `focus` and `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
  let ns = Cons { head = n; tail = 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 `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 @ region`.

We define `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 `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 **give** instruction, placed after the assignment `n.neighbors <- ns`, here we must use `nest` before the assignment (line 30) and use `focus` and `defocus` afterwards (lines 31 and 33). The reason why we can get away in Fig. 12 with just one **give** instruction is that the permission `ns @ list` **dynamic** already exists before we give `n` to `g`. (Every node has type **dynamic** at every time, regardless of which **give** and **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 `ns` has type `list (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 @ node r int`. And, to do this, we need `n.neighbors` to have type `list (inode r a)`. We work around this circularity by initializing `n.neighbors` with the empty list. This allows us to nest `n` in `r`, which is good, but unfortunately takes the permission `n @ node r int` away from us. We temporarily recover this permission via `focus` and `defocus`, which allows us to justify the assignment `n.neighbors <- ns`.

The call to `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 `focus` and `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 @ node r 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 **give** and **take** operations, we use a pair of `focus` and `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: `inode r a` is arguably a more satisfactory piece of information than **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 **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 (§4–§7) is expressed in terms of a simpler *core syntax*. The type-checker translates surface syntax down to core syntax, and performs the bulk of the type-checking work at that level.

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

As far as types and permissions are concerned, there are two differences between surface syntax and core syntax. One difference is that the surface syntax offers a *name introduction* construct `x: t`, together with a set of rules that dictate the scope of the name `x`. This construct does not exist in the core syntax, where we only have more 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 §4. Then, we proceed to give a precise definition of the translation. In §3.2, we define the (combined) surface and core syntaxes of types and permissions. We give a well-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] (consumes xs: list a) ->
    (x: a, wand (x @ a) (xs @ list a))
```

The name introduction construct `xs: 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 @ list a`) is bound by the name introduction.

The codomain of this function type is a pair `(..., ...)`. The left-hand component of this pair is another name introduction construct `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 : value]
    (consumes (=xs | xs @ list a)) ->
    {x : value}
    ((=x | x @ a), wand (x @ a) (xs @ 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: list a` is now replaced with `(=xs | xs @ list a)`. This is a conjunction of a (singleton) type and a permission. This means that the function `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 `find ys` is well-typed, provided the caller is able to provide the permission `ys @ list a`. Similarly, the name introduction `x: a` is replaced with `(=x | 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 `(x: t, y: u)`, 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:

$$\begin{array}{l} \forall(a : \mathtt{type}) \\ \forall(xs : \mathtt{value}) \\ (=xs \mid xs \mathbin{@} \mathtt{list}\ a) \rightarrow \\ \exists(x : \mathtt{value}) \\ ((=x \mid x \mathbin{@} a), \mathtt{wand}\ (x \mathbin{@} a)\ (xs \mathbin{@} \mathtt{list}\ a)) \end{array}$$

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

