Type Soundness and Race Freedom for Mezzo

Thibaut Balabonski, François Pottier, and Jonathan Protzenko

INRIA

Abstract. The programming language Mezzo is equipped with a rich type system that controls aliasing and access to mutable memory. We incorporate shared-memory concurrency into Mezzo and present a modular formalization of its core type system, in the form of a concurrent λ-calculus, which we extend with references and locks. We prove that well-typed programs do not go wrong and are data-race free. Our definitions and proofs are machine-checked.

1 Introduction

Strongly-typed programming languages rule out some programming mistakes by ensuring at compile-time that every operation is applied to arguments of suitable 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 [13] can prove that private state is properly encapsulated; concurrent separation logic [10] 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 [12] 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. In this paper, we incorporate shared-memory concurrency into Mezzo and present its meta-theory.

A race. In order to illustrate Mezzo, let us consider the tiny program in Fig. 1. This code exhibits a data race, hence is incorrect, and is rejected by the type system. Let us explain how it is type-checked. At line 1, we allocate a reference

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

Fig. 1. 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.

(i.e., a memory cell), and store its address in the global variable `r`. In the eyes of the type-checker, this gives rise to a *permission*, written `r @ ref int`. This permission has a double reading: it describes the layout of memory (i.e., "the variable `r` denotes the address of a cell that stores an integer") and grants *exclusive* read-write access to this cell. That is, the type constructor `ref` denotes a uniquely-owned reference, and the permission `r @ ref int` is a unique token that one must possess in order to dereference `r`. This token exists at type-checking time only.

A permission `r @ ref int` looks like a traditional assumption `r : ref int`. However, a type assumption would be valid everywhere in the scope of `r`, whereas a permission is 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 away this token, then, even though `r` is still in scope, one can no longer read or write it.

Although `r @ ref int` is an affine permission (i.e., it cannot be copied), some permissions are duplicable. For instance, `x @ int` is a duplicable permission. If one can get ahold of such a permission, then one can keep it forever (i.e., as long as `x` is in scope) *and* pass copies of it to other threads, if desired. Such a permission behaves like a traditional type assumption `x : int`.

The function `f` in Fig. 1 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 an affine permission.) This is declared by the type annotation[1] at lines 2 and 3. Thus, at line 5, in conjunction with `r @ ref int`, we get a new permission, `f @ (| r @ ref int) -> (| r @ ref int)`. This means that `f` is a function with zero (runtime) argument and result, which (at type-checking time) requires and returns the permission `r @ ref int`. The type `T | P` denotes a package of a value of type `T` and the permission `P`. We write `(| P)` for `(() | P)`, where `()` is the unit type.

At line 6, the type-checker analyzes the sequencing construct in a manner analogous to separation logic: the second `spawn` instruction is checked using the permissions that are left over by the first `spawn`. An instruction `spawn f` requires two permissions: a permission to invoke the function `f`, and `r @ ref int`, which `f` itself requires. It does *not* return these permissions: they are transferred to the spawned thread. Thus, in line 6, between the two `spawn`s, we no longer have a permission for `r`. (We still have `f @ (|...) -> (|...)`, because it is duplicable.) Therefore, the second `spawn` is ill-typed. The racy program of Fig. 1 is rejected.

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 abstractly as a polymorphic, higher-order function, `hide` (Fig. 2).

[1] In the surface syntax of Mezzo, in the absence of a `consumes` keyword, a permission that is taken by a function is considered also returned, so one need not repeat `r @ ref int` in the header or in the type of `f`. In this paper, we do not use this convention. We work in a simpler, lower-level syntax where functions consume their arguments.

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

Fig. 2. The polymorphic, higher-order function `hide` takes a function `f` of type `(a | s) -> (b | s)`, which means that `f` needs access to some state represented by the permission `s`. The function `hide` requires `s`, and consumes it. It returns a function of type `a -> b`, which does not require `s`, hence can be invoked by multiple threads concurrently.

In Fig. 2, `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 guard the use of the possibly affine permission `s`. This is materialized by a duplicable permission `l @ lock s`, which is produced by the `newlock` instruction, and added to the two permissions `s` and `f @ (a | s) -> (b | s)` already present at the beginning of line 4. The fact that `l @ lock s` is duplicable is a key point: this enables multiple threads to compete for the lock even if the guarded permission is affine. The lock is created in the "locked" state, and released at line 5. This consumes `s`: when one releases a lock, one must give up and give back the permission that it controls. The permissions for `f` and `l` remain, and, because they are duplicable, they are also available within the anonymous function defined at line 6. (A closure *can* capture a duplicable permission.)

