Type soundness for Core Mezzo

François Pottier
INRIA

***Abstract*—Mezzo is a programming language in the tradition of ML. It offers algebraic data types, first-class functions, and a system of duplicable or affine permissions that controls aliasing and access to mutable memory. We present a formal definition of Core Mezzo, an explicitly-typed calculus that underlies Mezzo, and establish the soundness of its type and permission system. Our definitions and proofs have been machine-checked.**

I. A glimpse of Mezzo

The programming language Mezzo [1], designed by Jonathan Protzenko and by the author, is equipped with a static type and permission discipline that aims to better control the use of mutable memory. Compared with the type systems of ML or Java, Mezzo is in some ways more restrictive (because a mutable memory block cannot be read or written without a permission) and in other ways more expressive (because, with an appropriate permission, it is possible to "change the type" of a mutable memory block, yielding an updated permission). Thus, Mezzo is intended to help catch more programming errors and to enable certain idioms that traditional typed programming languages forbid.

Mezzo has user-defined algebraic data types, which serve as types for memory blocks. It distinguishes immutable blocks, which have "an arbitrary number of readers", and mutable blocks, which have "a unique owner". For instance, line 1 of Figure 1 defines a type of references, or mutable memory cells of one field. In short, a value `r` of type "`ref a`" is a pointer to a mutable block whose `contents` field currently contains a value of type `a`.

When we write that `r` "has type" `ref a`, we mean that a permission to use `r` at this type, written "`r @ ref a`", is presently available. At different program points, different permissions for `r` may be available: `r` does not necessarily have a fixed type throughout its scope. The programmer declares which permissions must be available upon entry and exit of every function, and the type-checker ideally[1] infers which permissions are available at each point within the function body. Permissions do not exist at runtime.

The function `sswap` in figure 1 expects two parameters `r` and `s` and returns a unit result. It requires the permission "`r @ ref a * s @ ref b`" and returns another permission, namely "`r @ ref b * s @ ref a`". These permissions are analogous to a pre- and postcondition in separation logic [2]. At mutable data types, conjunction `*` is separating: thus, the function `sswap` can be applied only to two distinct cells. Its type, which is polymorphic in `a` and `b`, reflects its runtime behavior: it performs a strong update, that is, it "changes the types" of `r` and `s`.

[1] At present, we have a working yet imperfect implementation of permission inference, which is outside the scope of this paper.

```
mutable data ref a = Ref { contents: a }                          1
                                                                  2
val sswap [a, b]                                                  3
  (consumes r: ref a, consumes s: ref b)                          4
: (| r @ ref b * s @ ref a)                                       5
=                                                                 6
  let tmp = r.contents in                                         7
  r.contents <- s.contents;                                       8
  s.contents <- tmp                                               9
                                                                  10
mutable data pool a = Pool {} adopts ref a                        11
                                                                  12
val wswap [a] (p: pool a, r: dynamic, s: dynamic) : () =          13
  if r != s then begin                                            14
    take r from p;                                                15
    take s from p;                                                16
    sswap (r, s);                                                 17
    give s to p;                                                  18
    give r to p                                                   19
  end                                                             20
```

Fig. 1. Two Mezzo functions for swapping two references

Because conjunction is separating, "`r @ ref a`" does not imply "`r @ ref a * r @ ref a`". That is, the permission "`r @ ref a`" cannot be duplicated: it is affine. Thus, for each reference cell, there exists at most one permission, which represents the "ownership" of the cell and carries the type of its content.

The purely static fragment of Mezzo's type and permission discipline has limited expressiveness. It can describe tree-structured heap fragments, but cannot express more complex aliasing patterns. In order to work around this limitation without requiring the programmer to provide complex logical assertions and proofs, Mezzo offers an "escape hatch", which involves runtime checks.

In line 11 of figure 1, for instance, we declare an algebraic data type "`pool a`". An object of this type is able to "adopt" an arbitrary number of cells of type `ref a`. (It has no fields; only its address matters.) Two instructions, **`give`** and **`take`**, allow a cell to be adopted by or taken away from a pool. These instructions maintain, at runtime, a pointer from each cell to its adopter (if any). The instruction "**`give`** `r` **`to`** `p`" sets this pointer. The instruction "**`take`** `r` **`from`** `p`" checks that it points to `p`, then clears it. Thus, **`take`** fails if the cell `r` is not currently adopted by `p`. The possibility of a failure is the price to pay for the flexibility offered by this mechanism.