```
[a] (set a, consumes set a) -> ()
```

The domain of this function type is a pair, whose second component is marked with the keyword **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:

$$\begin{array}{l} \forall(a : \mathtt{type}) \\ \forall(x : \mathtt{value}) \\ (=x \mid x \mathbin{@} (\mathtt{set}\ a, \mathtt{set}\ a)) \rightarrow \\ (() \mid x \mathbin{@} (\mathtt{set}\ a, \top)) \end{array}$$

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

3.2. Well-kindedness

The combined surface and core syntaxes of Mezzo are presented in Fig. 15. The surface syntax has the function type $T_1 \rightsquigarrow T_2$, which is exposed to the user under the ASCII

κ	$::=$ `value` $\mid$ `type` $\mid$ `perm` $\mid \ldots$	(Kinds)
T, P	$::=$	(Types and permissions)
	$\mid x$	(Variable)
	$\mid =x$	(Singleton type)
	$\mid T \rightarrow T$	(Function type (core))
	$\mid (T \mid P)$	(Type/permission conjunction)
	$\mid (T, T)$	(Pair type)
	$\mid x @ T$	(Atomic permission)
	$\mid$ `empty`	(Empty permission)
	$\mid P * P$	(Permission conjunction)
	$\mid$ `duplicable` T	(Duplicability assertion)
	$\mid \forall(x : \kappa)\ T$	(Universal quantification)
	$\mid \exists(x : \kappa)\ T$	(Existential quantification)
	$\mid T \rightsquigarrow T$	(Function type (surface))
	$\mid x : T$	(Name introduction (surface))
	$\mid$ `consumes` T	(Consumes annotation (surface))

Fig. 15. Types and permissions: combined core and surface syntaxes

$\texttt{names}(x : T)$	$=$	$\{(x, \texttt{value})\} \uplus \texttt{names}(T)$	(Name introduction)
$\texttt{names}(T_1, T_2)$	$=$	$\texttt{names}(T_1) \uplus \texttt{names}(T_2)$	(Pair type)
$\texttt{names}(T \mid P)$	$=$	$\texttt{names}(T)$	(Type/permission conjunction)
$\texttt{names}(\texttt{consumes}\ T)$	$=$	$\texttt{names}(T)$	(Consumes annotation)
$\texttt{names}(T)$	$=$	$\emptyset$	(Any other type)

Fig. 16. Name collection function

$$\text{K-OPENNEWSCOPE}\quad \frac{\Gamma; \texttt{names}(T) \vdash T : \kappa}{\Gamma \vdash \#T : \kappa}$$

Fig. 17. Types and permissions: well-kindedness (auxiliary judgement)

form `T1 -> 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-kindedness judgement checks (among other things) that every name is properly bound. Thus, in a slightly indirect way, it defines the scope of every name. In addition to the universal and existential quantifiers, which are perfectly standard, Mezzo offers the name introduction construct $x : T$, which is 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-kindedness 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 $\texttt{names}(T)$, 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-kindedness judgement $\Gamma \vdash T : \kappa$ means that under the kind assumptions in Γ, the type T has kind κ. Its definition (Fig. 18) relies on an auxiliary judgement, $\Gamma \vdash \#T : \kappa$, which by definition means $\Gamma; \texttt{names}(T) \vdash T : \kappa$ (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

$$\textsc{K-Var}\ \frac{(x,\kappa)\in\Gamma}{\Gamma\vdash x:\kappa}\qquad \textsc{K-Sing}\ \frac{\Gamma\vdash x:\mathtt{value}}{\Gamma\vdash {=}x:\mathtt{type}}\qquad \textsc{K-InternalArrow}\ \frac{\Gamma\vdash T_1:\mathtt{type}\qquad \Gamma\vdash T_2:\mathtt{type}}{\Gamma\vdash T_1\to T_2:\mathtt{type}}\qquad \textsc{K-Bar}\ \frac{\Gamma\vdash T:\mathtt{type}\qquad \Gamma\vdash P:\mathtt{perm}}{\Gamma\vdash (T\mid P):\mathtt{type}}$$

$$\textsc{K-Pair}\ \frac{\Gamma\vdash T_1:\mathtt{type}\qquad \Gamma\vdash T_2:\mathtt{type}}{\Gamma\vdash (T_1,T_2):\mathtt{type}}\qquad \textsc{K-Atomic}\ \frac{\Gamma\vdash x:\mathtt{value}\qquad \Gamma\vdash \#T:\mathtt{type}}{\Gamma\vdash x\,@\,T:\mathtt{perm}}\qquad \textsc{K-Empty}\ \Gamma\vdash \mathtt{empty}:\mathtt{perm}$$

$$\textsc{K-Conjunction}\ \frac{\Gamma\vdash P_1:\mathtt{perm}\qquad \Gamma\vdash P_2:\mathtt{perm}}{\Gamma\vdash P_1 * P_2:\mathtt{perm}}\qquad \textsc{K-Duplicable}\ \frac{\Gamma\vdash \#T:\kappa\qquad \kappa\in\{\mathtt{type},\mathtt{perm}\}}{\Gamma\vdash \mathtt{duplicable}\ T:\mathtt{perm}}\qquad \textsc{K-Quantifier}\ \frac{\Gamma;(x,\kappa')\vdash \#T:\kappa}{\begin{array}{c}\Gamma\vdash\forall(x:\kappa')\ T:\kappa\\ \Gamma\vdash\exists(x:\kappa')\ T:\kappa\end{array}}$$

$$\textsc{K-ExternalArrow}\ \frac{\begin{array}{c}\Gamma;\mathtt{names}(T_1)\vdash\ \ T_1:\mathtt{type}\\ \Gamma;\mathtt{names}(T_1)\vdash \#T_2:\mathtt{type}\end{array}}{\Gamma\vdash T_1\rightsquigarrow T_2:\mathtt{type}}\qquad \textsc{K-NameIntro}\ \frac{\begin{array}{c}\Gamma\vdash x:\mathtt{value}\\ \Gamma\vdash T:\mathtt{type}\end{array}}{\Gamma\vdash (x:T):\mathtt{type}}\qquad \textsc{K-Consumes}\ \frac{\begin{array}{c}\Gamma\vdash T:\kappa\\ \kappa\in\{\mathtt{type},\mathtt{perm}\}\end{array}}{\Gamma\vdash \mathtt{consumes}\ T:\kappa}$$

Fig. 18. Types and permissions: well-kindedness

$$\textsc{T1-OpenNewScope}\ \frac{T\blacktriangleright T'}{\#T\blacktriangleright \exists(\mathtt{names}(T))\ T'}$$

Fig. 19. Types and permissions: first translation phase (auxiliary judgement)

$$\textsc{T1-Var}\ x\blacktriangleright x\qquad \textsc{T1-Sing}\ {=}x\blacktriangleright {=}x\qquad \textsc{T1-Bar}\ \frac{T\blacktriangleright T'\qquad P\blacktriangleright P'}{(T\mid P)\blacktriangleright (T'\mid P')}\qquad \textsc{T1-Pair}\ \frac{T_1\blacktriangleright T_1'\qquad T_2\blacktriangleright T_2'}{(T_1,T_2)\blacktriangleright (T_1',T_2')}\qquad \textsc{T1-Atomic}\ \frac{\#T\blacktriangleright T'}{x\,@\,T\blacktriangleright x\,@\,T'}$$

$$\textsc{T1-Empty}\ \mathtt{empty}\blacktriangleright\mathtt{empty}\qquad \textsc{T1-Conjunction}\ \frac{P_1\blacktriangleright P_1'\qquad P_2\blacktriangleright P_2'}{P_1 * P_2\blacktriangleright P_1' * P_2'}\qquad \textsc{T1-Duplicable}\ \frac{\#T\blacktriangleright T'}{\mathtt{duplicable}\ T\blacktriangleright \mathtt{duplicable}\ T'}\qquad \textsc{T1-Quantifier}\ \frac{\#T\blacktriangleright T'}{\begin{array}{c}\forall(x:\kappa)\ T\blacktriangleright\forall(x:\kappa)\ T'\\ \exists(x:\kappa)\ T\blacktriangleright\exists(x:\kappa)\ T'\end{array}}$$

$$\textsc{T1-ExternalArrow}\ \frac{T_1\blacktriangleright T_1'\qquad \#T_2\blacktriangleright T_2'}{T_1\rightsquigarrow T_2\blacktriangleright \forall(\mathtt{names}(T_1))\ T_1'\rightsquigarrow T_2'}\qquad \textsc{T1-NameIntro}\ \frac{T\blacktriangleright T'}{x:T\blacktriangleright ({=}x\mid x\,@\,T')}\qquad \textsc{T1-Consumes}\ \frac{T\blacktriangleright T'}{\mathtt{consumes}\ T\blacktriangleright \mathtt{consumes}\ T'}$$

Fig. 20. Types and permissions: first translation phase

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-kindedness 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-kindedness judgement with a Boolean parameter, which indicates whether **consumes** is allowed or disallowed. In order to reduce clutter, we omit this aspect.

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

$$\frac{\begin{array}{cc} T_1 \triangleright T_1' & T_2 \triangleright T_2' \\ T_1'^{in} = [T/\texttt{consumes}\ T]T_1' & T_1'^{out} = [?/\texttt{consumes}\ T]T_1' \end{array}}{T_1 \rightsquigarrow T_2 \triangleright \forall(x : \texttt{value})\ (=x \mid x \mathbin{@} T_1'^{in}) \rightarrow (T_2' \mid x \mathbin{@} T_1'^{out})}\ \text{T2-ExternalArrow}$$

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

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 $T \blacktriangleright T'$, whose definition (Fig. 20) relies on the auxiliary judgement $\#T \blacktriangleright T'$ (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 $x \mathbin{@} T'$. The two occurrences of x in this conjunction are free: they refer to a quantifier that has necessarily been introduced higher up by T1-OpenNewScope or T1-ExternalArrow.

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

— $\Gamma \vdash T : \kappa$ *and* $T \blacktriangleright T'$ *imply* $\Gamma \vdash T' : \kappa$.
— $\Gamma \vdash \#T : \kappa$ *and* $\#T \blacktriangleright T'$ *imply* $\Gamma \vdash T' : \kappa$.

Claim 3.2. *If* $T \blacktriangleright T'$ *or* $\#T \blacktriangleright T'$ *holds, then* T' *does not contain a name introduction construct.*

Phase 2. The second phase is described by the translation judgement $T \triangleright T'$, whose definition appears in Fig. 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 $\rightsquigarrow$ into a core arrow $\rightarrow$. 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_1', it constructs distinct pre- and postconditions, namely $x \mathbin{@} T_1'^{in}$ and $x \mathbin{@} T_1'^{out}$. These permissions respectively represent the properties of x that the function requires (prior to the call) and ensures (after the call).

The type $T_1'^{in}$ is defined as $[T/\texttt{consumes}\ T]T_1'$. 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 $T_1'^{out}$ is defined as $[?/\texttt{consumes}\ T]T_1'$. By this informal notation, we mean "a copy of T_1' where every subterm of the form consumes T is replaced with $\exists(x : \kappa)\ x$, where the kind κ is either `type` or `perm`, as appropriate". We note that $\exists(x : \kappa)\ x$ is a "top" type or permission: it does not provide any useful information.

Thus, the permission $x \mathbin{@} T_1'^{in}$ represents the ownership of the argument, including the components marked with **consumes**, whereas the permission $x \mathbin{@} T_1'^{out}$ represents the ownership of the argument, deprived of these components.

Claim 3.3. *Well-kindedness 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. KERNEL

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

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

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

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

In order to avoid any confusion, we must emphasize the fact that our Coq code is, strictly speaking, not modular. The kernel and its extensions are not independent artifacts: they cannot be independently type-checked and later brought together. In reality, the Coq code is monolithic: there is a single inductive type of the syntax of Core Mezzo, which includes the kernel and its extensions. Still, the code is "pseudo-modular" in the sense that the presence or absence of one extension in principle has little to no impact on the code of the kernel or of the other extensions. We briefly come back to this issue when we discuss the related work (§10).

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

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

4.1. Machine states and resources

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

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

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

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

At this stage, again, the nature of resources is unspecified. One should think of a resource as a partial, instrumented machine state.

A resource is "partial" because it represents possibly incomplete knowledge about the machine state. A heap fragment in the style of Separation Logic [Reynolds 2002] is an example of a partial resource: about a memory location in its domain, it contains precise information (i.e., it indicates which value is stored there); about a memory location outside its domain, it contains no information at all.

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

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

— A composition operator $\star$ allows two resources (i.e., the views of two threads) to be combined. It is total, commutative, and associative.
— A consistency predicate, R *ok*, identifies the well-formed resources. It is preserved by splitting, i.e., $R_1 \star R_2$ *ok* implies R_1 *ok*.
— A total function $\widehat{\cdot}$ maps every resource R to its *core* $\widehat{R}$, which represents the duplicable (shareable) information contained in R.
  — This element is a unit for R, i.e., $R \star \widehat{R} = R$.
  — Two compatible elements have a common core, i.e., $R_1 \star R_2 = R$ and R *ok* imply $\widehat{R_1} = \widehat{R}$.
  — A duplicable resource is its own core, i.e., $R \star R = R$ implies $R = \widehat{R}$.
  — Every core is duplicable, i.e., $\widehat{R} \star \widehat{R} = \widehat{R}$.
— A relation $R_1 \lhd R_2$, the *rely*, represents the interference that "other" threads are allowed to inflict on "this" thread, by specifying how a known resource R_1 can evolve into a resource R_2. This relation typically allows "other" threads to allocate new memory blocks, or new locks. The type discipline is defined in such a way that, if $R_1 \lhd R_2$ holds, then a typing judgement that assumes R_1 can be transformed into a typing judgement that assumes R_2. (This is Stability, Lemma A.5 in Appendix A.)
  — This relation is reflexive.
  — It preserves consistency, i.e., R_1 *ok* and $R_1 \lhd R_2$ imply R_2 *ok*.
  — It is preserved by core, i.e., $R_1 \lhd R_2$ implies $\widehat{R_1} \lhd \widehat{R_2}$.
  — Finally, it is compatible with $\star$, in the following sense:

$$\frac{R_1 \star R_2 \lhd R' \qquad R_1 \star R_2 \; ok}{\exists R_1' R_2',\, R_1' \star R_2' = R' \wedge R_1 \lhd R_1' \wedge R_2 \lhd R_2'}$$

  This means that an evolution of a composite resource can always be explained as evolutions of the components.

$$
\begin{array}{rcll}
\kappa & ::= & \mathtt{value} \mid \mathtt{term} \mid \mathtt{soup} \mid \mathtt{type} \mid \mathtt{perm} & \text{(Kinds)} \\[1ex]
v, t, u, T, U, P, Q, \theta & ::= & x & \text{(Everything)} \\
& \mid & \lambda x.t & \text{(Values: } v\text{)} \\
& \mid & v\ t \mid \mathtt{spawn}\ v\ v & \text{(Terms: } t, u\text{)} \\
& \mid & \mathtt{thread}\ (t) \mid t \parallel t & \text{(Soups: } t\text{)} \\
& \mid & {=}v \mid T \to T \mid (T \mid P) & \text{(Types: } T, U\text{)} \\
& \mid & v \mathbin{@} T \mid \mathtt{empty} \mid P * P \mid \mathtt{duplicable}\ \theta & \text{(Permissions: } P, Q\text{)} \\
& \mid & \forall x : \kappa.\theta \mid \exists x : \kappa.\theta & \text{(Types or permissions: } \theta\text{)} \\[1ex]
E & ::= & v\ [] & \text{(Shallow evaluation contexts)} \\
D & ::= & [] \mid E[D] & \text{(Deep evaluation contexts)}
\end{array}
$$

Fig. 22. Kernel: syntax of programs, types, and permissions

The consistency predicate is of no inherent interest, but plays a technical role of easing the reasoning about combinations of resources. Indeed, at the level of Coq it is important that $\star$ be total and (unconditionally) commutative and associative. This remark has been formulated by others before [Nanevski et al. 2010]. Now for $\star$ to be total, it must always produce some result, even in situations where its two arguments cannot be meaningfully combined. In such a case, it produces an *inconsistent* result, i.e., a resource R such that R *ok* does not hold. This allows us to reason relatively easily about combinations of resources; in particular, we rely on Braibant and Pous' plug-in [2011] for proving equalities and performing matching up to AC.

The above axiomatization is identical to that found in Pottier's previous work [2013, §10], up to a few technical simplifications. In particular, we are able to get away with just one relation on resources, $\lhd$, whereas the previous paper used two.

We note that this axiomatization is not necessary or canonical. It is just one possible axiomatization that suffices for the purposes of the type soundness proof. Perhaps it could be further simplified or generalized.

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

4.2. Syntax

Values, terms, soups (parallel compositions of several threads), types, and permissions form a single syntactic universe, defined in Fig. 22. There is also a single name space of variables.

The types and permissions in Fig. 22 are the same as in Fig. 15, except we omit the constructs that are specific to the surface syntax, and (for simplicity) we omit pairs.

Within this universe, we impose a kind discipline, so as to distinguish several syntactic categories: types, terms, etc. We find this approach particularly pleasant, as it obviates the need for a quadratic number of weakening and substitution lemmas (type-in-type, term-in-type, term-in-term, etc.).

For the sake of conciseness, we omit the (fairly mundane) definition of the well-kindedness judgement. The part that concerns types and permissions can be deduced from Fig. 18, while a complete definition can be found in the Coq code [Balabonski and Pottier 2014].

Furthermore, throughout the paper, we hide the well-kindedness premises in every typing rule and theorem. Instead, we use conventional metavariables (v, t, etc.) to indicate the intended kind of each syntactic element. We note that these well-kindedness premises also encode which variables are allowed to occur in which types or terms. Thus, by hiding them, we effectively adopt Barendregt's convention. In the Coq proof, naturally, everything is formal: names are encoded as de Bruijn indices.

There are five kinds, or syntactic categories (Fig. 22).

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

The terms t have kind `term`. Every value is a term. Function application $v\ t$ and thread creation `spawn` $v_1\ v_2$ are also terms (the latter is meant to execute the function call $v_1\ v_2$ in a new thread). The sequencing construct `let` $x = t_1$ `in` t_2 is encoded as $(\lambda x.t_2)\ t_1$.

By requiring the left-hand side of an application to be a value, and by requiring the arguments of `spawn` to be values, we reduce the number of evaluation contexts to one. The only (shallow) evaluation context is the right-hand side of an application, $v\ []$. This does not cause any loss of expressiveness: for instance, the application $t_1\ t_2$ can be encoded as `let` $x = t_1$ `in` $x\ t_2$. The Mezzo type-checker performs this transformation (which makes the evaluation order explicit, and names every intermediate result) on the fly.

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

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

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

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

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

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

Core Mezzo does not have type annotations. A term cannot refer to a type or permission[7]. In other words, the syntax of terms is untyped. This means that the operational semantics can be defined without any reference to the type discipline: Core Mezzo enjoys type erasure. Naturally, this makes type-checking undecidable: an (informal) argument is that Core Mezzo contains System F, whose type-checking problem, in the absence of any type annotation, is undecidable already. In the present paper, this is not a problem, as we are interested only in defining the type system and establishing its soundness. In surface Mezzo, some type annotations are necessary: for instance,

[7]In particular, a variable x that appears in a closed term must be λ-bound, as there are no other binding forms in the syntax of terms.

initial configuration		*new configuration*	*side condition*
$s \;/\; (\lambda x.t)\ v$	$\longrightarrow$	$s \;/\; [v/x]t$	
$s \;/\; E[t]$	$\longrightarrow$	$s' \;/\; E[t']$	$s \;/\; t \longrightarrow s' \;/\; t'$
$s \;/\; \texttt{thread}\ (t)$	$\longrightarrow$	$s' \;/\; \texttt{thread}\ (t')$	$s \;/\; t \longrightarrow s' \;/\; t'$
$s \;/\; t_1 \parallel t_2$	$\longrightarrow$	$s' \;/\; t_1' \parallel t_2$	$s \;/\; t_1 \longrightarrow s' \;/\; t_1'$
$s \;/\; t_1 \parallel t_2$	$\longrightarrow$	$s' \;/\; t_1 \parallel t_2'$	$s \;/\; t_2 \longrightarrow s' \;/\; t_2'$
$s \;/\; \texttt{thread}\ (D[\texttt{spawn}\ v_1\ v_2])$	$\longrightarrow$	$s \;/\; \texttt{thread}\ (D[()]) \parallel \texttt{thread}\ (v_1\ v_2)$	

Fig. 23. Kernel: operational semantics

every λ-abstraction must be annotated with its (argument and result) type, and type applications must sometimes be made explicit.

4.3. Operational semantics

The calculus is equipped with a small-step operational semantics. The reduction relation acts on configurations c, which are pairs of a machine state s and a closed term or soup t. In the rules of Fig. 23, the machine state is carried around, but never consulted or modified.

The rules allow interleaved execution of multiple threads. The spawn construct runs the application $v_1\ v_2$ in a new thread and produces a unit value in the original thread. `join` is not a primitive construct; in principle, it can be programmed up using channels, which themselves can be implemented on top of locks [Protzenko 2014a].

4.4. The typing judgement and the permission interpretation judgement

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

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

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

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

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

Definition of the typing judgement. The typing judgement is defined in Fig. 24. The manner in which the turnstiles $\vdash$ and $\diamond$ are used can be summed up as follows: a

$$\textsc{Singleton}\quad R; K; P \diamond v : =v$$

$$\textsc{Frame}\quad \frac{R; K; P \diamond t : T}{R; K; P * Q \diamond t : T \mid Q}$$

$$\textsc{Function}\quad \frac{\widehat{R}; K, x : \mathtt{value}; P * x @ T \vdash t : U}{R; K; (\mathtt{duplicable}\ P) * P \diamond \lambda x.t : T \to U}$$

$$\textsc{ForallIntro}\quad \frac{t \textit{ is harmless} \qquad R; K, x : \kappa; P \diamond t : T}{R; K; \forall x : \kappa.P \diamond t : \forall x : \kappa.T}$$

$$\textsc{ExistsIntro}\quad \frac{R; K; P \diamond v : [U/x]T}{R; K; P \diamond v : \exists x : \kappa.T}$$

$$\textsc{Cut}\quad \frac{R_2; K; P_1 * P_2 \diamond t : T \qquad R_1; K \Vdash P_1}{R_1 \star R_2; K; P_2 \diamond t : T}$$

$$\textsc{ExistsElim}\quad \frac{R; K, x : \kappa; P \vdash t : T}{R; K; \exists x : \kappa.P \vdash t : T}$$

$$\textsc{SubLeft}\quad \frac{K \vdash P_1 \leq P_2 \qquad R; K; P_2 \vdash t : T}{R; K; P_1 \vdash t : T}$$

$$\textsc{SubRight}\quad \frac{R; K; P \vdash t : T_1 \qquad K \vdash T_1 \leq T_2}{R; K; P \vdash t : T_2}$$

$$\textsc{Application}\quad \frac{R; K; Q \vdash t : T}{R; K; (v @ T \to U) * Q \vdash v\ t : U}$$

$$\textsc{Spawn}\quad R; K; (v_1 @ T \to U) * (v_2 @ T) \vdash \mathtt{spawn}\ v_1\ v_2 : \top$$

Fig. 24. Kernel: the typing judgement

canonical derivation must concern a value v (as opposed to an arbitrary term t) and cannot use the rules ExistsElim, SubLeft, or SubRight outside of a λ-abstraction. In other words, a canonical derivation uses only the first six rules of Fig. 24, except within a λ-abstraction, where it may use all of the rules. Canonical derivations are used in the definition of a "semantic" notion of permission subsumption, with respect to which our syntactic notion of permission subsumption is proved sound (§4.5).

The first five rules of Fig. 24 can be viewed as introduction rules: when applied to a value, they define the meaning of the five type constructors. Some of them can also be applied to a term.

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

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

As usual, Function states that a function $\lambda x.t$ has type $T \to U$ if the body t has type U under the assumption that the formal parameter x has type T. Here, one must separately extend the kind environment K with the binding x : value and augment the precondition P with the assumption $x @ T$. The last unusual aspect of this rule is its treatment of duplication. In Mezzo, by convention, every function type is considered duplicable. (We comment on this design choice in §10.) In other words, every function type carries an implicit, built-in "!" modality. This entails a necessary restriction: a permission P that is available at the function definition site is available also in the function body only if P is duplicable. For this reason, in the conclusion of Function, the precondition contains `duplicable` P. The resource R must be treated in the same manner. If R is available at the function definition site, only its duplicable core $\widehat{R}$ is available in the function body.

Applied to a value, ForallIntro is the introduction rule for universal quantification. If we wished to enforce the value restriction [Wright 1995] in its strictest form, then we would restrict this rule to values. However, that would be more restrictive than strictly necessary. We prefer to be more permissive, and to shed more light on the reason why a restriction is necessary. Thus, we allow applying ForallIntro to a term, but only to a so-called “harmless” term. The question is now: exactly how should the class of harmless terms be defined?

It is well-known that the unrestricted combination of weak (shareable) references and polymorphism is unsound [Damas 1985; Tofte 1988; Wright and Felleisen 1994]. This adverse interaction is traditionally avoided by viewing the allocation of a (weak) reference as “harmful” (that is, non-generalizable). Yet, Charguéraud and Pottier [2008] have pointed out that the combination of strong (uniquely-owned) references and polymorphism is sound. Thus, in Mezzo, allocating a (uniquely-owned) mutable memory block is harmless. The danger lies elsewhere: it is in fact inherent in the combination of *hidden state* and polymorphism. This was noted, in a sequential setting, by Pottier [2013]. In Mezzo, hidden state appears when one introduces locks (§6): in short, the `newlock` instruction must be considered “harmful”.

Here are two ways of seeing why `newlock` must be harmful. (1) Equipped with strong (unique) references and locks, one can encode a weak (shared) reference as a strong reference, protected by a lock. If `newref` and `newlock` were both considered harmless, then one could reproduce in Mezzo the unsound interaction between weak references and polymorphism. Yet, `newref` alone is fine. (2) More technically, when a new lock k whose invariant is some permission P is created by `newlock`, the current resource R, which contains a mapping of every lock to its invariant, is extended with a mapping of k to P. However, a resource R must map every lock to a *closed* invariant, otherwise the typing judgement would not even be stable by substitution. Thus, it is necessary, at this point, to ensure that P is closed. This can be guaranteed by viewing `newlock` as harmful. Then, we are certain that `newlock` never executes under ForallIntro, hence the invariant P of the newly-created lock cannot have a free variable. More generally, not just `newlock` itself, but any term t whose execution may lead to executing `newlock`, must be considered harmful. (Note that every value is harmless, as it does not execute. In particular, a λ-abstraction whose body contains a `newlock` instruction is harmless. The function `hide` of Fig. 3 illustrates this.)

This leads us to characterizing the class of *harmless* terms as follows. This class must encompass the values, must be stable by substitution and by reduction, and (in §6) must not contain a term of the form $D[\texttt{newlock}]$. Technically, the predicate “harmless” is inductively defined, but the details of the definition do not matter, as long as these properties are satisfied.

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

Because subtyping can be used to introduce an existential quantifier, one might think that ExistsIntro is redundant and can be eliminated altogether. It is in fact not redundant: in a canonical derivation, its use is permitted, whereas the use of subtyping is forbidden (outside of a λ-abstraction).

Cut moves information between the parameters P and R of a typing judgement. In short, it says, if t is well-typed under the syntactic assumption P_1, then it is well-typed under the resource R_1, provided R_1 justifies P_1. This justification takes the form of a permission interpretation judgement (the second premise). Cut is the only rule in Fig. 24 with two premises. As is standard in an affine type system, the resource that appears in its conclusion, $R_1 \star R_2$, is split between the premises.

ExistsElim is a left-elimination rule for the existential quantifier. (As usual, x must not occur in t or T. In Coq, this is enforced by shifting t and T in the premise.) The universal quantifier is eliminated via SubRight, as $K \vdash \forall x : \kappa.T \leq [U/x]T$ is part of the subtyping relation. The rules SubLeft and SubRight correspond to Hoare's rule of consequence.

$$\textsc{Atomic}\ \frac{R_1; K; P \Vdash v : T \qquad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v @ T} \qquad \textsc{Empty}\ R; K \Vdash \texttt{empty} \qquad \textsc{Star}\ \frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\textsc{Duplicable}\ \frac{\theta \textit{ is duplicable}}{R; K \Vdash \texttt{duplicable}\ \theta} \qquad \textsc{Forall}\ \frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa.P} \qquad \textsc{Exists}\ \frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa.P}$$

Fig. 25. Kernel: the interpretation of permissions

APPLICATION is standard, except in the manner in which the requirements about the subterms v and t are formulated. Whereas the requirement about t is formulated as a premise, the assumption about v appears as part of the precondition. This is convenient (because the rule has only one premise, as opposed to two, we do not have to split R), yet not essential, and a more familiar-looking rule can be derived (Appendix B). In fact, the simply-typed λ-calculus can be encoded in Core Mezzo (Appendix C).

According to SPAWN, the term spawn $v_1\ v_2$ is type-checked just like a function call $v_1\ v_2$, except a unit value is returned in the original thread, and the result of type U is lost.

Definition of the permission interpretation judgement. The judgement $R; K \Vdash P$, whose definition appears in Fig. 25, assigns a meaning to permissions in terms of resources. It is analogous to the interpretation of assertions in separation logic [Reynolds 2002]. The resource R could be an (instrumented) heap fragment, and the permission P can be thought of as a logical assertion that is satisfied, or justified, by R.

There is one introduction rule for each of the type constructors of kind perm. The rules EMPTY, STAR, FORALL, EXISTS are straightforward.

ATOMIC states, roughly, that R justifies $v @ T$ if v has type T under R. (A variant of CUT is built into this rule.) Its first premise is a canonical typing judgement: the rule is applicable only if v has type T "now", i.e., only if there is a derivation of this fact that does not use subsumption outside of a λ-abstraction.

DUPLICABLE defines the meaning of the permission duplicable θ in terms of a meta-level predicate, θ *is duplicable*. The latter is defined by cases over the syntax of the type or permission θ. We omit the full definition. Some of the cases are: a singleton type $=v$ is duplicable; a function type $T \rightarrow U$ is duplicable; a conjunction $T \mid P$ is duplicable if T and P are duplicable; and so on.

It is somewhat unpleasant that the syntactic permission duplicable θ is interpreted via a meta-level predicate θ *is duplicable* which itself is defined in a syntactic manner. One would intuitively prefer a "semantic" definition: for instance, at kind perm, one would expect P *is duplicable* to be defined (roughly) by "for every R, if $R; K \Vdash P$ holds, then $R; K \Vdash P * P$ holds as well". However, one cannot naïvely adopt such a definition, as the definition of $R; K \Vdash P$ would be ill-formed (recursive, but not positive). Perhaps one could solve this problem via some form of step indexing [Birkedal et al. 2011]. We have not investigated this avenue; for the moment, we break the circularity by giving a syntactic definition of θ *is duplicable* and by checking a posteriori that this predicate implies the desired semantic property (Lemma A.4).

4.5. Subsumption

There are two subsumption judgements:

— for permissions, $K \vdash P \leq Q$;
— for types, $K \vdash T \leq U$.

MIXSTARINTROELIM
$(v \,@\, T) * P \equiv v \,@\, T \mid P$

FRAMESUB
$v \,@\, T_1 \to T_2 \le v \,@\, (T_1 \mid P) \to (T_2 \mid P)$

HIDEDUPLICABLEPRECONDITION
$(v \,@\, (T \mid P) \to U) * ((\texttt{duplicable}\ P) * P)$
$\le\ v \,@\, T \to U$

DUPLICATE
$(\texttt{duplicable}\ P) * P \le P * P$

DUPSINGLETON
$\texttt{empty} \le \texttt{duplicable}\ {=}v$

DUPDUP
$\texttt{empty} \le \texttt{duplicable}\ (\texttt{duplicable}\ \theta)$

Fig. 26. Kernel: permission subsumption (a few rules only; $K \vdash$ omitted)

The latter is defined in terms of the former, as follows: by definition, the type subsumption judgement $K \vdash T \le U$ holds if and only if the permission subsumption judgement $K, x : \texttt{value} \vdash x \,@\, T \le x \,@\, U$ holds.

One might wish to define permission subsumption $K \vdash P \le Q$ in a "semantic" way, as follows: "for every R, if $R; K \Vdash P$ holds, then $R; K \Vdash Q$ holds as well". However, the permission interpretation judgement $R; K \Vdash P$ depends (via ATOMIC) on the typing judgement, which itself depends on permission subsumption. So, this definition would be ill-formed (recursive, but not positive).

The fact that the first premise of ATOMIC must be a canonical derivation does not directly eliminate this problem: indeed, a canonical derivation *can* use permission subsumption (under a λ-abstraction). This fact is nevertheless essential. It means, intuitively, that the circularity must go through a λ-abstraction. This allows us to proceed as follows. First, we give an axiomatic (i.e., inductive) definition of permission subsumption. Then, we prove that this axiomatization is sound with respect to the intended "semantic" definition (Lemma A.8). This proof relies crucially on the fact that the first premise of ATOMIC is a canonical derivation.

Although this approach does not yield the largest possible notion of subsumption, we find that, from a pragmatic standpoint, it works well. When some desirable subsumption rule is found to be missing, it can be easily added. This usually gives rise to one new case in the proof of Lemma A.8 and does not affect the rest of the system.

The permission subsumption judgement $K \vdash P \le Q$ is inductively defined by many rules, of which we show just a few (Fig. 26). In every rule, we omit the assumption "$K \vdash$", as the parameter K is used only in the well-kindedness side conditions, which (by convention) we omit everywhere.

Among the rules *not* shown in Fig. 26 are: subsumption is reflexive and transitive; conjunction is commutative, associative, and has `empty` as a unit; every permission is affine (i.e., can be silently discarded); equality of values is reflexive, symmetric, transitive, and a congruence (i.e., equals can be substituted for equals); the universal and existential quantifiers commute with many other type constructors; the universal quantifier can be eliminated, and the existential quantifier can be introduced; each type constructor is contravariant or covariant in each of its parameters.

The rules shown in Fig. 26 are the following.

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

FRAMESUB is a version of the frame rule (FRAME, Fig. 24), stated as a subsumption axiom. It asserts that a function with fewer side effects can be supplied in a context where a function with more side effects is expected.

HIDEDUPLICABLEPRECONDITION states that if some function v has precondition P, and if P is provably duplicable and exists now, then one may pretend that v has no precondition. This allows a closure to capture a duplicable permission *after* it has been

$$\frac{R; \varnothing; \texttt{empty} \vdash t : T}{R \vdash \texttt{thread}\ (t)}\ \textsc{Thread} \qquad \frac{R_1 \vdash t_1 \qquad R_2 \vdash t_2}{R_1 \star R_2 \vdash t_1 \parallel t_2}\ \textsc{Par} \qquad \frac{s \sim R \qquad R \vdash t}{\vdash s \,/\, t}\ \textsc{JConf}$$

Fig. 27. Kernel: typing rules for soups and configurations

constructed, whereas FUNCTION (Fig. 24) allows a closure to capture such a permission *when* it is constructed.

DUPLICATE states that if P is provably duplicable, then P can be turned into $P * P$. The fact that the permission duplicable P appears to be consumed in the process should not be a source of worry: one can in fact derive $K \vdash (\texttt{duplicable}\ P) * P \leq (\texttt{duplicable}\ P) * (P * P)$ by using DUPLICATE (twice!) and DUPDUP.

We note that if Core Mezzo was extended with support for first-class erasable coercions, i.e., extended in such a way that $P \leq Q$ is itself a permission, then duplicable P would be just an abbreviation for $P \leq P * P$, and the rule DUPLICATE would be replaced with a more general form of modus ponens: $K \vdash P * (P \leq Q) \leq Q$. We have not investigated this extension.

DUPSINGLETON, DUPDUP, and a family of similar rules (not shown) produce permissions of the form duplicable θ, where θ denotes a type or a permission. These rules repeat, at the object level, the rules that define the meta-level predicate θ *is duplicable*. DUPDUP is particularly interesting, as it resembles a form of "reflection": the predicate **duplicable** talks about itself.

4.6. Typing judgements for soups and configurations

The typing judgement for soups $R \vdash t$ (Fig. 27, first two rules) ensures that every thread is well-typed (the type of its eventual result does not matter) and constructs the composition of the resources owned by the individual threads. This judgement means that, under the precondition R, the thread soup t is safe to execute.

The typing judgement for configurations $\vdash s \,/\, t$ (Fig. 27, last rule) ensures that the thread soup t is well-typed under some resource R that corresponds to the machine state s. This judgement means that $s \,/\, t$ is safe to execute.

4.7. Type soundness

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

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

LEMMA 4.1 (SUBJECT REDUCTION). *If c_1 reduces to c_2, then $\vdash c_1$ implies $\vdash c_2$.*

LEMMA 4.2 (PROGRESS). *$\vdash c$ implies that c is acceptable.*

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

$$\begin{array}{rcll} v, t, T, P & ::= & \ldots & \text{(Everything)} \\ & | & \ell & \text{(Values: } v\text{)} \\ & | & \texttt{newref}\ v \mid \texttt{!}v \mid v \texttt{ := } v \mid \texttt{ghost} & \text{(Programs: } t\text{)} \\ & | & \texttt{ref}_m\ T & \text{(Types: } T\text{)} \\ m & ::= & \boldsymbol{D} \mid \boldsymbol{X} & \text{(Modes)} \end{array}$$

Fig. 28. References: syntax

initial configuration	*new configuration*	*side condition*
s / $\texttt{newref}\ v$	$\longrightarrow s'$ / *limit* h	$h = \downarrow s \wedge h' = h :: v \wedge s' = h' \uparrow s$
s / $\texttt{!}\ell$	$\longrightarrow s$ / v	$h = \downarrow s \wedge h(\ell) = v$
s / $\ell \texttt{ := } v'$	$\longrightarrow s'$ / $()$	$h = \downarrow s \wedge h(\ell) = v \wedge h' = h[\ell \mapsto v'] \wedge s' = h' \uparrow s$
s / $\texttt{ghost}$	$\longrightarrow s$ / $()$	

Fig. 29. References: operational semantics

5. REFERENCES

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

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

Of highest interest to an "end user" are the extensions of the syntax of terms and types (§5.1), the typing rules for the operations on references (§5.4), and the permission subsumption rules for references (§5.5). The rest concerns the extension of the operational semantics (§5.2, §5.3) and of the type soundness proof (§5.6–5.8).

5.1. Syntax

We extend the syntax as per Fig. 28.

Values now include the memory locations ℓ, which are natural numbers.

Terms now include the three standard primitive operations on references, namely allocating, reading, and writing. We add a special instruction, `ghost`. It represents a "freeze" instruction, which has no runtime effect, but modifies the current permission (see the typing rule Freeze in Fig. 30). This instruction does not specify which block one wishes to freeze: at this level, this is not necessary. In the surface syntax, the programmer must use a tag update instruction (§2.1).

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

5.2. Heaps

A *value heap* (or just a *heap*) h is either $\lightning$ or a function of an initial segment of the natural numbers to values. (In Coq, such a function is represented as a list.) As far as the operational semantics is concerned, a heap is never $\lightning$. We introduce this "error" element because it allows us, later on (§5.6), to equip heaps with the structure of a monotonic separation algebra (MSA, §4.1). A *memory location* is a natural number. We write $\emptyset$ for the empty heap. We write *limit* h for the first unallocated location in the heap h. We write $h :: v$ for the heap that extends h with a mapping of *limit* h

$$\frac{}{R; K; v \,@\, T \vdash \texttt{newref}\ v : \texttt{ref}_m\ T}\ \textsc{NewRef} \qquad \frac{}{R; K; (\texttt{duplicable}\ T) * (v \,@\, \texttt{ref}_m\ T) \vdash\ !v : T \mid (v \,@\, \texttt{ref}_m\ T)}\ \textsc{Read}$$

$$\frac{}{R; K; (v \,@\, \texttt{ref}_X\ T) * (v' \,@\, T') \vdash v \mathrel{\texttt{:=}} v' : \top \mid (v \,@\, \texttt{ref}_X\ T')}\ \textsc{Write} \qquad \frac{}{R; K; v \,@\, \texttt{ref}_X\ T \vdash \texttt{ghost} : \top \mid (v \,@\, \texttt{ref}_D\ T)}\ \textsc{Freeze}$$

Fig. 30. References: typing rules for terms

$$\textsc{DecomposeRef} \quad v \,@\, \texttt{ref}_m\ T \equiv \exists x : \texttt{value}.((v \,@\, \texttt{ref}_m = x) * (x \,@\, T))$$

$$\textsc{UnifyRef} \quad (v \,@\, \texttt{ref}_{m_1} = v_1) * (v \,@\, \texttt{ref}_{m_2} = v_2) \leq (v \,@\, \texttt{ref}_{m_1} = v_1) * (v \,@\, \texttt{ref}_{m_2} = v_2) * (v_1 = v_2)$$

$$\textsc{DupRef} \quad \texttt{duplicable}\ T \leq \texttt{duplicable}\ (\texttt{ref}_D\ T) \qquad \frac{T \leq U}{v \,@\, \texttt{ref}_m\ T \leq v \,@\, \texttt{ref}_m\ U}\ \textsc{CoRef}$$

Fig. 31. References: subsumption rules

$$\frac{R_1; K \Vdash v \,@\, T \qquad \downarrow R_2(\ell) = m\, v}{R_1 \star R_2; K; P \diamond \ell : \texttt{ref}_m\ T}\ \textsc{Ref}$$

Fig. 32. References: typing rules for values

to the value v. If the memory location ℓ is in the domain of h, then $h[\ell \mapsto v]$ is the heap that maps ℓ to v and agrees with h elsewhere. Later in the paper, we use the same notation for other kinds of heaps: for instance, an instrumented value heap maps memory locations to instrumented values (§5.6).

5.3. Operational semantics

In the kernel calculus (§4), the nature of machine states was completely unspecified. At this point, we need a machine state s to contain at least a heap h. We specify that a machine state is a tuple of several components, one of which is a heap: $s ::= (\ldots, h, \ldots)$. If s is a machine state, we write $\downarrow s$ (pronounced "get") for its "heap" component. If h is a heap and s is a machine state, we write $h{\uparrow}s$ (pronounced "set") for the machine state obtained by updating the "heap" component of s with h.

The reduction rules for references are standard, up to the noise introduced by the conversions between machine states and heaps (Fig. 29). The expression `newref` v expands the heap with a new binding of *limit* h (the first unallocated memory location) to the value v, and reduces to the memory location *limit* h. The expression $!\ell$ looks up the value stored at location ℓ in the heap. The expression $\ell \mathrel{\texttt{:=}} v'$ stores the value v' at location ℓ. The instruction `ghost` does nothing: it reduces in one step to the unit value.

5.4. Assigning types to terms

The typing rules for the operations on references appear in Fig. 30.

According to NewRef, a memory allocation expression `newref` v consumes $v \,@\, T$ and produces a memory location of type $\texttt{ref}_m\ T$. The mode m is arbitrary[8]. If m is *X*, it can be later changed to *D* by freezing this reference.

One could restrict NewRef to the case where T is the singleton type $=v$. In that case, the precondition $v \,@\, {=v}$ is a tautology, and the postcondition $\texttt{ref}_m\ {=v}$ is an ex-

[8]In surface Mezzo, the data constructor determines m. For instance, in the memory allocation expression `Cons { head = x; tail = xs }`, the mode is *D*, because the data constructor `Cons` is part of an immutable algebraic data type.

act description of the newly allocated memory block. This echoes our comment about line 111 in §2.2. Restricting NEWREF in this manner does not cause any loss of expressive power[9].

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