The anonymous function at line 6 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 `a -> b`, which does not mention `s`.

It is easy to fix the code in Fig. 1 by inserting the redefinition `val f = hide f` before line 5. This call consumes `r @ ref int` and produces `f @ () -> ()`, so the two `spawn` instructions are now type-checked without difficulty.

Channels. Acquiring or releasing a lock produces or consumes a permission: a transfer of ownership takes place between the lock and the active thread. This can be used to encode other patterns of ownership transfer. For example, a (multiple-writer, multiple-reader) communication channel, which allows exchanging messages and permissions between threads, is easily implemented as a FIFO queue, protected by a lock. Let us briefly describe the interface and implementation of this user-defined abstraction.

Channels are described by the interface in Fig. 3. Line 1 advertises the existence of an abstract type `channel a` of channels along which values of type `a` may be transferred. Line 2 advertises the fact that this type is duplicable. (We explain below why the definition of `channel` satisfies this claim.) This means that the permission to use a channel (for sending or receiving) can be shared between several threads. The type of `send` means that sending a value `x` along a channel `c` of type `channel a` consumes the permission `x @ a`. Symmetrically, the

```
1 abstract channel a
2 fact duplicable (channel a)
3 val new:     [a] ()              -> channel a
4 val send:    [a] (channel a, a) -> ()
5 val receive: [a] (channel a)    -> a
```

Fig. 3. An interface for communication channels

```
1  alias channel a =
2    (q: unknown, lock (q @ fifo a))
3  val new [a] () : channel a =
4    let q = queue::create () in
5    let l : lock (q @ fifo a) = newlock() in
6    release l;
7    (q, l)
8  val send [a] (c: channel a, x: a) : () =
9    let (q, l) = c in
10   acquire l;
11   queue::insert (x, q);
12   release l
13 val receive [a] (c : channel a) : a =
14   let (q, l) = c in
15   acquire l;
16   let rec loop (| q @ fifo a * l @ locked) : a =
17     match queue::retrieve q with
18     | None ->                  loop()
19     | Some { contents = x } -> release l; x
20     end
21   in loop()
```

Fig. 4. A simple implementation of channels using a queue and a lock

type of `receive` means that receiving a value `x` along such a channel produces the permission `x @ a`. It is important to note that the type `a` of messages is not necessarily duplicable. If it is not, then a transfer of ownership, from the sender thread to the receiver thread, is taking place.

Fig. 4 implements channels using a FIFO queue and a lock. The lock guards the exclusive permission to access the queue. In lines 1–2, the type `channel a` is defined as an abbreviation for a pair[2] of a value `q` of *a priori* unknown type (i.e., no permission is available for it) and a lock of type `lock (q @ fifo a)`. Acquiring the lock produces the permission `q @ fifo a`, so that, within a critical section, `q` is recognized by the type-checker as a queue, which can be accessed and updated. The type-checker accepts the claim that the type `channel a` is duplicable because it is defined as a pair of two duplicable types, namely `unknown` and `lock (...)`.

[2] The dependent pair notation used in this definition is desugared into existential types and singleton types, which are part of Mezzo's core type discipline (§2).

Contributions. Mezzo appeared in a previous paper by Pottier and Protzenko [12]. That paper does not cover concurrency. It presents Mezzo's type discipline in a monolithic manner, and does not contain any details about the proof of type soundness. In the present paper, Mezzo includes shared-memory concurrency, and its presentation is modularly organized in several layers. We identify a kernel layer: a concurrent, call-by-value λ-calculus extended with a construct for dynamic thread creation (§2). In its typed version, it is a polymorphic, value-dependent system, which enjoys type erasure: values exist at runtime, whereas types and permissions do not. The system provides a framework for handling duplicable as well as affine permissions, and is equipped with a rich set of subsumption rules that are analogous to separation logic entailment. 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 two layers, which are independent of one another, are heap-allocated references (§3) and locks (§4). Our definitions and proofs are machine-checked [2].

2 Kernel

2.1 Machine states and resources

The kernel calculus does not include any explicit effectful operations. Yet, in order to later add such operations without altering the statements of the main lemmas that lead to the type soundness result, we build into the kernel calculus the idea of a *machine state* s. At this stage, the nature of machine states is unspecified. Later on, we make it partially concrete, by specifying that a machine state is a tuple of a heap (§3), a lock heap (§4), and possibly more: the type of machine states is informally considered open-ended. The execution of a program begins in a distinguished machine state *initial*.