The function `wswap` expects a pool `p` and two pointers `r` and `s`, which the programmer knows (or believes) are currently adoptees of `p`. The type `dynamic` describes a pointer to

an arbitrary object in the heap. At this type, conjunction is not separating: hence, `r` and `s` may or may not be distinct. The type `dynamic` does not have an ownership reading: the permission "`r @ dynamic`" does not imply that the object at address `r` can be read or written. In the body of `wswap`, the **`take`** instructions check (at runtime) that `r` and `s` are members of (i.e., adopted by) the pool `p`, and take them out of the pool. After these instructions, the permissions "`r @ ref a`" and "`s @ ref a`" are available, so it is legal to call `sswap`. These permissions are preserved by the call and consumed by the **`give`** instructions, which return `r` and `s` to the pool. The dynamic test `r != s` is not an optimization: it is required, because an attempt to **`take`** a cell twice would fail at runtime.

By lack of space, we cannot fully illustrate the use and expressive power of Mezzo. A separate paper [3] offers a slower-paced presentation, with motivating examples. Our purpose in the present paper is to give a definition of Mezzo's type and permission discipline and to establish its soundness.

II. Overview of the paper

We formalize Core Mezzo, a calculus that is slightly simpler and lower-level than Mezzo. We do not formalize Mezzo itself or the translation from Mezzo down to Core Mezzo, which are outside of the scope of this paper. Informally, the main differences between the two are:

- Core Mezzo has simpler (more verbose) types, which often involve polymorphism[2] and singleton types. For instance, in Core Mezzo, the type of `sswap` would be:

  $\forall r, s : \mathsf{value}.\quad \forall a, b : \mathsf{type}.$
  $(=r, =s \mid r \mathbin{@} \mathsf{ref}\ a * s \mathbin{@} \mathsf{ref}\ b) \to (() \mid r \mathbin{@} \mathsf{ref}\ b * s \mathbin{@} \mathsf{ref}\ a)$
- Mezzo has named data constructors and fields, whereas Core Mezzo numbers them sequentially;
- Mezzo offers an ML-like **`match`** construct, whereas Core Mezzo has a simpler **`switch`** construct, which performs case analysis but does not bind any variables;
- the syntax of Core Mezzo is designed in such a way that every intermediate computation must be named.

The paper is organized as follows. We present the syntax (§III) of the programs, types, and permissions. We equip Core Mezzo with an operational semantics (§IV). Then, we define the type and permission system (§V), and we sketch a proof of its soundness (§VI). We conclude with a discussion of some of the related work (§VII).

All definitions and statements have been machine-checked using the Coq proof assistant, and are available online [4]. The versions presented in this paper are manually transcribed and contain a few abuses of notation. In particular, we use an informal "nominal" notation for variables, whereas the mechanized proof uses de Bruijn indices.

We believe the main contributions of this paper are:

- a relatively simple definition and proof for a rich system, which supports a realistic surface programming language;
- the novel concept of adoption and abandon, together with its proof of soundness.

[2]Polymorphism is omitted from this paper, but is present in the Coq proof.

$m ::=$	*Modes*
$\mid D$	duplicable (immutable)
$\mid X$	exclusive (mutable)
$v, t, u, T, U, P, Q ::=$	*Everything*
$\mid x$	variable
	Terms
$\mid \lambda x.t$	function
$\mid v\ t$	function application
$\mid \ell$	memory location
$\mid \mathsf{new}\ i\ \{\vec{v}\}$	memory allocation
$\mid v.f$	field access
$\mid v.f \leftarrow v$	field update
$\mid \mathsf{switch}\ (v)\ \vec{t}$	case analysis
$\mid \mathsf{tag\ of}\ v \leftarrow i$	tag update
$\mid \mathsf{give}\ v\ \mathsf{to}\ v$	adoption
$\mid \mathsf{take}\ v\ \mathsf{from}\ v$	abandon
$\mid \mathsf{fail}$	runtime failure
	Types
$\mid T \to U$	function type
$\mid m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U$	memory block type
$\mid \mathsf{dynamic}$	arbitrary pointer type
$\mid \tau$	algebraic data type
$\mid =v$	singleton type
$\mid T \mid P$	qualified type
$\mid \bot$	empty type
	Permissions
$\mid v \mathbin{@} T$	atomic permission
$\mid \mathsf{empty}$	empty permission
$\mid P * P$	permission conjunction
$\mid \exists x.P$	existential permission
	Sequences
$\mid \epsilon$	empty sequence
$\mid t; \vec{t}$	non-empty sequence
$\kappa ::=$	*Kinds*
$\mid \mathsf{prog} \mid \mathsf{value} \mid \mathsf{type} \mid \mathsf{perm} \mid \mathsf{seq}\ \kappa$	