WRITE requires an exclusive permission $x \,@\, \mathtt{ref}_X\ T$, which ensures not only that x is a valid memory location and stores a value of type T, but also that "nobody else" knows about (or has access to) x. The rule allows strong update: the type of x changes to $\mathtt{ref}_X\ T'$, where T' is the type of v'. Again, one could without loss of expressive power restrict WRITE to the case where T is a singleton type and T' is the singleton type $=v'$. This is illustrated by the assignment on line 15 of Fig. 7 and its explanation.

FREEZE states that the instruction `ghost` can transform $v \,@\, \mathtt{ref}_X\ T$ into $v \,@\, \mathtt{ref}_D\ T$.

Following Charguéraud and Pottier [Charguéraud and Pottier 2008], we view all operations on references, including allocation of mutable memory, as harmless. This means that they can be used under JFORALLINTRO (§4.4), or in other words, they are "generalizable". Although we formalize only references here, this remains valid when Core Mezzo is extended with full-blown memory blocks, containing a tag and multiple fields (§8). For instance, the expression `Nil` (Fig. 6) may be formally considered as the allocation of a memory block with zero fields. Because memory allocation is harmless, `Nil` admits the polymorphic type `[a] list a`. The allocation of a mutable reference can be generalized too: for instance, in surface syntax, the expression `newref Nil` has type `[a] ref (list a)`. This typing judgement is safe. The classic scenario where (first) someone writes a list of apples into this reference and (later on) someone else reads the reference, expects to find a list of oranges, and crashes, cannot arise, because this reference is not shareable.

5.5. Subsumption

The subsumption relation is extended with new rules for reasoning about references (Fig. 31). The Mezzo type-checker applies these rules transparently [Protzenko 2014a].

DECOMPOSEREF introduces a fresh name x for the content of the reference v. This allows reasoning separately about the reference and about its content. Decomposition was used in §2.2 when we examined lines 14–17 of Fig. 7. It is reversible: the rule can be used in both directions.

UNIFYREF states that if we have two names for the content of a reference, then these names must denote the same value. (In that case, we must also have $m_1 = m_2 = D$.) For instance, the conjunction of the permissions $x \,@\, \mathtt{ref}_D =x_1$ and $x \,@\, \mathtt{ref}_D =x_2$ implies the equation $x_1 = x_2$, which is sugar for $x_1 \,@\, =x_2$.

DUPREF states that the type $\mathtt{ref}_D\ T$ of immutable references is duplicable, provided the type T of the content is duplicable.

COREF states that $\mathtt{ref}_m \cdot$ is covariant, regardless of m. Again, this is safe, even if m is X, because $\mathtt{ref}_X \cdot$ is a type of uniquely-owned references.

[9]By using the subsumption rule that introduces an existential quantifier, followed with the subsumption rule DECOMPOSEREF (§5.5), used from right to left, one can combine the permissions $x \,@\, \mathtt{ref}_m =v$ and $v \,@\, T$ so as to obtain $x \,@\, \mathtt{ref}_m\ T$.

5.6. Resources

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

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

$$\begin{array}{rcl} D\,v \star D\,v &=& D\,v \\ N \star X\,v &=& X\,v \\ X\,v \star N &=& X\,v \\ N \star N &=& N \\ \_ \star \_ &=& \lightning \end{array} \qquad \begin{array}{rcl} \widehat{\lightning} &=& \lightning \\ \widehat{N} &=& N \\ \widehat{(D\,v)} &=& D\,v \\ \widehat{(X\,v)} &=& N \end{array} \qquad \begin{array}{rcl} N &\lhd& D\,v \\ iv &\lhd& iv \end{array} \qquad \begin{array}{c} N\ ok \\ D\,v\ ok \\ X\,v\ ok \end{array}$$

The definition of composition $\star$ requires agreement about which locations are shared ($D\,v$) versus uniquely-owned (N or $X\,v$). Furthermore, it requires agreement about the content of shareable memory locations ($D\,v_1 \star D\,v_2$ is $\lightning$ if the values v_1 and v_2 differ) and requires separation at uniquely-owned memory locations ($X\,v_1 \star X\,v_2$ is $\lightning$).

The definition of the core $\widehat{\cdot}$ contains the clause $\widehat{(X\,v)} = N$, which means that an exclusive instrumented value contains no shareable information.

The definition of rely $\lhd$ contains the clause $N \lhd D\,v$, which means that "someone else" may decide to turn a memory location that "they" own exclusively into a read-only, shared location. This clause is needed when proving subject reduction for the operation of freezing a reference.

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

The composition operation $\star$ is defined pointwise:

$$\begin{array}{rcl} \emptyset \star \emptyset &=& \emptyset \\ (ih_1 :: iv_1) \star (ih_2 :: iv_2) &=& (ih_1 \star ih_2) :: (iv_1 \star iv_2) \\ \_ \star \_ &=& \lightning \end{array}$$

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

The function "core" is also defined pointwise:

$$\begin{array}{rcl} \widehat{\lightning} &=& \lightning \\ \widehat{\emptyset} &=& \emptyset \\ \widehat{(ih :: iv)} &=& \widehat{ih} :: \widehat{iv} \end{array}$$

The relation "rely" looks slightly more complex:

$$\lightning \lhd \lightning \qquad \frac{\begin{array}{c} \mathit{limit}\ ih_1 \le \mathit{limit}\ ih_2 \\ \forall \ell \quad \ell < \mathit{limit}\ ih_1 \Rightarrow ih_1(\ell) \lhd ih_2(\ell) \\ \forall \ell \quad \mathit{limit}\ ih_1 \le \ell < \mathit{limit}\ ih_2 \Rightarrow \left\{ \begin{array}{l} ih_2(\ell)\ ok \\ \widehat{ih_2(\ell)} = ih_2(\ell) \end{array} \right. \end{array}}{ih_1 \lhd ih_2}$$

The rule on the right-hand side has three premises. The first premise states that the allocation limit can only increase with time. Thus, deallocation is forbidden[10], and allocation is permitted, subject to the next two premises. The second premise requires that, at every existing memory location, one follows the "rely" relation over instrumented values. The last premise requires that every newly allocated location ℓ be mapped by ih_2 to some consistent and duplicable instrumented value. The requirement that $ih_2(\ell)$ be consistent is necessary in order to prove that "rely preserves consistency", i.e., ih_1 *ok* and $ih_1 \lhd ih_2$ imply ih_2 *ok*. This is one of the MSA axioms (§4.1). The requirement that $ih_2(\ell)$ be duplicable is not essential at this point; it is exploited in order to establish that certain predicates are stable in §7.6.

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

In the same way that we have taken a machine state to be a tuple of a heap and possibly other components (§5.3), we take a *resource* R to be a tuple of a heap resource and possibly other components. Again, we write $\downarrow R$ for the "heap resource" component of the resource R.

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

5.7. Assigning types to values

REF (Fig. 32) is the introduction rule for the type constructor `ref`. For now, it is the only rule that assigns a type to a memory location. (New rules that concern memory locations are introduced when we describe adoption and abandon in §7.) This rule splits the current resource into two fragments. The fragment R_2 must map ℓ to $m\ v$: this means that R_2 grants m-access to the location ℓ and, at the same time, that v is the value stored at this location. The fragment R_1 justifies that v has type T. The rule concludes that ℓ has type $\texttt{ref}_m\ T$. Thus, intuitively, the type $\texttt{ref}_m\ T$ represents the separate ownership of the memory cell at address ℓ *and* of the value v that is currently stored there, to the extent dictated by the type T.

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

5.8. Type soundness and data-race freedom

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

We now express and prove the fact that "well-typed programs are data-race free". We need an auxiliary judgement t `accesses` ℓ `for` am. This judgement (whose definition is omitted) means that the term t (which represents either a single thread or a thread soup) is ready to access the memory location ℓ for reading or writing, as indicated by the access mode am, which is *R* or *W*. Using this judgement, we define a *racy* thread

[10]Naturally, in practice, one can use a garbage collector to reclaim unreachable objects. The fact that this is a valid implementation technique could be proved separately, if desired. This proof would not exploit the type discipline in any way.

$$
\begin{array}{rcll}
v, t, T, P & ::= & \ldots & \text{(Everything)} \\
& | & k & \text{(Values: } v\text{)} \\
& | & \texttt{newlock} \mid \texttt{acquire}\ v \mid \texttt{release}\ v & \text{(Programs: } t\text{)} \\
& | & \texttt{lock}\ P \mid \texttt{locked} & \text{(Types: } T\text{)}
\end{array}
$$

Fig. 33. Locks: syntax

initial configuration		*new configuration*	*side condition*
s / `newlock`	$\longrightarrow$	s' / *limit* kh	$kh = \downarrow s \wedge kh' = kh :: \boldsymbol{L} \wedge s' = kh' \uparrow s$
s / `acquire` k	$\longrightarrow$	s' / $()$	$kh = \downarrow s \wedge kh(k) = \boldsymbol{U} \wedge kh' = kh[k \mapsto \boldsymbol{L}] \wedge s' = kh' \uparrow s$
s / `release` k	$\longrightarrow$	s' / $()$	$kh = \downarrow s \wedge kh(k) = \boldsymbol{L} \wedge kh' = kh[k \mapsto \boldsymbol{U}] \wedge s' = kh' \uparrow s$

Fig. 34. Locks: operational semantics

soup t as one where two distinct threads are about to access a single memory location ℓ and at least one of these accesses is a write.

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

LEMMA 5.1 (TYPED ACCESS). *Every memory access is justified by a suitable access right.*

$$
\frac{R \vdash t \qquad t \text{ accesses } \ell \text{ for } am \qquad R\ \mathit{ok}}{\exists m,\ \exists v,\ (\downarrow R(\ell) = m\ v) \wedge (am = \boldsymbol{W} \Rightarrow m = \boldsymbol{X})}
$$

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

THEOREM 5.2 (DATA-RACE FREEDOM). *A well-typed configuration is not racy.*

$$
\frac{\vdash h \mathbin{/} t}{\neg(t \textit{ is racy})}
$$

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

6. LOCKS

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

We follow the same outline as in the previous section (§5). An "end user" should be interested only in the extensions of the syntax of terms and types (§6.1) and in the typing rules for the operations on locks (§6.3). The rest concerns the extension of the operational semantics (§6.2) and of the type soundness proof (§6.4–6.7).

6.1. Syntax

We extend the syntax as per Fig. 33. Values now include lock addresses k, which are implemented as natural numbers. For simplicity, we allocate references and locks in two separate heaps, with independent address spaces.

NewLock
$R; K; Q \vdash \texttt{newlock} : \exists x : \texttt{value}.(=x \mid (x \,@\, \texttt{lock}\ P) \ast (x \,@\, \texttt{locked}))$

Acquire
$R; K; v \,@\, \texttt{lock}\ P \vdash \texttt{acquire}\ v : \top \mid P \ast (v \,@\, \texttt{locked})$

Release
$R; K; P \ast (v \,@\, \texttt{locked}) \ast (v \,@\, \texttt{lock}\ P) \vdash \texttt{release}\ v : \top$

Fig. 35. Locks: typing rules for terms

DupLock
$\texttt{empty} \leq \texttt{duplicable}\ (\texttt{lock}\ P)$

Fig. 36. Locks: subsumption rules

Lock
$$\frac{\downarrow R(k) = (P, \_)}{R; K; Q \ \diamond\ k : \texttt{lock}\ P}$$

Locked
$$\frac{\downarrow R(k) = (\_, X)}{R; K; Q \ \diamond\ k : \texttt{locked}}$$

Fig. 37. Locks: typing rules for values

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

Types now include the type `lock` P of a lock whose invariant is the permission P. The type `lock` P is duplicable (Fig. 36), regardless of P. This allows several threads to share (and compete for) a lock. Types now also include the type `locked`. This type is not duplicable. It serves as a proof that a lock is held and (hence) as a permission to release the lock.

6.2. Operational semantics

A *lock status* is U (unlocked) or L (locked). A *lock heap* kh maps a lock address to a lock status. We re-use the operations on heaps introduced earlier (§5.2). Mirroring the steps of §5.3, we take a machine state to be a tuple of several components, one of which is a lock heap: $s ::= (\ldots, kh, \ldots)$. Again, we write $\downarrow s$ ("get") for the "lock heap" component of the machine state s and $kh \uparrow s$ ("set") for the machine state obtained by updating the "lock heap" component of s with kh[11].

The reduction rules for locks appear in Fig. 34. `newlock` creates a new lock in the locked state (L). This may seem nonstandard, and is opposite to surface Mezzo, where `lock::new` constructs an unlocked lock (§1). In principle, this convention is preferable, because it offers more expressive power: see §6.3. `acquire` k needs the lock k to be unlocked (U), and sets it to the locked state (L). `release` k does exactly the opposite. As in concurrent separation logic [O'Hearn 2007], we choose non-reentrant locks, because reentrant locks are more difficult to describe in terms of permissions [Haack et al. 2008].

6.3. Assigning types to terms

The typing rules for the operations on locks appear in Fig. 35. They are analogous to (and inspired by) the axioms of concurrent separation logic with dynamically-allocated

[11]The operators $\downarrow\cdot$ and $\cdot\uparrow\cdot$ are overloaded here, as well as the other operations on heaps. This overloading is also present in the Coq development, and is managed by type classes.

locks [O'Hearn 2007; Gotsman et al. 2007; Hobor et al. 2008]; see, in particular, Buisse *et al.*'s formulation [2011].

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

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

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

According to Acquire and Release, the expressions acquire x and release x have the precondition $x \,@\, \texttt{lock}\ P$, which guarantees that x is a valid lock with invariant P. acquire x produces the permissions P and $x \,@\, \texttt{locked}$, whereas, symmetrically, release x requires and consumes these permissions. It should be intuitively clear that the type system prevents double release: indeed, because $x \,@\, \texttt{locked}$ is affine, release x cannot be invoked twice. Formally, a configuration where a thread attempts to release an unlocked lock cannot make progress (see Fig. 34) and is considered unacceptable (see §6.7). Hence, the type soundness theorem implies that a well-typed program cannot reach such a configuration.

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

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

6.4. Resources

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

We achieve this by equipping the type of all permissions with the structure of a "discrete MSA", deduced from the notion of discrete separation algebra by Dockins *et al.* [2009]. An element of the *discrete MSA of permissions* is either $\lightning$ or a permission P. The MSA operations are defined as follows:

$$\begin{array}{rcl} P \star P & = & P \\ \_ \star \_ & = & \lightning \end{array} \qquad \begin{array}{rcl} \widehat{P} & = & P \\ \widehat{\lightning} & = & \lightning \end{array} \qquad \begin{array}{c} P \lhd P \\ \lightning \lhd \lightning \end{array} \qquad P\ ok$$

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

In addition to this, and independently of this, for every lock in existence, we wish to keep track of "who" (if anyone) has acquired this lock and (hence) has an exclusive right to release this lock. To do this, we use an *MSA of exclusive access rights*. The elements of this MSA are $\lightning$, N, and X, where N intuitively represents no access right and X represents an exclusive access right[12]. The MSA operations are defined as follows:

$$\begin{array}{rcl} N \star X & = & X \\ X \star N & = & X \\ N \star N & = & N \\ \_ \star \_ & = & \lightning \end{array} \qquad \begin{array}{rcl} \widehat{\lightning} & = & \lightning \\ \widehat{N} & = & N \\ \widehat{(X)} & = & N \end{array} \qquad \begin{array}{c} \lightning \lhd \lightning \\ N \lhd N \\ X \lhd X \end{array} \qquad \begin{array}{c} N \; ok \\ X \; ok \end{array}$$

In short, X represents an exclusive right to release the lock: $X \star X$ is $\lightning$.

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

A *lock resource* is an instrumented lock status heap. Lock resources form an MSA, whose construction is the same as the construction of heap resources in §5.6. As we did there, we take a *resource* R to be a tuple of a lock resource and possibly other components. Again, we write $\downarrow R$ for the "lock resource" component of the resource R.

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

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

6.5. Hidden state

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

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

This leads us to refine our understanding of the correspondence $s \sim R$. This relation should not be taken to mean that R represents the *entire* instrumented state; instead, it means that R is the *fragment* of the instrumented state that is *visible* to the program,

[12]This can be viewed as a simplified version of the MSA of instrumented values of §5.6. This time, X does not carry any argument, and there is no element D.

while the rest is *hidden*. To account for this idea, we define the relation $s \sim R$ as follows:

$$\frac{s \textit{ and } R \star R' \textit{ agree} \qquad R'; \varnothing \Vdash \texttt{hidden invariants of} \downarrow(R \star R')}{s \sim R}$$

A machine state is a monolithic entity: it cannot be split. As a result, the premise s *and* $R \star R'$ *agree* implies that the resource $R \star R'$ represents the entire instrumented state. We split this resource between a visible part R, which appears in the conclusion, and a hidden part R'.

The second premise requires the hidden part R' to justify the conjunction of the invariants of all currently unlocked locks. The *hidden invariant* of an instrumented lock status is defined by the equations `hidden invariant of` $(P, N) = P$ and `hidden invariant of` $(P, \_) =$ `empty`. That is, if the lock is currently unlocked, then its invariant P is currently hidden; otherwise, nothing is currently hidden. The *hidden invariants* of a lock resource R are defined as follows: `hidden invariants of` R is the (syntactic) conjunction, over all lock addresses k, of `hidden invariant of` $R(k)$.

Thanks to this somewhat elaborate construction, we *can* now establish the subject reduction lemma, and our earlier statement of it (Lemma A.10) does not need to be altered in any way. The proof cases for `acquire` and `release` involve transferring the resource fragment that justifies the invariant P between the hidden resource R' and the visible resource R.

6.6. Assigning types to values

The typing rules Lock and Locked (Fig. 37) assign types to lock addresses, thus giving meaning to the types `lock` P and `locked`. Their premises look up the lock resource $\downarrow R$. According to Lock, a lock address k whose invariant (as recorded in $\downarrow R$) is P receives the type `lock` P. A well-kindedness premise (which, by convention, we have hidden) requires P to be closed. According to Locked, a lock address k whose access right (as recorded in $\downarrow R$) is X receives the type `locked`.

6.7. Soundness

A configuration is now deemed acceptable if every thread either (i) has reached a value; (ii) is waiting on a lock that is currently held; or (iii) is able to take a step. The statements of type soundness (including those of the main intermediate lemmas, described in Appendix A) are unchanged. Well-typed programs cannot go wrong (i.e., they can reach only acceptable configurations) (§4.7) and are data-race free[13] (§5.8).

7. ADOPTION AND ABANDON

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

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

[13]The definition of a race does not change with locks. In particular, two competing accesses to a lock are not considered as conflicting, since this is precisely what locks are used for.

v, t, T, P ::=	...	(Everything)
	$\mid$ `give` v_1 `to` v_2 $\mid$ `take` v_1 `from` v_2 $\mid$ `fail` $\mid$ `take!` v_1 `from` v_2	(Terms: t)
	$\mid$ `adoptable` $\mid$ `unadopted` $\mid$ `adopts` T	(Types: T)

Fig. 38. Adoption and abandon: syntax

initial configuration	*new configuration*	*side condition*
s / `give` ℓ `to` ℓ'	$\longrightarrow s'$ / ()	$h = \downarrow s \wedge h(\ell) = \langle p \mid v \rangle \wedge h' = h[\ell \mapsto \langle \ell' \mid v \rangle] \wedge s' = h' \uparrow s$
s / `take` ℓ `from` ℓ'	$\longrightarrow s$ / `take!` ℓ `from` ℓ'	$h = \downarrow s \wedge h(\ell) = \langle \ell' \mid v \rangle$
s / `take` ℓ `from` ℓ'	$\longrightarrow s$ / `fail`	$h = \downarrow s \wedge h(\ell) = \langle p \mid v \rangle \wedge p \neq \ell'$
s / `take!` ℓ `from` ℓ'	$\longrightarrow s'$ / ()	$h = \downarrow s \wedge h(\ell) = \langle p \mid v \rangle \wedge h' = h[\ell \mapsto \langle \mathit{null} \mid v \rangle] \wedge s' = h' \uparrow s$
s / E[`fail`]	$\longrightarrow s$ / `fail`	

Fig. 39. Adoption and abandon: operational semantics

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

We follow roughly the same outline as in the previous sections (§5, §6). An "end user" should be interested mainly in the extensions of the syntax of terms and types (§7.1) and in the typing rules and subsumption rules for adoption and abandon (§7.4, §7.5). We also informally explain the gap between Mezzo and Core Mezzo (§7.2), discuss the properties of adoption and abandon in a concurrent setting (§7.9), and reflect on the design of adoption and abandon (§7.10). The rest concerns the extension of the operational semantics (§7.3) and of the type soundness proof (§7.6–7.8).

7.1. Syntax

We extend the syntax as per Fig. 38. Two new instructions appear. The instruction `give` x_1 `to` x_2 transfers the ownership of the object x_1 from the executing thread to the object x_2. (We use the word "object" as a synonym for "memory block".) In other words, its effect is that x_2 adopts x_1. The instruction `take` x_1 `from` x_2 has the reverse meaning. It checks (at runtime) that x_1 is presently adopted by x_2. If this check is successful, then the ownership of x_1 is taken away from x_2 and transferred back to the executing thread. In other words, x_2 abandons x_1.

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

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

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

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

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

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

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

7.2. From Mezzo to Core Mezzo

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

This gap can be bridged via a desugaring process, which we explain very briefly and informally. The definition of the type `graph a` (Fig. 12) is annotated with the clause **`adopts`** `node a`. This means that every graph can be used as an adopter of nodes; more precisely, the permission `g @ graph a` in Mezzo is desugared as `g @ graph a * g @ adopts (node a)` in Core Mezzo. Furthermore, in Mezzo, every node can be adopted: to account for this, the permission `n @ node a` in Mezzo is desugared as `n @ node a * n @ unadopted` in Core Mezzo. Finally, the type **`dynamic`** of Mezzo is known as `adoptable` in Core Mezzo.

Core Mezzo is more verbose, but also more orthogonal and more expressive. For instance, a function that takes an argument of type `adopts (node a)` is applicable not just to a graph, but to any object that adopts nodes. In the future, we would like to make this expressive power available in Mezzo. A set of automatically-generated type abbreviations could be used to retain conciseness.

7.3. Operational semantics

Contrary to what happened when we extended the kernel with references and locks, we need not extend the machine state with a new component. However, we must modify the structure of the heap. Instead of mapping memory locations to values (§5.2), a *heap* h now maps memory locations to blocks, where a *block* $\langle p \mid v \rangle$ is a pair of an adopter pointer p and a value v. A *pointer* p is either *null* or a memory location ℓ.

The operational semantics of references (Fig. 29) must be slightly adjusted so as to account for the presence of adopter fields. When a new block is allocated, its adopter field is *null*. When a block is read or written, its adopter field is ignored and unaffected. We omit the details.

The operational semantics of adoption and abandon appears in Fig. 39.

The first reduction rule indicates that give ℓ to ℓ' is just a write instruction: the adopter field at address ℓ is overwritten and receives the nonnull value ℓ'. As we will see, the type discipline guarantees that the value p previously held in this field was *null*. This is not visible in the operational semantics. We intentionally omit the side condition $p = \textit{null}$: this makes it clear that no runtime check is required.

NEWREFWITHADOPTION
$R; K; v \mathbin{@} T \vdash \texttt{newref}\ v : \exists x : \texttt{value}.(=x \mid (x \mathbin{@} \texttt{ref}_m\ T) * (x \mathbin{@} \texttt{adopts}\ \bot) * (x \mathbin{@} \texttt{unadopted}))$

GIVE
$R; K; (v_2 \mathbin{@} \texttt{adopts}\ U) * (v_1 \mathbin{@} U) * (v_1 \mathbin{@} \texttt{unadopted}) \vdash \texttt{give}\ v_1\ \texttt{to}\ v_2 : \top \mid (v_2 \mathbin{@} \texttt{adopts}\ U)$

TAKE
$R; K; (v_2 \mathbin{@} \texttt{adopts}\ U) * (v_1 \mathbin{@} \texttt{adoptable}) \vdash \texttt{take}\ v_1\ \texttt{from}\ v_2 : \top \mid (v_2 \mathbin{@} \texttt{adopts}\ U) * (v_1 \mathbin{@} U) * (v_1 \mathbin{@} \texttt{unadopted})$

FAIL
$R; K; P \vdash \texttt{fail} : T$

TAKE!
$$\frac{R; K \Vdash \ell' \mathbin{@} \texttt{adopts}\ U \qquad \downarrow R \vdash \ell \textit{ is adopted by } \ell'}{R; K; P \vdash \texttt{take!}\ \ell\ \texttt{from}\ \ell' : \top \mid (\ell' \mathbin{@} \texttt{adopts}\ U) * (\ell \mathbin{@} U) * (\ell \mathbin{@} \texttt{unadopted})}$$

Fig. 40. Adoption and abandon: typing rules for terms

DUPADOPTABLE
$\texttt{empty} \leq \texttt{duplicable adoptable}$

UNADOPTEDADOPTABLE
$v \mathbin{@} \texttt{unadopted} \leq (v \mathbin{@} \texttt{unadopted}) * (v \mathbin{@} \texttt{adoptable})$

COADOPTS
$$\frac{T \leq U}{v \mathbin{@} \texttt{adopts}\ T \leq v \mathbin{@} \texttt{adopts}\ U}$$

Fig. 41. Adoption and abandon: subsumption rules

The second and third reduction rules describe the runtime check performed by the instruction take ℓ from ℓ'. If the pointer p stored in the adopter field at address ℓ is equal to ℓ', then the check succeeds, and the instruction reduces to take! ℓ from ℓ'. Otherwise, the check fails, and the instruction reduces to fail.

The fourth rule indicates that take! ℓ from ℓ' is just a write instruction: the adopter field at address ℓ is overwritten with the value *null*. The type discipline guarantees that the value previously held in this field was ℓ'. In a concurrent setting, this is nonobvious: even though the current thread has just ascertained that the adopter field contains ℓ', another thread could in principle have stepped in and written a different value. We come back to this issue below (§7.9).

The last rule in Fig. 39 is a standard reduction rule for fail. The evaluation context is discarded, which means, intuitively, that once a thread encounters fail, it stops.

One could criticize the fact that, if take ℓ from ℓ' fails, this failure cannot be caught and handled. Mezzo provides an expression `y` **`adopts`** `x`, which (statically) requires the ownership of `y` and (at runtime) tests whether `x` is currently adopted by `y`, producing a Boolean result. This test can be followed by an ordinary **`if`** construct; in the first branch, **`take`** `x` **`from`** `y` is guaranteed to succeed, even in the face of interference by other threads; in the second branch, appropriate action can be taken. The introduction of this construct means that valuable information can be stored in the adopter field: for instance, by using two distinct adopters, a graph traversal can use the adopter field to record which nodes have been visited.

7.4. Assigning types to terms

The typing rules for adoption and abandon appear in Fig. 40. There is a new version of the typing rule for memory allocation, as well as one typing rule for each of the constructs that were introduced in Fig. 38. Remember that $\top$ is a unit type and that $\bot$ is an uninhabited type.

The typing rule NEWREFWITHADOPTION replaces the rule NEWREF that was given earlier (Fig. 30). The previous rule states that a memory allocation instruction of the form "let x = newref v in ..." consumes the permission $v @ T$ and gives rise to the permission $x @ \texttt{ref}_m\ T$. The new rule states that two additional permissions are produced, namely $x @ \texttt{adopts}\ \bot$ and $x @ \texttt{unadopted}$. This reflects the fact that the newly allocated memory location plays three distinct roles. The permission $x @ \texttt{ref}_m\ T$ still states that x denotes the address of a reference, which can be read and (if permitted by the mode m) written. The permission $x @ \texttt{adopts}\ \bot$ states that x can be used as an adopter (i.e., as the second argument of give or take), and that all of its adoptees currently have type $\bot$ (i.e., it currently has no adoptees). The permission $x @ \texttt{unadopted}$ states that x is currently not adopted and also implies that it can be used as an adoptee (i.e., as the first argument of give or take; this implication relies on the subsumption rule UNADOPTEDADOPTABLE, see §7.5). The permission $x @ \texttt{ref}_m\ T$ is duplicable or affine, depending on the mode m, whereas $x @ \texttt{adopts}\ \bot$ and $x @ \texttt{unadopted}$ are affine.

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

TAKE is the mirror image of GIVE. One needs permission to use v_2 as an adopter ($v_2 @ \texttt{adopts}\ U$), and this permission is preserved. Furthermore, if the instruction take v_1 from v_2 succeeds, then one learns that v_1 was among the adoptees of v_2, and has now been abandoned by v_2. Thus, v_1 must have type U ($v_1 @ U$) and is no longer adopted ($v_1 @ \texttt{unadopted}$). The ownership of v_1 is taken away from v_2 and transferred back to the executing thread: the footprint of the permission $v_2 @ \texttt{adopts}\ U$ shrinks. One apparent source of asymmetry between GIVE and TAKE is that the latter requires proof that v_1 has an adopter field ($v_1 @ \texttt{adoptable}$). This is required for soundness: the instruction take v_1 from v_2 would be unsafe if the value v_1 turned out to be something other than a memory location (say, a λ-abstraction). This asymmetry is only apparent, though. Because permission subsumption allows $v_1 @ \texttt{adoptable}$ to arise out of $v_1 @ \texttt{unadopted}$ (see further on), one can derive a variant of GIVE where $v_1 @ \texttt{adoptable}$ is part of the postcondition.

FAIL is standard. Since fail never terminates normally, its postcondition is false. In other words, it is deemed to have every type T.

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

7.5. Subsumption

The subsumption relation is extended with new rules for reasoning about adoption and abandon (Fig. 41).

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

The subsumption rule UNADOPTEDADOPTABLE states that, when one holds the affine permission $v @ \texttt{unadopted}$, one can obtain, in addition, the duplicable permission $v @ \texttt{adoptable}$. Indeed, $v @ \texttt{unadopted}$ means that there is an adopter field at address v which currently contains a *null* pointer, whereas $v @ \texttt{adoptable}$ asserts only that there is an adopter field at address v. According to NEWREFWITHADOPTION (Fig. 40), a new memory location has type `unadopted`; hence, via this subsumption rule, a new memory location also has type `adoptable`, and since this is a duplicable type, this fact remains true forever.

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

7.6. Resources

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

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

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

An *adoption status* is a pair of an adoptee status and an adopter status. For every memory location ℓ, the roles "ℓ as an adoptee" and "ℓ as an adopter" are logically independent, which is why we use a pair, whose components can be looked up and updated independently of one another. Adoption statuses form an MSA.

An *adoption resource* is an adoption status heap. Adoption resources form an MSA. We will shortly restrict our attention to a subset of "round" adoption resources, which also forms an MSA.

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

(1) $R \vdash \ell$ *is adoptable* holds iff ℓ is in the domain of R, i.e., ℓ is a valid memory location. Because every memory block has an adopter field, this condition is sufficient to ensure that there is an adopter field at address ℓ.
(2) $R \vdash \ell$ *is unadopted* holds iff R maps ℓ to a pair of the form (*X null*, $\_$). That is, R owns ℓ as an adoptee and the adopter pointer at ℓ is currently *null*.
(3) $R \vdash \ell'$ *is an adopter* holds iff R maps ℓ' to a pair of the form ($\_$,*X*). That is, R owns ℓ' as an adopter.
(4) $R \vdash \ell$ *is adopted by* ℓ' holds iff R maps ℓ to a pair of the form (*X* ℓ', $\_$). That is, R owns ℓ as an adoptee, and there is an edge from ℓ to ℓ', which one can think of as "owned by R" as well.
(5) $R \vdash \vec{\ell}$ *are the adoptees of* ℓ' holds iff the following two conditions are met:
— $R \vdash \ell'$ *is an adopter* holds; and

— the list $\vec{\ell}$ contains all of the addresses ℓ such that $R \vdash \ell$ *is adopted by* ℓ' holds, and it contains each such address just once. That is, $\vec{\ell}$ is a list of all adoptees of ℓ' according to R, and R owns every member of the list $\vec{\ell}$ as an adoptee, and R owns every edge from a member of $\vec{\ell}$ to ℓ'.

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

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

$$\frac{R_1 \vdash \vec{\ell}\ \textit{are the adoptees of}\ \ell' \qquad R_1 \star R_2\ \textit{ok}}{R_1 \star R_2 \vdash \vec{\ell}\ \textit{are the adoptees of}\ \ell'}$$

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

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

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

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

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

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

$$\text{ADOPTABLE}\ \frac{\downarrow R \vdash \ell \textit{ is adoptable}}{R; K; P \vdash \ell : \texttt{adoptable}} \qquad \text{UNADOPTED}\ \frac{\downarrow R \vdash \ell \textit{ is unadopted}}{R; K; P \vdash \ell : \texttt{unadopted}} \qquad \text{ADOPTS}\ \frac{\downarrow R_1 \vdash \vec{\ell} \textit{ are the adoptees of } \ell' \quad R_2; K \Vdash \vec{\ell} @ U}{R_1 \star R_2; K; P \vdash \ell' : \texttt{adopts}\ U}$$

Fig. 42. Adoption and abandon: typing rules for values

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

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

7.7. Assigning types to values

The typing rules ADOPTABLE, UNADOPTED, and ADOPTS (Fig. 42) assign types to memory locations, thus giving meaning to the types `adoptable`, `unadopted`, and `adopts` U. They come in addition to the typing rule REF (Fig. 32), which is unmodified.

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

7.8. Soundness

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

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

7.9. Adoption and abandon in a concurrent setting

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

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

Because the type unadopted (which, on the side of the adoptee, enables give) is affine, the case of two conflicting give instructions cannot occur in well-typed programs. On the other hand, the type adoptable (which, on the side of the adoptee, enables a take attempt) is duplicable. Hence, the other two cases can in fact arise: they are not ruled out by the type discipline.

Is this a problem? We argue in the following that it is not. We give two arguments: a formal one, under the assumption of sequential consistency, and an informal one, without this assumption.

Our type soundness theorem guarantees that, whichever interleaving is considered, “a well-typed program does not go wrong”. Since the operational semantics of take is in two steps, this includes interleavings where the execution of a take instruction is interrupted by another thread. Thus, under the assumption of a sequentially consistent memory model, we have a formal proof that the type discipline is sound, even though take is not implemented by an atomic compare-and-set instruction.

Our claim that “well-typed programs are data-race free” is formally valid as well, because our definition of a data race (§5.8) views conflicting accesses to an ordinary field as a race, but does *not* view conflicting accesses to an adopter field as a race. The reader who is unhappy with this way of stating things may prefer the longer statement: “well-typed programs are data-race free, except for possible races on adopter fields”.

We note that two concurrent give and/or take instructions must involve distinct adopters. This follows from the fact that both give and take require a unique permission of the form v_2 @ adopts U for the adopter.

This remark allows us to informally argue that all possible interleavings of give and/or take must lead to the same outcome. The argument is as follows. Let us consider two concurrent instructions i_1 and i_2 on a common adoptee ℓ. At least one of them, say i_1, must be a take instruction, say take ℓ from ℓ'. This instruction reads the adopter pointer stored at address ℓ and performs a dynamic test: is this pointer equal to ℓ', or is it some other (possibly *null*) address? This test is stable under the interference that may be inflicted by i_2. Although i_2 may write the adopter field at address ℓ, it must involve an adopter other than ℓ', which implies that it can change the adopter field from some value other than ℓ' to some value other than ℓ', but not from ℓ' to some other value or from some other value to ℓ'. Thus, i_2 cannot affect the outcome of the dynamic test performed by i_1. Furthermore, if this test succeeds, then i_2 cannot be a give instruction or a successful take instruction (that would contradict the fact that the adopter pointer is ℓ'). Thus, i_2 must be a failing take instruction. This implies that i_2 will not write to memory and (therefore) commutes with the write performed by i_1.

In summary, we have just informally argued that, under sequential consistency, any two concurrent give and/or take instructions commute. In other words, a data race on an adopter field is benign: it cannot be a source of nondeterminism.

It seems plausible that, under a relaxed memory model, appropriate memory-fence instructions could be added to the implementation of give and take so as preserve type soundness. In fact, we hypothesize that no memory fences are needed as part of give and take. Intuitively, a memory fence is required when some permission is transferred from one thread to another; but give and take cannot cause such a transfer. That said, adapting our proof of type soundness to relaxed memory is a standing challenge. One would need, first, to understand how to define a tractable operational semantics for a relaxed memory model (either for one specific model in existence, such as TSO, or for an imaginary “most relaxed” model), and that in itself is an open issue. One should also clarify which sequence of machine instructions is executed when a high-level read or write instruction takes place. Indeed, many garbage collectors impose read or write barriers: a high-level read or write instruction can translate to more than one machine instruction.

7.10. Design discussion

One might wonder why the type **`dynamic`**, or `adoptable`, is so uninformative: it gives no clue as to the type of the adoptee or the identity of the adopter. Would it be possible to parameterize it so as to carry either information? The short answer is negative. The type `adoptable` is duplicable, so the information that it conveys should be stable (i.e., forever valid). However, through a combination of strong updates and **`give`** and **`take`** instructions, the type of an adoptee may change with time (e.g., a graph node may move from the type `node ()` to the type `node int`), and the identity of its adopter may change as well (e.g. a node may be adopted by some graph `g1` at one point in time and later adopted by some other graph `g2`). Thus, in theory, it does not make sense for `adoptable` to carry more information.

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

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

— To avoid paying the cost of one adopter field in every object, one could let the programmer decide, for each data type, whether objects of this type should be adoptable (hence, need an `adopter` field) or not. The tag update instruction would be restricted so as to forbid going from an adoptable data type to a nonadoptable one, or vice-versa. For now, for the sake of simplicity, we have considered only the uniform model where every object has an `adopter` field.

— To permanently delegate `y1`'s adopter role to some other object `y2`, one could add an instruction `merge y1 into y2`. The effect of this instruction would be that all adoptees of `y1` immediately become adopted by `y2`, and **`take`** `x` **`from`** `y1` and **`give`** `x` **`to`** `y1` thereafter are synonymous with **`take`** `x` **`from`** `y2` and **`give`** `x` **`to`** `y2`. Its implementation would rely on a union-find data structure, and would cost one extra field per (adopter) object.

8. EXTENSIONS

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

(1) Allow memory blocks to have *multiple fields*, designated by an integer offset. The primitive type $\texttt{ref}_m\ T$ is replaced with a record type, say $m\ \{\vec{T}\}$.
(2) Equip memory blocks with a *tag*, represented as an integer. The record type $m\ \{\vec{T}\}$ is replaced with a structural type $i\ m\ \{\vec{T}\}$, where i is a tag. This type is analogous to the structural type `Cons { head: a; tail: list a }` found in Mezzo. Introduce the *tag update* instruction, which allows changing the tag of a memory block.

(3) Introduce a *union* type, of the form $T_0 \cup \ldots \cup T_{n-1}$, with the condition that the i-th summand, T_i, must exhibit the tag i. (That is, T_i should be a subtype of $i\ m\ \{\vec{T}\}$, for some m and $\vec{T}$.) Add the *injection* axiom $T_i \leq T_0 \cup \ldots \cup T_{n-1}$. Introduce a *switch* construct, whose typing rule refines $T_0 \cup \ldots \cup T_{n-1}$ to T_i in the i-th branch.
(4) (This item is independent of the previous three.) Add parameterized iso-recursive type definitions, of the form $\tau\ \vec{x} \sim T$, which are (un)folded via subsumption. For instance, the *nominal type* `list` x could be declared isomorphic to the union type $0\,\boldsymbol{D}\,\{\} \cup 1\,\boldsymbol{D}\,\{x; \texttt{list}\ x\}$.

In Mezzo, the type `ref` of mutable references is defined as part of the core library: **`data mutable`** `ref a = Ref { contents: a }`. Thus, `ref t` is a nominal type, and is interconvertible with the structural type `Ref { contents: t }`. The latter corresponds in spirit to the primitive type $\texttt{ref}_X\ T$ of §5. Similarly, in Mezzo, one can define a type of immutable references, as follows: **`data`** `iref a = IRef { contents: a }`. The type `IRef { contents: t }` then corresponds to the primitive type $\texttt{ref}_D\ T$. Freezing, which is written `tag` **`of`** `r <- IRef` in Mezzo, is represented by the instruction `ghost` in §5. The implementation of write-once references (§2.1) is a closely related example of use of these concepts.

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

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

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

9. 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].