A program under execution is composed of multiple threads, each of which has partial knowledge of the current machine state and partial rights to alter this state. In the proof of type soundness, we account for this by working with a notion of *resource*, of which one can think as the "view" of a thread [7]. At this stage, again, the nature of resources is unspecified. One should think of a resource as a partial, instrumented machine state: a resource may contain additional information that does not exist at runtime, such as an access right for a memory location (§3), or the invariant associated with a lock (§4).

We require resources to form a *monotonic separation algebra* [11, §10]. That is, we assume the following:

- A composition operator $\star$ allows two resources (i.e., the views of two threads) to be combined. It is total, commutative, and associative.
- A 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. For instance, the allocation of new memory blocks, or of new locks, is typically permitted by this relation.
  - This relation is reflexive.
  - It preserves consistency, i.e., R_1 *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:

$$\frac{R_1 \star R_2 \lhd R' \qquad R_1 \star R_2 \ ok}{\exists R'_1 R'_2,\ R'_1 \star R'_2 = R' \wedge R_1 \lhd R'_1 \wedge R_2 \lhd R'_2}$$

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

2.2 Syntax

Values, terms, types, and permissions form a single syntactic category. There is a single name space of variables. Within this universe, defined in Fig. 5, we impose a kind discipline, so as to distinguish the following sub-categories[3].

The values v have kind value. They are the variables of kind value (the λ binder introduces such a variable) and the λ-abstractions.

The terms t have kind term. They encompass values. 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 $(\lambda x.t_2)\ t_1$. We reduce the number of evaluation contexts by requiring the left-hand side of an application to be a value. This does not reduce expressiveness: $t_1\ t_2$ can be encoded as let $x = t_1$ in $x\ t_2$.

The soups, also written t, have kind soup. They are parallel compositions of threads. A thread takes 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 $T \to U$; and the conjunction $T \mid P$ of a type and a permission.

[3] For the sake of conciseness, we omit the definition of the well-kindedness judgement, and omit the well-kindedness premises in the definition of the typing judgement. Instead, we use conventional metavariables (v, t, etc.) to indicate the intended kind of each syntactic element.

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

Fig. 5. Kernel: syntax of programs, types, and permissions

$$
\begin{array}{lcll}
\textit{initial configuration} & & \textit{new configuration} & \textit{side condition} \\
s \mathbin{/} (\lambda x.t)\ v & \longrightarrow & s \mathbin{/} [v/x]t & \\
s \mathbin{/} E[t] & \longrightarrow & s' \mathbin{/} E[t'] & s \mathbin{/} t \longrightarrow s' \mathbin{/} t' \\
s \mathbin{/} \textbf{thread}\ (t) & \longrightarrow & s' \mathbin{/} \textbf{thread}\ (t') & s \mathbin{/} t \longrightarrow s' \mathbin{/} t' \\
s \mathbin{/} t_1 \parallel t_2 & \longrightarrow & s' \mathbin{/} t_1' \parallel t_2 & s \mathbin{/} t_1 \longrightarrow s' \mathbin{/} t_1' \\
s \mathbin{/} t_1 \parallel t_2 & \longrightarrow & s' \mathbin{/} t_1 \parallel t_2' & s \mathbin{/} t_2 \longrightarrow s' \mathbin{/} t_2' \\
s \mathbin{/} \textbf{thread}\ (D[\textbf{spawn}\ v_1\ v_2]) & \longrightarrow & s \mathbin{/} \textbf{thread}\ (D[()]) \parallel \textbf{thread}\ (v_1\ v_2) &
\end{array}
$$

Fig. 6. Kernel: operational semantics

The permissions P include the atomic form $v \mathbin{@} T$, 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. A permission of the form duplicable θ is typically used as part of a constrained quantified type. For instance, $\forall x : \mathsf{type}.(x \mid (\mathsf{duplicable}\ x)) \to \ldots$ describes a polymorphic function which, for every duplicable type x, is able to take an argument of type x.

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

2.3 Operational semantics

The calculus is equipped with a small-step operational semantics (Fig. 6). The reduction relation acts on configurations c, which are pairs of a machine state s and a closed term or soup t. In the kernel rules, the machine state is carried around, but never consulted or modified.

2.4 Typing judgement and interpretation of permissions

The main two judgements, which depend on each other, are the *typing judgement* $R; K; P \vdash t : T$ and the *permission interpretation judgement* $R; K \Vdash P$. 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[4]. The kind environment K contains

[4] 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 judgement.