Fig. 2. Syntax of programs, types, and permissions

III. Syntax

Programs, types, and permissions form a single syntactic category, whose definition appears in figure 2. We let v, t, u, T, U, P, Q range over this category, where, by convention, we use v for "values", t and u for "programs" or "terms", T and U for "types", P and Q for "permissions". A kind system is imposed later on (§III-G) so as to give formal meaning to these sub-categories. There is a single name space of variables.

A. Programs

The "program" or "term" fragment of the syntax forms a fairly standard λ-calculus. It is untyped: terms do not refer to types or permissions.

There are only two forms of values, namely functions $\lambda x.t$ and memory locations ℓ. A memory location is a natural number, which represents an address in the heap.

As we will see (§IV), a memory block contains an integer tag i and a number of fields, each of which holds a value. For this reason, the memory allocation instruction "$\mathsf{new}\ i\ \{\vec{v}\}$" specifies the desired values of the tag and fields. In the field access and field update expressions, the field f is an

integer number. The tag of a block is examined via the C-like construct "switch (v) $\vec{t}$", which performs case analysis and transfers control to an appropriate branch, and can be updated via the instruction "tag of $v \leftarrow i$".

"give v_1 to v_2" and "take v_1 from v_2" are the adoption and abandon instructions (§I). Every block implicitly includes a special field, known as the *adopter* field, which contains either *null* or a valid memory location. The value of this field changes from *null* to non-*null* during adoption and from non-*null* to *null* during abandon. Abandon can fail at runtime.

Sequencing "let $x = u$ in t" is sugar for "$(\lambda x.t)\ u$". The right-hand side of applications is the only evaluation context.

B. Types

A function type "$T \to U$" describes a function that maps an argument of type T to a result of type U.

A memory block type "$m\ i\ \{\vec{T}\}$ adopts U", also known as a "structural" type, describes the address of a block whose tag is currently i and whose fields currently have types $\vec{T}$. The *mode* m is either *D*, which indicates that this block is immutable and shared by arbitrarily many readers, or *X*, which means that this block is mutable and has an exclusive owner. The clause "adopts U" indicates that this block may have adoptees of type U. In the particular case where U is the empty type $\bot$, this block in fact does not have any adoptees. We use the words "block" and "object" interchangeably.

The type "dynamic" describes the address of a block that has an implicit *adopter* field. For simplicity, we adopt the convention that every block has such a field, so a value has type dynamic if and only if it is a (valid) memory location. The only operation permitted by the type dynamic is take.

We assume that τ ranges over a fixed set of (names for) algebraic data types[3]. We further assume that each τ comes with a definition, which consists of:

- a mode, $\mathit{mode}(\tau)$, which indicates whether a block of type τ is immutable or mutable;
- an integer n and a function of $[0, n)$ to sequences of types; this means that the valid tags for a block of type τ are the elements of $[0, n)$ and tells, for every such tag i, how many fields exist and what their types are; we write "τ *has a branch* $i\ \{\vec{T}\}$" when this function maps i to $\vec{T}$.
- a type, $\mathit{adopts}(\tau)$; this is the type of the objects that can be adopted by a block of type τ.

The algebraic data type τ is isomorphic to the sum of the structural types "$m\ i\ \{\vec{T}\}$ adopts T", where m is $\mathit{mode}(\tau)$, τ *has a branch* $i\ \{\vec{T}\}$, and T is $\mathit{adopts}(\tau)$.

We write "τ *is a record* $\{\vec{T}\}$" if τ admits just one tag (namely the tag 0) and τ *has a branch* $0\ \{\vec{T}\}$.