```
val x = newref 0
val l = lock::new [(x @ ref int)] ()
```

Fig. 43. An explicit type application

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

Fig. 44. A more idiomatic type annotation

9.1. 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 `xs @ Cons { head: a; tail: list a }`, then, in order to justify the call `length xs`, the type-checker must first convert this permission to `xs @ 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 [Chrzaszcz 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 FRAMESUB and HIDEDUPLICABLEPRECONDITION in Fig. 26.

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.

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

Fig. 45. An ambiguity at a join point

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: perm] (| consumes p) -> ()` 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: perm] (| p) -> ()`, 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. 43, or via an annotation that indicates which permissions one expects to possess after the call, as in Fig. 44.

Intersection types. Mezzo is able to encode intersection types: in the presence of the permissions `f @ t1 -> u1` and `f @ t2 -> u2`, the function `f` has type `t1 -> u1` *and* `t2 -> u2` at the same time. Thus, a function call `f x` is well-typed if the current permission entails `x @ t1` *or* `x @ t2`. 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 P_1 and P_2, then the type-checker must compute a permission P such that $P_1 \leq P$ and $P_2 \leq P$ hold. Naturally, it should compute the most precise such P, if one exists.

This problem would be trivial if the disjunction of two permissions, $P_1 \vee P_2$, was part of the syntax of permissions. However, we deliberately choose *not* to allow it, for several reasons. First, that would make type-checking significantly more costly: potentially exponentially so. Second, that would change the model that the programmer must keep in mind. Instead of "one current permission at each program point", the programmer would have to think in terms of "one out of several possible current permissions at each program point, depending on which path was followed up to this point". That would draw us further away from the spirit of type-checking and closer to program analysis.

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

Fig. 46. An ambiguity at a join point, resolved by an explicit annotation

In the absence of a syntactic disjunction $P_1 \vee P_2$, the type-checker must compute an over-approximation P of it, as described above. This problem has been studied also in the setting of shape analysis [Chang and Rival 2008]. In the presence of nonduplicable resources, it does not in general admit a principal solution. For instance, in Fig. 45, two possible types for `x` are `(unknown, ref int)` and `(ref int, unknown)`. They are incomparable. Furthermore, `(ref int, ref int)` is not a valid type for `x`, as the two components of the pair `x` are not always distinct references. In such a situation, the current type-checker emits a warning and makes a heuristic choice. We note that, by performing a (backward) liveness analysis and by ignoring any variables that are dead at the join point, one increases the chances that the join problem admits a principal solution. This idea was suggested to us by Xavier Rival.

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 **match** construct appears in terminal position in a function, the join point after the **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. 46, 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.

9.2. 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:

```
xs @ Cons { head = h; tail = t } *
h  @ (=h1, =h2) *
h1 @ int *
h2 @ int *
t  @ list (int, int)
```

Assume further that P' is `xs @ 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 `length xs`.

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

Then, the type-checker compares the structural permission that describes `xs`, namely `xs @ Cons { head = h; tail = t }`, with the goal, and deduces that the goal can be reduced to the conjunction `h @ a * t @ 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 @ 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 @ list (=h)` does not hold: not all elements of the list are equal to its head `h`.
— Instantiating `a` with `(=h1, =h2)` similarly solves the first sub-goal and leads to a failure in the second sub-goal.
— Instantiating `a` with `(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 `int` and `a2` with `int`, among several other possibilities.
— In fact, there are infinitely many successful ways of instantiating `a`. For instance, one may instantiate with `(int, int |` **`empty`**`)`, with `(int, int |` **`empty`** `*` **`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. 43) and 28 explicit type annotations (as in Fig. 44). 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.

9.3. 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].

10. RELATED WORK

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

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

10.1. Annotating types with owners

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

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

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

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

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

10.2. Annotating types with permissions

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

In Boyland's work [2003], a permission is a fraction q in the interval $(0, 1]$. The unit fraction 1 means "we" have read-write access, whereas "others" have no access at all. Any fraction less than 1 means "everyone" has read-only access. Permissions can be split and recombined, which allows a heap fragment to transition from the state "read-write, exclusive" to the state "(temporarily) read-only, shared" and back. As of today, this is impossible in Mezzo. Although it should be possible to extend Mezzo with fractional permissions, we have not investigated this path yet.

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

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

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

10.3. Replacing types with permissions

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

Separation logic obeys a principle that we dub "owner-as-asserter". (In O'Hearn's words, "ownership is in the eye of the asserter" [2007]. This principle has also been referred to as "owner-as-privilege" [Clarke et al. 2004].) As usual in a program logic, objects are described by assertions. The novelty lies in the fact that a separation logic assertion is a permission: it not only represents knowledge about the current state of an object, but also implies that "we" may act in certain ways on this object, and that "others" may not. Thus, "to assert is to own". The assumption that "x is a linked list" means that "we" may read and write the cells that form this list, and "others" may not. Whereas the systems mentioned previously (§10.1, §10.2) combine traditional structural descriptions (i.e., types) with owner or permission annotations, separation logic assertions are at once structural descriptions and claims of ownership.

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

One motivation is that this makes the system conceptually less redundant. Indeed, a traditional type assumption, such as "x has type list", can be viewed as a duplicable permission in the sense of Mezzo, stating that "everyone may assume that x points to a linked list, and everyone must preserve this fact". From a pragmatic standpoint, eliminating the redundancy between (traditional) types and permissions leads to a more concise system. In Mezzo, types are not annotated with owners (§10.1) or with access permissions (§10.2). Polymorphic code is just type-polymorphic; it does not have to be both type- and annotation-polymorphic. That said, there are situations in Mezzo where something that resembles "ownership annotations" appears anyway: this is the case, for instance, when working with static regions, whose encoding on top of nesting is shown later on (§10.6).

The second motivation for this design decision is that it allows the "type" of an object to change. In a traditional type system, types are fixed: because every object is potentially shared, its type cannot be altered. In a system where permissions are layered on top of types (§10.2), a strong update can alter the information carried by a unique permission. For instance, in Bierhoff and Aldrich's system [2007], the "typestate" of an object can be changed by its owner. However, the object's underlying type still cannot be modified. In separation logic or in Mezzo, where information about the "type" of an object is carried by the permission, a strong update can alter this information. This enables gradual initialization (§2.2), memory re-use, and certain forms of typestate tracking [Guéneau et al. 2013].

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

In contrast with ownership type systems (§10.1), Mezzo does not purposely impose any topological restrictions on the heap. Nevertheless, it provides strong "end-to-end" safety guarantees: well-typed programs do not go wrong (which, in the presence of type-changing updates, is a nontrivial result) and are data-race free.

10.4. Containers, ownership, and ownership transfer

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

In principle, ownership type systems (§10.1) can easily describe both situations. It is just a matter of annotating the type of the elements with an appropriate owner.

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

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