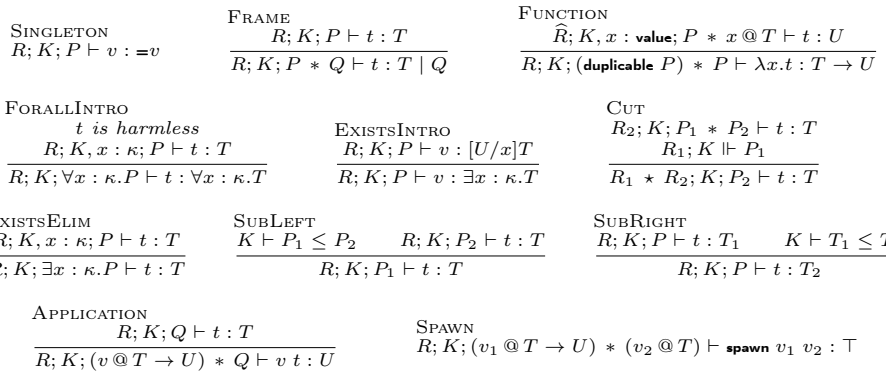

Fig. 7. Kernel: typing rules

information that does not evolve with time (i.e., the kind of every variable) whereas the precondition P contains information that evolves with time (i.e., the available permissions).

The typing judgement $R; K; P \vdash t : T$ 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 resource R plays a role only when reasoning about programs under execution: it is the "view" that each thread has of the machine state. When type-checking source programs, R is *void*.

The permission interpretation judgement $R; K \Vdash P$ means that R justifies, or satisfies, 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 is analogous to the interpretation of assertions in separation logic. It gives meaning, in terms of resources, to the syntax of permissions.

The typing judgement is defined in Fig. 7. The first five rules are introduction rules: they define the meaning of the type constructors. Singleton states that v is one (and the only) inhabitant of the singleton type $=v$. Frame can be applied to a value v or to a term t. In the latter case, it is a frame rule in the sense of separation logic. Because every function type is considered duplicable, a function body must be type-checked under duplicable assumptions. For this reason, in Function, P is required to be duplicable and R is replaced in the premise with its core $\widehat{R}$. ForallIntro can be applied to a value or to a term: there is no value restriction. Once hidden state is introduced (§4), polymorphism must be restricted to a syntactic category of *harmless* terms. For now, every term is harmless. ExistsIntro is standard.

Cut moves information between the parameters P and R of a judgement. In short, it says, if t is well-typed under the assumption P_1, then it is well-typed under R_1, provided the resource R_1 satisfies the permission P_1.

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

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

Fig. 8. Kernel: the interpretation of permissions

$$\textsc{MixStarIntroElim}\ (v \mathbin{@} T) * P \equiv v \mathbin{@} T \mid P \qquad \textsc{Frame}\ v \mathbin{@} T_1 \to T_2 \leq v \mathbin{@} (T_1 \mid P) \to (T_2 \mid P)$$

$$\textsc{Duplicate}\ (\mathsf{duplicable}\ P) * P \leq P * P \qquad \textsc{DupSingleton}\ \mathsf{empty} \leq \mathsf{duplicable}\ {=}v \qquad \textsc{DupArrow}\ \mathsf{empty} \leq \mathsf{duplicable}\ (T \to U)$$

Fig. 9. Kernel: permission subsumption (a few rules only; $K \vdash$ omitted)

Next, we find three non-syntax-directed rules, namely ExistsElim, SubLeft, SubRight. An important part of the type soundness proof consists in proving that every well-typed, closed value can be type-checked without using these rules.

Application is standard. Spawn states that spawn v_1 v_2 is type-checked just like a function application v_1 v_2, except a unit value is returned in the original thread. We write $\top$ for the type $\exists x : \mathsf{value}.{=}x$.

We now review the interpretation of permissions (Fig. 8). These rules play a role in the proof of type soundness, where they establish a connection between the syntax of permissions and their intended meaning in terms of resources. Empty, Star, Forall, Exists correspond to the interpretation of assertions in separation logic. Atomic states, roughly, that the resource R satisfies the permission $v \mathbin{@} T$ if the value v has type T under R. 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 θ, as follows: a variable x is not duplicable; a singleton type $=v$ is duplicable; a function type $T \to U$ is duplicable; a conjunction $T \mid P$ is duplicable if T and P are duplicable; and so on. We omit the full definition.