For instance, assuming that *int* is the type of integers, an algebraic data type *ref* of integer references would be characterized by: $\mathit{mode}(\mathit{ref}) = X$, "*ref is a record* $\{\mathit{int}; \epsilon\}$", and $\mathit{adopts}(\mathit{ref}) = \bot$. An algebraic data type *pool* analogous to that of figure 1 would be characterized by: $\mathit{mode}(\mathit{pool}) = X$, "*pool is a record* $\{\epsilon\}$", and $\mathit{adopts}(\mathit{pool}) = \mathit{ref}$.

[3]In the current proof, algebraic data types are not parameterized.

Core Mezzo does not have built-in product types, sum types, reference types, or recursive types, because these notions are subsumed by the combination of structural types and algebraic data types. In particular, the unit type $()$ is sugar for the structural type "*D* $0\ \{\epsilon\}$ adopts $\bot$", and the unit term $()$ is sugar for the memory allocation expression "new $0\ \{\epsilon\}$".

If v is a value, then "$=v$" is a type, whose sole inhabitant is the value v. Singleton types are used to keep track of must-alias relationships.

The type "$T \mid P$" can be thought of as a product type, whose left-hand component is a type and whose right-hand component is a permission. It is typically used to express rich function types, such as $(T \mid P) \to (U \mid Q)$, which describes a function of T to U with precondition P and postcondition Q. Permissions do not exist at runtime: a value of type $T \mid P$ and a value of type T have the same runtime representation. We write $(\mid P)$ as a short-hand for $(() \mid P)$.

C. Permissions

An atomic permission "$v \mathbin{@} T$" can be viewed as an assertion that the value v currently has type T, or as a permission to use v at type T. In particular, if T is a structural type, then "$v \mathbin{@} T$" is an assertion about the current contents of the block at address v as well as a permission to read and (if T indicates that this block is mutable) to update this block. It is analogous to a "points-to" assertion in separation logic [2].

An atomic permission that involves a singleton type, such as "$v_1 \mathbin{@} {=}v_2$", means that v_1 admits the singleton type $=v_2$. In other words, it means that v_1 and v_2 denote the same value. We write "$v_1 = v_2$" as sugar for such a permission.

"$P * Q$" is the conjunction of the permissions P and Q. The permission "empty" is a left and right unit for conjunction.

An existential permission "$\exists x.P$" asserts that there exists a value, denoted by the variable x, such that P holds. Existential permissions offer a formalism for explaining how the Mezzo type-checker generates "fresh" auxiliary variables when breaking composite permissions into smaller pieces (DECOMPOSEBLOCK, §V-B).

D. Sequences

Sequences of values $\vec{v}$, sequences of terms $\vec{t}$, and sequences of types $\vec{T}$ are part of the single, all-encompassing syntactic category. This is achieved simply by adding the productions $t ::= \epsilon$ and $t ::= t; \vec{t}$ to the syntax.

E. Duplicable types and permissions

Mezzo is affine: the duplication of permissions is subject to certain restrictions. In the tradition of Linear Logic, many affine type systems in the literature adopt the convention that every type is by default affine (non-duplicable) and that the duplicable types are marked with an explicit "!" modality. Here, instead, this information is implicit. The predicate "T *is duplicable*", which is co-inductively defined by the rules in figure 3, identifies a subset of the types and permissions that are considered inherently duplicable. Duplication, when permitted, is implicit: it takes the form of a permission subsumption axiom (DUPLICATE, §V-B).

$$T \to U \text{ is duplicable} \qquad \frac{\vec{T} \text{ is duplicable}}{D\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U \text{ is duplicable}} \qquad \mathsf{dynamic} \text{ is duplicable} \qquad \frac{\begin{array}{c}mode(\tau) = D \\ \text{for every } i \text{ and } \vec{T} \text{ such that } \tau \text{ has a branch } i\ \{\vec{T}\},\ \vec{T} \text{ is duplicable}\end{array}}{\tau \text{ is duplicable}}$$

$$=v \text{ is duplicable} \qquad \frac{T \text{ is duplicable} \qquad P \text{ is duplicable}}{T \mid P \text{ is duplicable}} \qquad \bot \text{ is duplicable} \qquad \frac{T \text{ is duplicable}}{v \mathbin{@} T \text{ is duplicable}} \qquad \mathsf{empty} \text{ is duplicable}$$

$$\frac{P \text{ is duplicable} \qquad Q \text{ is duplicable}}{P * Q \text{ is duplicable}} \qquad \frac{P \text{ is duplicable}}{\exists x.P \text{ is duplicable}} \qquad \epsilon \text{ is duplicable} \qquad \frac{T \text{ is duplicable} \qquad \vec{T} \text{ is duplicable}}{T; \vec{T} \text{ is duplicable}}$$