```
val put: [a] (container a, key, consumes a) -> ()
val get: [a] (container a, key) -> a
```

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

In contrast, the early ownership type systems (§10.1) do not keep track of uniqueness, hence do not allow strong updates or ownership transfer. Later proposals [Clarke and Wrigstad 2003; Müller and Rudich 2007] include a notion of "external uniqueness" which supports ownership transfer.

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

There are several ways by which a mutable object may be assigned a duplicable type. (1) Adoption and abandon allows assigning the duplicable type **`dynamic`** (also known as `adoptable`, see §7) to an object. Access to the object, together with information about its "real" type, is obtained via a `take` instruction. (2) Nesting allows assigning the duplicable type `inhabitant r a` (Fig. 14) to an object `x`. The permission `x @ a`, which allows accessing the object, is obtained via a `focus` instruction. Similarly, our encoding of regions on top of nesting, shown later on (§10.6), allows assigning the duplicable type `rref r a` to a mutable cell that inhabits the region `r`. (3) Pairing an object `x` together with a lock that protects the permission `x @ a` results in a package of duplicable type `protected a`, which is defined as an abbreviation for `(x: unknown, lock (x @ a))`. Access is obtained by acquiring the lock. An example of this last possibility is a simple definition of communication channels in Mezzo, which can be done using a (mutable) queue protected by a lock [Protzenko 2014a].

10.5. Linearity, singleton types, and capabilities

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

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

Alias Types makes another major contribution, which is to recognize that it does not really matter that there exist at most one pointer to an object. It is fine for multiple pointers to exist, as long as there is a unique (static) capability to dereference any of these pointers. Thus, Smith *et al.* introduce the distinction between values, which are duplicable, and capabilities (or, in our terminology, permissions), which are linear or affine, and need not exist at runtime.

In order to keep track of which capabilities give access to which objects, and (at the same time) to keep track of the known equations between pointers, Alias Types uses singleton types. A pointer `p` receives a singleton type $\texttt{ptr}\,\ell$, where ℓ is a type-level name for a memory location. A pointer `q` that happens to also have type $\texttt{ptr}\,\ell$ is statically known to be equal to `p`. A capability $\{\ell \mapsto \tau\}$ guarantees that the memory location represented by ℓ currently contains data of type τ. This capability allows reading and writing, through `p` or `q`, and a strong update can change this capability to $\{\ell \mapsto \tau'\}$, for some new type τ'.

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

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

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

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

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

In the tradition of linear logic, many linear or affine type systems in the literature adopt the convention that types are by default not duplicable. An explicit modality, written "!", must be used to indicate that a value is duplicable. This is the case, for instance, in L^3 [Ahmed et al. 2007], and in Pottier's earlier work [2013], which was inspired in part by dual intuitionistic linear logic [Barber 1996]. From a theoretical point of view, this approach may seem pleasant. In practice, however, it seems too verbose to be tolerable. Several authors have suggested recording duplicability information at the level of kinds [Charguéraud and Pottier 2008; Mazurak et al. 2010; Tov and Pucella 2011]. In Mezzo, in contrast, this information is implicit in the syntax of types. The type `int`, for instance, is inherently duplicable. The type `ref int` is inherently nonduplicable. The tuple type `(t, u)` is duplicable if and only if `t` and `u` are duplicable; and so on. The type-checker "knows" these rules and applies them transparently. (As