2.5 Subsumption

The permission subsumption judgement takes the form $K \vdash P \leq Q$. It is inductively defined by many rules, of which, by lack of space, we show very few (Fig. 9). MixStarIntroElim is a compact way of summing up the relationship between the two forms of conjunction. Frame is analogous to the typing rule by the same name (Fig. 7), and means that a function that performs fewer side effects can be passed where a function that performs more side effects is allowed. Duplicate states that if P is provably duplicable, then P can be turned into $P * P$. DupSingleton, DupArrow, and a family of similar rules (not shown) allow constructing permissions of the form duplicable θ.

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

Fig. 10. Kernel: typing rules for soups and configurations

The subtyping judgement used in SubRight is defined in terms of permission subsumption: we write $K \vdash T \leq U$ when $K, x : \mathsf{value} \vdash x \mathbin{@} T \leq x \mathbin{@} U$ holds.

2.6 Typing judgements for soups and configurations

The typing judgement for soups $R \vdash t$ (Fig. 10, 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. It means that, under the precondition R, the thread soup t is safe to execute.

The typing judgement for configurations $\vdash s \mathbin{/} t$ (Fig. 10, 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 $s \mathbin{/} t$ is safe to execute.

2.7 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, §3, and locks, §4) does not require any alteration to the statements or proofs of the main lemmas. Introducing new primitive operations (such as the operations that manipulate references and locks) does not require altering the statements either; naturally, it does create new proof cases.

For the sake of brevity, we state only the main two lemmas.

Theorem 1 (Subject reduction). *If* $c_1 \longrightarrow c_2$*, then* $\vdash c_1$ *implies* $\vdash c_2$*.*

Theorem 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 (§4) to allow for the possibility for a thread to be blocked (i.e., waiting for a lock).

3 References

We extend the kernel calculus with heap-allocated references. We show how the type system is extended and prove that it ensures data-race freedom.

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

Fig. 11. References: syntax

$$
\begin{array}{lcll}
\textit{initial config.} & & \textit{new configuration} & \textit{side condition} \\
h \mathbin{/} \mathsf{newref}\ v & \longrightarrow & h +\!\!+ v \mathbin{/} \textit{limit}\ h & \\
h \mathbin{/} !\ell & \longrightarrow & h \mathbin{/} v & h(\ell) = v \\
h \mathbin{/} \ell := v' & \longrightarrow & h[\ell \mapsto v'] \mathbin{/} () & h(\ell) = v
\end{array}
$$

Fig. 12. References: operational semantics

Syntax. We extend the syntax as per Fig. 11. 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. Types now include the type $\mathsf{ref}_m\ T$ of references whose current content is a value of type T. The mode m indicates whether the reference is shareable (*duplicable*, D) or uniquely-owned (*exclusive*, X). Only the latter mode allows writing: this is key to enforcing data-race freedom. The type $\mathsf{ref}\ T$ (§1) is short for $\mathsf{ref}_X\ T$.

Operational semantics. A *heap* h is a function of an initial segment of the natural numbers to values. We write *limit* h for the first unallocated address in the heap h. We write $h +\!\!+ v$ for the heap that extends h with a mapping of *limit* h 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.

We specify that a machine state s is a tuple, one of whose components is a heap h. In Fig. 12, we abuse notation and pretend that a machine state *is* a heap; thus, the reduction rules for references are written in a standard way. In Coq, we use overloaded "get" and "set" functions to mediate between the two levels.

Assigning types to terms. The typing rules for the operations on references appear in Fig. 13. A memory allocation expression $\mathsf{newref}\ v$ consumes the permission $v \mathbin{@} T$ and produces a new memory location of type $\mathsf{ref}_m\ T$ with mode m. Reading or writing a reference x requires a permission $x \mathbin{@} \mathsf{ref}_m\ T$, which guarantees that x is a valid memory location, and holds a value of type T. Because reading a reference creates a new copy of its content without consuming $x \mathbin{@} \mathsf{ref}_m\ T$, Read requires T to be duplicable. Write requires the exclusive mode X, in which the permission $x \mathbin{@} \mathsf{ref}_X\ T$ ensures that "nobody else" has any knowledge of (or access to) x. The rule allows strong update: the type of x changes to $\mathsf{ref}_X\ T'$, where T' is the type of v'. All three operations are harmless: there is no adverse interaction between polymorphism and uniquely-owned references [4,11].