Fig. 3. Co-inductive definition of the duplicable types

Let us comment on the first four rules in figure 3. Because we do not wish to impose a visually cumbersome distinction between duplicable and non-duplicable functions, we adopt the convention that function types are duplicable[4]. The structural types whose mode is D are duplicable, while those whose mode is X are not. The type dynamic is duplicable. This allows creating multiple pointers to a block, albeit none of these pointers comes with a read or write access right. An algebraic data type τ is duplicable if and only if its unfolding is duplicable. The presence of this rule is the reason why the judgement "T *is duplicable*" must be co-inductively defined. An inductive definition would be sound, but too strict.

F. Adoption as a relation between types

A type is *exclusive* if it is of the form "$X\ i\ \{\vec{T}\}$ adopts U" or "τ" where $mode(\tau) = X$.

It is convenient to define a relation between types, written "T *adopts* U", which means that an object of type T may have adoptees of type U. It is defined as follows:

1) if U is exclusive, then the block type "$X\ i\ \{\vec{T}\}$ adopts U" *adopts* U;
2) if $adopts(\tau) = U$ and $mode(\tau) = X$ and U is exclusive, then the algebraic data type τ *adopts* U.

This definition means, in particular, that (a) adopters and adoptees must have exclusive type; and (b) the type of the adopter determines the type of the adoptee. Both of these conditions are required for soundness.

G. Kinds

The *kinds* of figure 2 define the following subsets of the syntactic universe.

The kind prog classifies source *programs* and programs under execution. Memory locations ℓ are inaccessible to the programmer and appear only during execution.

The kind value classifies *values*. The values include λ-abstractions, memory locations, and variables. The kind value forms a subset of the kind prog: a value is a program.

The kinds type and perm classify *types* and *permissions*. There are two ways in which types and permissions can depend on (that is, refer to) values. One is the singleton type $=v$; the other is the atomic permission $v \mathbin{@} T$.

The kind seq κ classifies a sequence of things of kind κ.

The well-kindedness judgement takes the form $K \vdash t : \kappa$, where a *kind environment* K maps variables to kinds, t is a piece of syntax, and κ is a kind. Its definition appears in an appendix (§A). The empty kind environment is written $\varnothing$.

IV. Dynamic semantics

Core Mezzo is equipped with a small-step operational semantics. It is a standard call-by-value λ-calculus, equipped with mutable, heap-allocated memory blocks, and extended with *adopter* fields and give and take operations.

An *adopter pointer* p is either *null* or a memory location ℓ.

A *block* $\langle\, i \mid p \mid \vec{v} \,\rangle$ is a triple of an integer tag i, an adopter pointer p, and a sequence $\vec{v}$ of closed values.

A *heap* h is a function of an initial segment of the natural numbers to blocks. We write $\emptyset$ for the empty heap. We write *limit* h for the first unallocated address in the heap h. We write $h + b$ for the heap that extends h with a mapping of *limit* h to the block b. If the memory location ℓ is in the domain of the heap h, then we write $h[\ell \mapsto b]$ for the heap that maps ℓ to b and agrees with h elsewhere. We use this operation only when the new block b has the same length as the previous block $h(\ell)$, so that Core Mezzo can be implemented on top of a runtime system that does not allow blocks to be resized.

A *configuration* $h \,/\, t$ pairs a heap h and a closed term t.

The small-step operational semantics appears in figure 4.

The reduction rule for memory allocation extends the heap with a new block $\langle\, i \mid \mathit{null} \mid \vec{v} \,\rangle$ and returns its address.

In the reduction rules for field access and field update, we write $\vec{v}$ for a sequence of values that has a hole in its f-th position, and we write $\vec{v}\,[f := v]$ for the sequence obtained by filling this hole with the value v.

The same notation is used in the reduction rule for switch. The heap is looked up at address ℓ. The tag i that is found there is used to select the i-th branch of the switch construct.

The tag update instruction changes the tag of the block found at address ℓ from i to i'.

The give instruction expects its arguments to be addresses ℓ' and ℓ and expects the *adopter* field of the block at ℓ' to contain the value *null*. The value ℓ is written to this field, so as to record the fact that ℓ' has been adopted by ℓ.