```
abstract region

abstract rref (r : value) a
fact duplicable (rref r a)

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

Fig. 47. A signature for regions, which can be implemented on top of nesting

noted in §2.2, the type-checker infers the rules that govern user-defined types, such as `list t`.) Formally, **duplicable** `t` is viewed as a permission, and the rules listed above are permission subsumption rules. This approach seems lightweight, both in theory and in practice. As illustrated by the types of `get` and `set` in Fig. 5, it is easy to express both quantification with respect to an arbitrary type and quantification with respect to a duplicable type. Mezzo's duplicable permissions are analogous to the necessary assertions found in CaReSL [Turon et al. 2013].

By convention, in Mezzo, every function type is duplicable. This means functions are easy to use: they can be freely shared, and can be called as many times as one wishes, provided one is able to supply an argument of appropriate type. The flip side is that this can make functions difficult to construct: in particular, a function is not allowed to capture an affine permission that exists at its definition site (see FUNCTION in §4.4). This is not a problem, though, as the affine type $T \multimap U$ of functions that can be called at most once can be defined as an abbreviation for $\exists p : \mathtt{perm}.(((T \mid p) \rightarrow U) \mid p)$. (This is Core Mezzo syntax.) This encoding, which was exploited already in the definition of magic wands (§2.4), can be intuitively understood as a package of a shotgun and one cartridge, represented by the abstract permission p. The shotgun can be fired at most once, because it consumes p, which is not known to be duplicable. In fact, Mezzo allows encoding not only the type $T \multimap U$ of one-shot functions, but also variations on this theme. For instance, an affine type of functions that can be called many times can be defined as $\exists p : \mathtt{perm}.(((T \mid p) \rightarrow (U \mid p)) \mid p)$. As another example, double-barrelled CPS style, where one is handed a pair of a success continuation and a failure continuation and one is allowed to invoke at most one of them, can be easily and faithfully encoded in Mezzo.

10.6. Regions, nesting, adoption and abandon

From a type-theoretic point of view, a region is a type-level name for a set of values, typically a set of memory locations.

A region may also exist at runtime, in which case it is typically an area where objects can be dynamically allocated one by one, and can be deallocated all at once. In Tofte and Talpin's original work [1994; 1997], regions exist at type-checking time and at runtime. This is the case also in Walker *et al.*'s paper [2000], in Vault [DeLine and Fähndrich 2001; Fähndrich and DeLine 2002], and in Cyclone [Swamy et al. 2006]. In some of these works, so as to avoid confusion, the type-level entity is referred to as a *region*, whereas the runtime entity is referred to as a *region handle*.

Nesting [Boyland 2010] is closely related to static regions. The name `r` of the "nester" can be viewed as a region name; the "nestees" can be viewed as region inhabitants. Whoever has (exclusive) permission for `r` also owns every inhabitant. This relationship was illustrated in Fig. 14, where the types `region` and `inhabitant r a` reflect this idea. If one wishes to push this idea slightly further, one can define in Mezzo a library module that offers the signature in Fig. 47. The abstract type `region` is defined internally by **`data mutable`** `region = Region`, as in Fig. 14. The function call **`let`** `r = newregion()` **`in`** `...` produces a new region `r`. The affine permission `r @ region` then serves as a unique right to access the cells allocated in the region `r`. The duplicable type `rref r a` describes a reference that inhabits the region `r`. This abstract type is defined internally as an abbreviation for `inhabitant r (ref a)`, where `inhabitant` is defined as in Fig. 14. The operations `newrref`, `get`, and `set` respectively allocate, read, and write a reference which (conceptually) inhabits the region `r`. Each of these operations requires (and returns) the permission `r @ region`. The operation `get` is restricted to the case where the type `a` is duplicable: this is another instance of the "borrowing problem", which we have discussed at length (§2.4).

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

```
val runST [a] (f: [r: value] (| r @ region) -> a) : a =
  let r = newregion() in
  f [r] ()
```

This implementation of `runST` is just an exercise, though: in practice, a user would typically use `newregion()` directly, instead of calling `runST`. We note in passing that in Mezzo it is easy to work with multiple regions simultaneously, whereas we believe that this would be awkward in Haskell.

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

As far as we understand, Rust [The Mozilla foundation 2014] uses (static) regions, much in the same way as Cyclone and Haskell, so as to guarantee that a piece of memory is not accessed outside of a certain lexical scope. Rust's "borrowed pointer" type `&'a T` is indexed with a "lifetime" `'a`, another name for a region. Within function bodies, lifetimes are usually inferred. Within function signatures, they must in principle be explicitly written down, although a small number of syntactic conventions allow them to be "elided" in many cases. Rust and Mezzo have several common aspects. They both borrow features from ML, such as first-class functions and algebraic data types and pattern matching. They both keep track of ownership at type-checking time so as to ensure memory safety and data race freedom. Their type systems seem rather different: the central concepts in Rust appear to be lifetimes and borrowing, whereas Mezzo attempts to explain everything in terms of permissions and permission transfer.

Some features of Rust, such as read-only borrows, cannot be simulated in Mezzo. For efficiency, Rust supports several modes of memory management (garbage-collected; stack-allocated; reference-counted; etc.) as well as various forms of unboxing, whereas Mezzo has none of these features.

Bugliesi *et al.* [2015] define an affine refinement type system for the concurrent λ-calculus RCF and use it to verify security properties of application code. This system is doubly interesting in our eyes because it looks and feels very much like Mezzo and furthermore (we believe) it includes a form of nesting, under the name "exponential serialization". Bugliesi *et al.* use affine formulae (or permissions, in our terminology) to establish an injective correspondence between **assert** and **assume** instructions. For instance, supposing the affine formula `order x y` means "an order has been placed by client `x` for item `y`", the instruction **assume** `(order x y)` produces this formula, while the instruction **assert** `(order x y)` consumes it. Affine permissions also control the use of nonces: in Mezzo syntax, the primitive function `mknonce` could have type `() -> (x: bytes | x @ nonce)`, where `nonce` is an affine type. A permission can be sent on the network as part of a message; however, because the adversary may capture and replay messages, every message is required to have duplicable type. This creates an apparent difficulty: an affine permission such as `order x y` cannot be sent across the network. How, then, can one send from principal A to principal B the information that an order has been placed? Bugliesi's *et al.*'s solution, rephrased in our terms, involves nesting. Suppose B has created a nonce `z` and (via an earlier message) has communicated `z` to A. Suppose B has retained the affine permission `z @ nonce`. Then, A proceeds as follows. Instead of sending the affine permission `order x y` to B, which is prohibited, A uses a `nest` operation (§2.6) to nest `order x y` in `z`. In return, A obtains a duplicable nesting witness, `nests z (order x y)`, which it can send to B. On the receiving end, B uses a `focus` operation. This operation requires the nesting witness, consumes `z @ nonce`, and produces `order x y`. Thus, the affine permission `order x y` has been transmitted, as desired.

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

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

For many practical purposes, adoption and abandon is a more flexible mechanism than nesting. One price to pay is the runtime cost of **give** and **take**, as well as the introduction of potential runtime failures. Another weakness of adoption and abandon is that, in its simplest form, it does not support transferring all adoptees of adopter `y1` to adopter `y2` in constant time (this was discussed at the end of §7), whereas in the case of purely static regions, one can imagine a ghost instruction that merges one region into another.

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

10.7. Concurrency and locks

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

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

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

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

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

The idea of using a type discipline to enforce the correct usage of locks can be traced back to Flanagan and Abadi [1999]. There, every lock receives a type-level name, and every reference cell receives a type that mentions this name. The code is type-checked under a "current permission", which is the set of the currently-held locks. This discipline can be simulated in Mezzo (if desired) by letting a lock protect a region (in the sense of §10.6). Thus, by acquiring the lock, one obtains a permission for the region, which in turn allows one to access the region inhabitants. If one prefers to use adoption and abandon rather than a static region, one can let a lock protect an adopter. Acquiring the lock yields a permission for the adopter, which (via a **take** instruction) gives access to the adoptees.

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

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

Fractional permissions [Boyland 2003; Bornat et al. 2005; Boyland 2010] are supported by several tools, including VeriFast [Jacobs and Piessens 2008] and Chalice, which strives to hide them from the user, when possible [Heule et al. 2013]. Amighi *et al.* [2015] are also developing such a tool. For greater simplicity, Mezzo does not have fractional permissions; it distinguishes only between immutable and mutable data. In Mezzo, a mutable data structure can become immutable and shareable, but not the other way around. Furthermore, in Mezzo, one cannot create a temporary read-only view of a mutable data structure. Extending Mezzo with fractional permissions should allow removing these restrictions; we have not studied this extension. Regions, as in Cyclone [Swamy et al. 2006] and Rust [The Mozilla foundation 2014], offer another way of creating temporary read-only views, which does not require accounting.

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

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

Locks form the foundation of the POSIX and Java concurrency APIs, and are widely used in practice. Nevertheless, it is well-known that it is difficult to write correct, modular, efficient code that uses locks. Their numerous pitfalls include locking too little, which (in the absence of a static discipline such as Mezzo's) leads to data races; locking too much, which leads to lack of parallelism and (in the worst case) deadlocks; and contention. Furthermore, locks are a source of nondeterminism, which makes debugging difficult.

Bocchino *et al.* [2009a] argue that parallel programming (where concurrency is not part of the problem specification, but is used only for increased performance) should be deterministic by default. They envision a type-and-effect discipline where the heap is partitioned into disjoint regions, so as to ensure the absence of interference between threads. Deterministic Parallel Java [Bocchino Jr. et al. 2009b; Bocchino Jr. and Adve 2011] can be viewed as a concrete proposal along these lines. Its type system groups objects into (static) regions, and requires methods to be annotated with effects, which indicate which regions are accessed. A region can be a class parameter or a class field. Regions can be "nested" so as to express tree structure (hence, region disjointness information). The constructs that express parallelism, namely the **`cobegin`** block and the **`foreach`** loop, use effect information to ensure that parallel computations do not interfere. There is support for subarrays, with dynamic bounds checking. In later work, Bocchino *et al.* [2011] mix deterministic and nondeterministic code in a disciplined manner, and guarantee data-race freedom. Bocchino [2013] provides a good summary of this line of work as well as a survey of the related work.

Mezzo's static discipline could conceivably serve as the basis of a similar programme. It too can express tree structure and regions (for instance, in Fig. 14, a region is used as a type parameter and as a record field). The only source of nondeterminism in Mezzo is locks. Instead of defining (multiple-sender, multiple-receiver) communication channels on top of locks, as done at present in Mezzo's library, one could remove locks and view (blocking, single-sender, single-receiver) channels as a primitive feature, thus preserving determinism. This would form the basis of a simple framework for deterministic `fork`/`join` parallelism.

10.8. Proof techniques and modularity

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

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

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

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

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

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

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

11. 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 (§7), 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 (§4) can be described as a concurrent call-by-value λ-calculus, equipped with an affine, polymorphic, value-dependent type-and-permission system. The extensions are:

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

This paper is accompanied with a Coq proof [Balabonski and Pottier 2014]. It is about 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 (§9). 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

Peter Achten and Marinus J. Plasmeijer. 1995. The Ins and Outs of Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundamenta Informaticæ* 77, 4 (2007), 397–449.

Afshin Amighi, Christian Haack, Marieke Huisman, and Clément Hurlin. 2015. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science* 11, 1 (2015), 1–66.

Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. 2014. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In *Foundations of Software Science and Computation Structures (FOSSACS) (Lecture Notes in Computer Science)*, Vol. 8412. Springer, 411–425.

Thibaut Balabonski and François Pottier. 2014. A Coq formalization of Mezzo, take 2. (July 2014). http://gallium.inria.fr/~fpottier/mezzo/mezzo-coq.tar.gz.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2014. Type Soundness and Race Freedom for Mezzo. In *Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS 2014) (Lecture Notes in Computer Science)*, Vol. 8475. Springer, 253–269.

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic*. Technical Report ECS-LFCS-96-347. Laboratory for Foundations of Computer Science, School of Informatics at the University of Edinburgh.

Batteries included. 2014. `BatList`. (2014).

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS) (Lecture Notes in Computer Science)*, Vol. 3328. Springer, 97–109.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science)*, Vol. 4111. Springer, 115–137.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 3780. Springer, 52–68.

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

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2009. Practical API Protocol Checking with Access Permissions. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 5653. Springer, 195–219.

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2011. Checking Concurrent Typestate with Access Permissions in Plural: A Retrospective. In *Engineering of Software*, Peri L. Tarr and Alexander L. Wolf (Eds.). Springer, 35–48.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *Principles of Programming Languages (POPL)*. 119–132.

Robert L. Bocchino Jr. 2013. Alias Control for Deterministic Parallelism. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 156–195.