Subsumption. Subsumption is extended with new rules for reasoning about references (Fig. 14). DecomposeRef introduces a fresh name x for the content of the reference v. This allows separate reasoning about the ownership of the reference cell and about the ownership of its content. This step is reversible. CoRef states that ref is covariant. For uniquely-owned references, this is standard [4,11].

$$\textsc{NewRef}\quad R; K; v \mathbin{@} T \vdash \mathbf{newref}\ v : \mathbf{ref}_m\ T$$

$$\textsc{Read}\quad R; K; (\mathbf{duplicable}\ T) * (v \mathbin{@} \mathbf{ref}_m\ T) \vdash\ !v : T \mid (v \mathbin{@} \mathbf{ref}_m\ T)$$

$$\textsc{Write}\quad R; K; (v \mathbin{@} \mathbf{ref}_X\ T) * (v' \mathbin{@} T') \vdash v := v' : \top \mid (v \mathbin{@} \mathbf{ref}_X\ T')$$

$$\textsc{Loc}\quad \frac{R_1; K \Vdash v \mathbin{@} T \qquad R_2(\ell) = m\, v}{R_1 \star R_2; K; P \vdash \ell : \mathbf{ref}_m\ T}$$

Fig. 13. References: typing rules for terms and values

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

$$\textsc{CoRef}\quad \frac{T \leq U}{v \mathbin{@} \mathbf{ref}_m\ T \leq v \mathbin{@} \mathbf{ref}_m\ U}$$

Fig. 14. References: subsumption rules

Resources. An *instrumented value* is $\lightning$, N, Dv, or Xv, where v is a value. N represents no information and no access right about a memory location, whereas for any $m \in \{D, X\}$, $m\ v$ represents full information (one knows that the value stored there is v). Dv (resp. Xv) moreover indicates a shared read-only access right (resp. an exclusive read/write access right). The type of instrumented values forms a monotonic separation algebra, where $Dv \star Dv$ is Dv, $N \star Xv$ and $Xv \star N$ are Xv; $N \star N$ is N; and every other combination yields $\lightning$.

A *heap resource* is either $\lightning$ or an instrumented value heap. Heap resources form a monotonic separation algebra, whose $\star$ operation requires agreement of the allocation limits (i.e., the next unallocated location is shared knowledge) and is defined pointwise. A heap resource is essentially a heap fragment in the sense of separation logic [13] and $\star$ is a union operation that requires disjointness at mutable locations and agreement at immutable locations. We specify that a *resource* R is a tuple of several components, one of which is a heap resource.

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

Assigning types to values. Loc (Fig. 13) is the introduction rule for the type constructor ref. It splits R: intuitively, the type $\mathsf{ref}_m\ T$ 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.

Data-race freedom. The auxiliary jugdement t $\mathsf{accesses}$ ℓ for am (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. A *racy* thread soup t is one where two distinct threads are ready 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

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

Fig. 15. Locks: syntax

$$
\begin{array}{lll}
\textit{initial config.} & \textit{new configuration} & \textit{side condition} \\
kh \mathbin{/} \textsf{newlock} \longrightarrow kh +\!\!+ L \mathbin{/} \textit{limit}\ kh & & \\
kh \mathbin{/} \textsf{acquire}\ k \longrightarrow kh[k \mapsto L] \mathbin{/} () & & kh(k) = U \\
kh \mathbin{/} \textsf{release}\ k \longrightarrow kh[k \mapsto U] \mathbin{/} () & & kh(k) = L
\end{array}
$$

Fig. 16. Locks: operational semantics

instrumented heap R must contain a right to access ℓ; moreover, in the case of a write access, this access right must be exclusive.

Lemma 3 (Typed access). *Every memory access is justified by a suitable access right.*

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

Theorem 4 (Data-race freedom). *A well-typed configuration is not racy.*

4 Locks

We extend the kernel calculus with dynamically-allocated locks. This extension is independent of the previous one (§3), although references and locks are of course intended to be used in concert.

Syntax. We extend the syntax as per Fig. 15. Values now include lock addresses k, which are implemented as natural numbers. (We allocate references and locks in two separate heaps, with independent address spaces.) Terms now include the three standard primitive operations on locks, namely allocating, acquiring, and releasing. Types now include the type $\textsf{lock}\ P$ of a lock whose invariant is the permission P. The type $\textsf{lock}\ P$ is duplicable, regardless of P. Types now also include the type $\textsf{locked}$. This type is not duplicable. It serves as a proof that a lock is held and (hence) as a permission to release the lock.

Operational semantics. We specify that a machine state s comprises a lock heap kh. A lock heap maps a valid lock address to a lock status: either U (unlocked) or L (locked). The reduction rules for locks appear in Fig. 16.

Assigning types to terms. We create new locks in the locked state, because this is more flexible: a lock of type $\textsf{lock}\ P$ can be created before the invariant P is established. The expression $\textsf{newlock}$ creates a new lock, say x, and produces the permissions $x \mathbin{@} \textsf{lock}\ P$ and $x \mathbin{@} \textsf{locked}$[5] (Fig. 17). The former guarantees that x is a lock and records its invariant, whereas the latter guarantees that x is

[5] In surface Mezzo, the type of newlock is written `(x: lock p | x @ locked)`.

$$\textsc{NewLock}\quad R; K; Q \vdash \mathsf{newlock} : \exists x : \mathsf{value}.(=x \mid (x \mathbin{@} \mathsf{lock}\ P) * (x \mathbin{@} \mathsf{locked}))$$

$$\textsc{Acquire}\quad R; K; v \mathbin{@} \mathsf{lock}\ P \vdash \mathsf{acquire}\ v : \top \mid P * (v \mathbin{@} \mathsf{locked}) \qquad \textsc{Release}\quad R; K; P * (v \mathbin{@} \mathsf{locked}) * (v \mathbin{@} \mathsf{lock}\ P) \vdash \mathsf{release}\ v : \top$$

Fig. 17. Locks: typing rules for terms

$$\textsc{Lock}\ \frac{R(k) = (P, \_)}{R; K; Q \vdash k : \mathsf{lock}\ P} \qquad \textsc{Locked}\ \frac{R(k) = (\_, X)}{R; K; Q \vdash k : \mathsf{locked}}$$

Fig. 18. Locks: typing rules for values

held and represents a permission to release it. The expressions $\mathsf{acquire}\ x$ and $\mathsf{release}\ x$ have the precondition $x \mathbin{@} \mathsf{lock}\ P$, which guarantees that x is a valid lock with invariant P. $\mathsf{acquire}\ x$ produces the permissions P and $x \mathbin{@} \mathsf{locked}$, whereas, symmetrically, $\mathsf{release}\ x$ requires (and consumes) these permissions.

The interaction between polymorphism and hidden state is unsound. When a new lock is allocated by newlock, its invariant (a permission P) becomes hidden, and it is necessary, at this point, to ensure that P is closed: newlock must not be allowed to execute under ForallIntro. This is why this rule is restricted to a class of *harmless* terms. This class does not contain any term of the form $D[\mathsf{newlock}]$; encompasses the values; and is stable by substitution and reduction. It is nevertheless possible to use the typing rule NewLock with a permission P that is not closed, as illustrated by `hide` (§1).

Resources. An *instrumented lock status* is a pair of a closed permission P and an access right, one of $\lightning$, N, and X. (These are the same as the instrumented values of §3, except this time X does not carry an argument and D does not appear.) The permission P is the lock invariant. The access right indicates whether releasing the lock is permitted: N represents no right, whereas X means that the lock is held and represents an exclusive right to release the lock. Instrumented lock statuses form a monotonic separation algebra, where, e.g., $(P, X) \star (P, N)$ is (P, X). That is, the lock invariant is shared (and immutable) information, whereas the ownership of a held lock is exclusive.

A *lock resource* is $\lightning$ or an instrumented lock status heap. Lock resources form a monotonic separation algebra. 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 agreement between a lock heap and a lock resource.

To summarize, if we extend the kernel with both references (§3) 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 an instrumented value heap and an instrumented lock heap. The agreement relation s *and* R *agree* requires agreement between each heap and the corresponding instrumented heap.

Hidden state. One might expect the correspondence relation $s \sim R$ to be just agreement, i.e., *s and R agree*, as in the previous section (§3). However, there is something more subtle to 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. While the lock is unlocked, the resource that justifies P is not available to any thread.

This leads us to refine our understanding of the correspondence $s \sim R$. The assertion should no longer mean that R is the entire instrumented (value/lock) heap; instead, it should mean that R is the fragment of the instrumented heap that is visible to the program, while the rest is hidden.

To account for this, we define the relation $s \sim R$ as follows.

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

The machine state s represents the entire (value/lock) heap. Thus, the agreement assertion *s and $R \star R'$ agree* implies that $R \star R'$ represents the entire instrumented (value/lock) heap. We split this resource between a visible part R, which appears in the conclusion, and a hidden part R', which must justify the conjunction of the invariants of all currently unlocked locks. This conjunction is constructed by inspection of $R \star R'$. We omit its definition, and denote it $\mathsf{hidden\ invariants\ of}\ (R \star R')$.

Assigning types to values. The typing rules Lock and Locked (Fig. 18) assign types to lock addresses, thus giving meaning to the types $\mathsf{locked}\,P$ and locked. Their premises look up the (lock) resource R. A lock address k whose invariant (as recorded in R) is P receives the type $\mathsf{lock}\,P$. A lock address k whose access right (as recorded in R) is X receives the type locked.