Robert L. Bocchino Jr. and Vikram S. Adve. 2011. Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 6813. Springer, 306–332.

Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009a. Parallel Programming Must Be Deterministic by Default. In *USENIX Conference on Hot Topics in Parallelism (HotPar)*. 1–6.

Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009b. A type and effect system for deterministic parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 97–116.

Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Principles of Programming Languages (POPL)*. 535–548.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 211–230.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 2694. Springer, 55–72.

John Tang Boyland. 2010. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems* 32, 6 (2010), 22:1–22:33.

Thomas Braibant and Damien Pous. 2011. Tactics for Reasoning Modulo AC in Coq. In *Certified Programs and Proofs (Lecture Notes in Computer Science)*, Vol. 7086. Springer, 167–182.

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 16–34.

Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2015. Affine Refinement Types for Secure Distributed Programming. (2015). To appear.

Alexandre Buisse, Lars Birkedal, and Kristian Støvring. 2011. A Step-Indexed Kripke Model of Separation Logic for Storable Locks. *Electronic Notes in Theoretical Computer Science* 276 (2011), 121–143.

Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. 2015. Open-sourcing Facebook Infer: Identify bugs before you ship. https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/. (2015).

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational inductive shape analysis. In *Principles of Programming Languages (POPL)*. 247–260.

Arthur Charguéraud. 2010. *Characteristic Formulae for Mechanized Program Verification*. Ph.D. Dissertation. Université Paris 7.

Arthur Charguéraud and François Pottier. 2008. Functional Translation of a Calculus of Capabilities. In *International Conference on Functional Programming (ICFP)*. 213–224.

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

Jacek Chrzaszcz. 1998. Polymorphic Subtyping Without Distributivity. In *International Symposium on Mathematical Foundations of Computer Science (Lecture Notes in Computer Science)*, Vol. 1450. Springer, 346–355.

Dave Clarke, Sophia Drossopoulou, and James Noble. 2004. Aliasing, Confinement, and Ownership in Object-Oriented Programming. In *Object-Oriented Technology. ECOOP 2003 Workshop Reader*. Lecture Notes in Computer Science, Vol. 3013. Springer, 197–207.

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58.

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 2743. Springer, 176–200.

David G. Clarke, James Noble, and John Potter. 2001. Simple Ownership Types for Object Containment. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 2072. Springer, 53–76.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 48–64.

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 23–42.

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 6901. Springer, 235–249.

Luis Damas. 1985. *Type Assignment in Programming Languages*. Ph.D. Dissertation. University of Edinburgh.

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à La Carte. In *Principles of Programming Languages (POPL)*. 207–218.

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Programming Language Design and Implementation (PLDI)*. 59–69.

David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. 1998. *Wrestling with rep exposure*. Research Report 156. SRC.

Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *Journal of Object Technology* 4, 8 (2005), 5–32.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Principles of Programming Languages (POPL)*. 287–300.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 6183. Springer, 504–528.

Dino Distefano and Matthew J. Parkinson. 2008. jStar: towards practical verification for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 213–226.

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 5904. Springer, 161–177.

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2014. The Spirit of Ghost Code. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 1–16.

Cormac Flanagan and Martín Abadi. 1999. Types for Safe Locking. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 1576. Springer, 91–108.

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

Manuel Fähndrich and Robert DeLine. 2002. Adoption and focus: practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*. 13–24.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 21–40.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. *Local Reasoning for Storable Locks and Threads*. Technical Report MSR-TR-2007-39. Microsoft Research.

Armaël Guéneau, François Pottier, and Jonathan Protzenko. 2013. The ins and outs of iteration in Mezzo. Higher-Order Programming and Effects (HOPE). (2013). http://goo.gl/NrgKc4.

Christian Haack, Marieke Huisman, and Clément Hurlin. 2008. Reasoning about Java's Reentrant Locks. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 5356. Springer, 171–187.

Christian Haack and Clément Hurlin. 2009. Resource Usage Protocols for Iterators. *Journal of Object Technology* 8, 4 (2009), 55–83.

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Vol. 7737. Springer, 315–334.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 4960. Springer, 353–367.

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuipe. 2015. Modular Termination Verification. In *European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics)*. 99–1023.

Bart Jacobs and Frank Piessens. 2008. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven.

Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Design Patterns in Separation Logic. In *Types in Language Design and Implementation (TLDI)*. 105–116.

James Richard Larus. 1989. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. Technical Report UCB/CSD-89-502.

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 378–393.

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 6012. Springer, 407–426.

Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. 2011. Extended Alias Type System using Separating Implication. In *Types in Language Design and Implementation (TLDI)*. 29–42.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in system F°. In *Types in Language Design and Implementation (TLDI)*. 77–88.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Principles of Programming Languages (POPL)*. 75–84.

Peter D. Mosses. 2004. Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228.

Peter Müller and Arsenii Rudich. 2007. Ownership transfer in universe types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 461–478.

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

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *Principles of Programming Languages (POPL)*. 261–274.

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation logic + superposition calculus = heap theorem prover. In *Programming Language Design and Implementation (PLDI)*. 556–566.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (2007), 271–307.

Simon Peyton Jones and Philip Wadler. 1993. Imperative functional programming. In *Principles of Programming Languages (POPL)*. 71–84.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 773–789.

François Pottier. 2008. Hiding local state in direct style: a higher-order anti-frame rule. In *Logic in Computer Science (LICS)*. 331–340.

François Pottier. 2013. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming* 23, 1 (2013), 38–144.

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. In *International Conference on Functional Programming (ICFP)*. 173–184.

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

Jonathan Protzenko. 2014a. *Mezzo: a typed language for safe effectful concurrent programs*. Ph.D. Dissertation. Université Paris Diderot.

Jonathan Protzenko. 2014b. A Mezzo sample project. (2014).

Jonathan Protzenko. 2014c. Mezzo-web: try Mezzo in your browser. (2014).

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested Hoare triples and frame rules for higher-order store. In *Computer Science Logic (Lecture Notes in Computer Science)*, Vol. 5771. Springer, 440–454.

Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. 2010. A Semantic Foundation for Hidden State. In *Foundations of Software Science and Computation Structures (FOSSACS) (Lecture Notes in Computer Science)*, Vol. 6014. Springer, 2–17.

Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. 1994. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Dagstuhl Seminar on Graph Transformations in Computer Science (Lecture Notes in Computer Science)*, Vol. 776. Springer, 358–379.

Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias Types. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 1782. Springer, 366–381.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 149–168.

Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory Management in Cyclone. *Science of Computer Programming* 62, 2 (2006), 122–144.

The Mozilla foundation. 2014. The Rust programming language. (2014).

Mads Tofte. 1988. *Operational Semantics and Polymorphic Type Inference*. Ph.D. Dissertation. University of Edinburgh.

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Principles of Programming Languages (POPL)*. 188–201.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997), 109–176.

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Principles of Programming Languages (POPL)*. 447–458.

Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 211–230.

Thomas Tuerk. 2010. Local reasoning about while-loops. (2010). Unpublished.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *International Conference on Functional Programming (ICFP)*. 377–390.

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351.

Jan Vitek and Boris Bokowski. 2001. Confined types in Java. *Software: Practice and Experience* 31, 6 (2001), 507–532.

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

David Walker, Karl Crary, and Greg Morrisett. 2000. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems* 22, 4 (2000), 701–771.

David Walker and Greg Morrisett. 2000. Alias Types for Recursive Data Structures. In *Types in Compilation (TIC) (Lecture Notes in Computer Science)*, Vol. 2071. Springer, 177–206.

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–356.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94.

A. PROOF OF TYPE SOUNDNESS

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

A.1. Hygiene properties of well-kindedness

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

A.2. Hygiene properties of well-typedness

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

LEMMA A.1 (SUBSTITUTION). *Let κ be* `value`*,* `type`*, or* `perm`*. Typing is preserved by the substitution of a syntactic element u of kind κ for a variable of kind κ.*

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

Note that, when one replaces a variable of kind `value` with a value v, this value is not required to be well-typed. This is a natural consequence of the fact that our kind environments do not contain any type assumptions. This should be contrasted with the substitution lemma of (say) simply-typed λ-calculus. A more conventional lemma can be recovered by using the above lemma in conjunction with the rule CUT (Fig. 24):

LEMMA A.2 (SUBSTITUTION/CUT). *Typing is preserved by the substitution of a value v of type U for a variable x that was assumed to have type U.*

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

A.3. Resources and well-typedness

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

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

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

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

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

LEMMA A.4 (DUPLICATION). *Duplicable permissions can be justified by duplicable resources.*

$$\frac{R; K \Vdash P \qquad R \textit{ ok} \qquad P \textit{ is duplicable}}{\widehat{R}; K \Vdash P}$$

As an immediate corollary of the above lemma, if R *ok* and P *is duplicable* hold, then $R; K \Vdash P$ implies $R; K \Vdash P * P$. This shows that the subsumption rule DUPLICATE (Fig. 26) is sound.

The actions of a thread cannot cause an inactive thread to become ill-typed. In other words, well-typedness is stable in the face of permitted interference, as defined by the "rely" relation $\lhd$ (§4.1).

LEMMA A.5 (STABILITY). *Typing is preserved under an evolution of the resource along the relation* $\lhd$.

$$\frac{R_1; K; P \vdash t : T \qquad R_1 \textit{ ok} \qquad R_1 \lhd R_2}{R_2; K; P \vdash t : T}$$

A.4. Classification and decomposition

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

LEMMA A.6 (CLASSIFICATION). *Among the values, only* λ*-abstractions admit a function type.*

$$\frac{R; K \Vdash v \,@\, T \to U}{\exists x,\ \exists t,\ v = \lambda x.t}$$

These statements must allow for a nonempty environment K. Indeed, as FORALLINTRO (Fig. 24) is not restricted to values, we must be able to reason about "reduction under Λ", that is, reduction in a nonempty environment.

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

LEMMA A.7 (DECOMPOSITION). *If $\lambda x.t$ has type $T \to U$, then t has type U under the precondition $x \,@\, T$.*

$$\frac{R; K \Vdash \lambda x.t \,@\, T \to U \qquad R \text{ ok}}{\widehat{R}; K, x : \texttt{value}; x \,@\, T \vdash t : U}$$

The above decomposition lemma is slightly stronger than one might expect in view of the typing rule FUNCTION (Fig. 24). Indeed, the lemma does not mention the fact that the function body may need a duplicable permission P. We are able to establish this strong statement because the permission P, if there is one, can be hidden by application of Lemma A.4 and of the typing rule CUT.

A.5. Soundness of subsumption and canonicalization

Permission subsumption, which we have inductively defined (Fig. 26), is sound with respect to the "semantic" notion of subsumption that arises out of the interpretation of permissions. All of the previous lemmas (except Lemma A.5, Stability, which is used only in the proof of Subject reduction) are used in this proof.

LEMMA A.8 (SOUNDNESS OF SUBSUMPTION). *Permission subsumption is sound:*

$$\frac{K \vdash P \leq Q \qquad R; K \Vdash P \qquad R \text{ ok}}{R; K \Vdash Q}$$

and so is subtyping:

$$\frac{K \vdash T \leq U \qquad R; K \Vdash v \,@\, T \qquad R \text{ ok}}{R; K \Vdash v \,@\, U}$$

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

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

$$\frac{R; K; \texttt{empty} \vdash v : T \qquad R \text{ ok}}{R; K \Vdash v \,@\, T}$$

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

A.6. Subject reduction and progress

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

LEMMA A.10 (SUBJECT REDUCTION, PRELIMINARY FORM). *Let the configuration $s_1 \ / \ t_1$ have kind* `term` *under an empty environment. (Thus, the term t_1 is closed and*

$$\textsc{Value}\quad R_{init}; K; v \,@\, T \vdash v : T$$

$$\textsc{LetFrame}\quad \frac{R_{init}; K; P \vdash t : T \qquad R_{init}; K, x : \mathtt{value}; Q * (x \,@\, T) \vdash u : U}{R_{init}; K; P * Q \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : U}$$

$$\textsc{NormalApp}\quad \frac{R_{init}; K; P \vdash t : U \to T \qquad R_{init}; K; Q \vdash u : U}{R_{init}; K; P * Q \vdash \mathtt{let}\ x = t\ \mathtt{in}\ x\ u : T}$$

Fig. 48. Derived typing rules: variable, sequencing, application

represents a single thread.) Assume s_1 / t_1 reduces in one step to s_2 / t_2. Assume the machine state s_1 corresponds to the resource $R_1 \star R_1'$, where R_1 allows arguing that t_1 has type T. Then, the machine state s_2 corresponds to some resource of the form $R_2 \star R_2'$, where R_2 allows arguing that t_2 has type T; and the interference that has been imposed to the environment, from R_1' to R_2', is permitted.

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1; \varnothing; \mathtt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \begin{cases} s_2 \sim R_2 \star R_2' \\ R_2; \varnothing; \mathtt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{cases}}$$

This result allows deriving the following, much more compact corollary, which is phrased in terms of the typing judgement for configurations (Fig. 27):

LEMMA A.11 (SUBJECT REDUCTION). *Reduction preserves well-typedness.*

$$\frac{c_1 \longrightarrow c_2 \qquad \vdash c_1}{\vdash c_2}$$

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

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

In other words, a configuration is acceptable if no thread has gone wrong.

LEMMA A.12 (PROGRESS). *Every well-typed configuration is acceptable.*

$$\frac{\vdash c}{c \textit{ is acceptable}}$$

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

THEOREM A.13 (TYPE SOUNDNESS). *Let t be a well-typed source program: that is, assume $R_{init}; \varnothing; \mathtt{empty} \vdash t : T$. Then, by executing the configuration s_{init} / t, one can reach only acceptable configurations.*

B. DERIVED RULES

None of the typing rules resemble the "axiom" rule of simply-typed λ-calculus, which states that x has type T under the assumption that x has type T. Indeed, SINGLETON

(Fig. 24) only allows proving that x has type $=x$. Fortunately, an axiom rule, VALUE, presented in Fig. 48, can be derived using SINGLETON, FRAME, and subsumption.

We have defined "let $x = u$ in t" as sugar for "$(\lambda x.t)\ u$". Using the rules FUNCTION, APPLICATION, CUT, FRAME, and subsumption, it is possible to derive a typing rule for this construct. The rule LETFRAME, shown in Fig. 48, uses part of the precondition (namely P) to prove that the term t has type T, while the rest (namely Q), together with the new hypothesis $x \mathbin{@} T$, is used to type-check the term u.

Finally, using LETFRAME, it is straightforward to obtain a new type-checking rule for function application, NORMALAPP (Fig. 48). This rule has two premises and splits the current permission between them. This is in contrast to APPLICATION (Fig. 24), which has only one premise, and requires the operator to be a value v. Here, the operator can be a term t, but an explicit sequencing construct must again be used.

C. ENCODING THE SIMPLY-TYPED λ-CALCULUS

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

$$\begin{aligned} [\![x]\!] &= x \\ [\![\lambda x.t]\!] &= \lambda x.[\![t]\!] \\ [\![t\ u]\!] &= \texttt{let}\ x = [\![t]\!]\ \texttt{in}\ x\ [\![u]\!] \end{aligned}$$

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

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

$$\begin{aligned} [\![x_1 : T_1, \ldots, x_n : T_n]\!] &= x_1 : \texttt{value}, \ldots, x_n : \texttt{value} \\ (\!|x_1 : T_1, \ldots, x_n : T_n|\!) &= x_1 \mathbin{@} T_1 \ast \ldots \ast x_n \mathbin{@} T_n \end{aligned}$$

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

The encoding is type-preserving:

LEMMA C.1 (ENCODING). *If $E \vdash t : T$ holds in the simply-typed λ-calculus, then $R_{init}; [\![E]\!]; (\!|E|\!) \vdash [\![t]\!] : [\![T]\!]$ holds.*