Soundness. A configuration is now deemed acceptable if every thread either (i) has reached a value; or (ii) is waiting on a lock that is currently held; or (iii) is able to take a step. The statements of type soundness are unchanged. Well-typed programs remain acceptable (§2.7) and are data-race free (§3).

5 Related work

Mezzo has close ties with L^3 [1]. Both are affine λ-calculi with strong references. They distinguish between a pointer and a capability to dereference it; the former is duplicable, the latter affine. Both record must-alias information via singleton types. However, Mezzo is meant to be a surface language, as opposed to a low-level calculus, and this leads to different designs. For instance, L^3 has type-level names ϱ for values, whereas, for greater conciseness and simplicity, Mezzo allows types to depend directly on values. Also, L^3 views capabilities as unit values, which one hopes can be erased by the compiler, whereas Mezzo views permissions as purely static entities, and has no syntax for manipulating them.

Mezzo is strongly inspired by separation logic [13] in its treatment of heap-allocated data and by concurrent separation logic [10] and its successors [8,3] in its treatment of locks. Like second-order separation logic, as found at the core of CaReSL [14], Mezzo supports higher-order functions and quantification over permissions (assertions) and types (predicates). Our duplicable permissions are analogous to Turon *et al.*'s necessary assertions, and our function `hide` (§1) is essentially identical to their `mkSync` [14, §3.2].

Although the formalization of Mezzo was carried out independently, and in part grew out of earlier work by the second author [11], it is in several ways closely related to the Views framework [7]. 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.

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 [5, 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.* [6] 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 [9]. 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.

6 Conclusion

We have presented a formalisation of three basic layers of Mezzo, namely:

- a concurrent call-by-value λ-calculus, equipped with an affine, polymorphic, value-dependent type-and-permission system;
- an extension with strong (i.e., affine, uniquely-owned) mutable references;
- an extension with dynamically-allocated, shareable locks.

This paper is accompanied with a Coq proof [2], which covers just these three layers. It is about ten thousand (non-blank, non-comment) lines of code. Out of

this, a de Bruijn index library and a monotonic separation algebra library, both of which are reusable, occupy about 2Kloc each. The remaining 6Kloc are split between the kernel (4Kloc), references (1Kloc), and locks (1Kloc).

The full Mezzo language offers more features, including richer memory blocks, carrying a tag and multiple fields; the possibility of turning a mutable block into an immutable one; iso-recursive types; and adoption and abandon [12], a mechanism that allows the unique-owner policy to be relaxed and enforced in part at runtime. All of these features are covered by an older Coq proof. In the future, we plan to port these features into the new proof without compromising its modularity. In particular, we wish to revisit the treatment of adoption and abandon so as to better isolate it from the treatment of memory blocks.

References

1. Ahmed, A., Fluet, M., Morrisett, G.: L^3: A linear language with locations. Fundamenta Informaticæ 77(4), 397–449 (2007)
2. Balabonski, T., Pottier, F.: A Coq formalization of Mezzo (Dec 2013), `http://gallium.inria.fr/~fpottier/mezzo/mezzo-coq.tar.gz`
3. Buisse, A., Birkedal, L., Støvring, K.: A step-indexed Kripke model of separation logic for storable locks. Electronic Notes in Theoretical Computer Science 276, 121–143 (2011)
4. Charguéraud, A., Pottier, F.: Functional translation of a calculus of capabilities. In: International Conference on Functional Programming (ICFP). pp. 213–224 (2008)
5. Chlipala, A.: Certified Programming and Dependent Types. MIT Press (2013)
6. Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: Principles of Programming Languages (POPL). pp. 207–218 (2013)
7. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: Principles of Programming Languages (POPL). pp. 287–300 (2013)
8. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. Tech. Rep. MSR-TR-2007-39, Microsoft Research (2007)
9. Mosses, P.D.: Modular structural operational semantics. Journal of Logic and Algebraic Programming 60–61, 195–228 (2004)
10. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1–3), 271–307 (2007)
11. Pottier, F.: Syntactic soundness proof of a type-and-capability system with hidden state. Journal of Functional Programming 23(1), 38–144 (2013)
12. Pottier, F., Protzenko, J.: Programming with permissions in Mezzo. In: International Conference on Functional Programming (ICFP). pp. 173–184 (2013)
13. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
14. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: International Conference on Functional Programming (ICFP). pp. 377–390 (2013)