[4] In principle, a type of functions-that-can-be-used-at-most-once can be encoded as a package of type $\exists P.(((T \mid P) \to U) \mid P)$. However, existential quantification over permissions is not yet part of the formalization.

initial configuration		*new configuration*		*side condition*
$h \mathbin{/} v\ t$	$\longrightarrow h$		$\mathbin{/} v\ t'$	$h \mathbin{/} t \longrightarrow h' \mathbin{/} t'$
$h \mathbin{/} v$ fail	$\longrightarrow h$		$\mathbin{/}$ fail	
$h \mathbin{/} (\lambda x.t)\ v$	$\longrightarrow h$		$\mathbin{/} [v/x]t$	$\varnothing \vdash v$: value
$h \mathbin{/}$ new $i\ \{\vec{v}\}$	$\longrightarrow h + \langle\, i \mid \mathit{null} \mid \vec{v} \,\rangle$		$\mathbin{/}$ *limit* h	
$h \mathbin{/} \ell.f$	$\longrightarrow h$		$\mathbin{/} v$	$h(\ell) = \langle\, i \mid p \mid \vec{v}\,[f := v] \,\rangle$
$h \mathbin{/} \ell.f \leftarrow v'$	$\longrightarrow h[\ell \mapsto \langle\, i \mid p \mid \vec{v}\,[f := v'] \,\rangle]$		$\mathbin{/} ()$	$h(\ell) = \langle\, i \mid p \mid \vec{v}\,[f := v] \,\rangle$
$h \mathbin{/}$ switch $(\ell)\ (\vec{t}[i := t])$	$\longrightarrow h$		$\mathbin{/} t$	$h(\ell) = \langle\, i \mid p \mid \vec{v} \,\rangle$
$h \mathbin{/}$ tag of $\ell \leftarrow i'$	$\longrightarrow h[\ell \mapsto \langle\, i' \mid p \mid \vec{v} \,\rangle]$		$\mathbin{/} ()$	$h(\ell) = \langle\, i \mid p \mid \vec{v} \,\rangle$
$h \mathbin{/}$ give ℓ' to ℓ	$\longrightarrow h[\ell' \mapsto \langle\, i \mid \ell \mid \vec{v} \,\rangle]$		$\mathbin{/} ()$	$h(\ell') = \langle\, i \mid \mathit{null} \mid \vec{v} \,\rangle$
$h \mathbin{/}$ take ℓ' from ℓ	$\longrightarrow h[\ell' \mapsto \langle\, i \mid \mathit{null} \mid \vec{v} \,\rangle]$		$\mathbin{/} ()$	$h(\ell') = \langle\, i \mid \ell \mid \vec{v} \,\rangle$
$h \mathbin{/}$ take ℓ' from ℓ	$\longrightarrow h$		$\mathbin{/}$ fail	$h(\ell') = \langle\, i \mid p \mid \vec{v} \,\rangle \wedge p \neq \ell$

Fig. 4. Operational semantics

The take instruction also expects two locations. It tests whether the *adopter* field of the block at ℓ' contains the value ℓ. If so, *null* is written to this field, so as to record the fact that ℓ' is no longer adopted by ℓ. Otherwise, the instruction fails.

An *answer* is a configuration of the form $h \mathbin{/} v$, where v is a value, or $h \mathbin{/}$ fail. A configuration that cannot take a reduction step, but is not an answer, is *stuck*. The central result of this paper is that *a well-typed program cannot reach a stuck configuration*. It *can*, however, fail. Unless the programmer has used fail in the source code, the only way in which fail can arise is via the last reduction rule, where one attempts to execute "take ℓ' from ℓ" and finds that the *adopter* field at ℓ' does not contain ℓ. The static discipline does not rule out this error.

V. Static semantics

A. Typing judgement and interpretation of permissions

The main two judgements, which depend on each other, are:

1) the *typing judgement* $R; K; P \vdash t : T$;
2) the *permission interpretation judgement* $R; K \Vdash P$.

When type-checking a source program, the parameter R is trivial (it is the empty resource *void*). Non-trivial *resources* R are used only when considering programs under execution. In the figures that define the above judgements (figures 5, 6, and 7), the aspects that concern programs under execution are shaded and can be ignored for the moment. They are explained later on (§VI).

The typing judgement $R; K; P \vdash t : T$ states that, under the assumption represented by the permission P, the term t has type T. It is analogous to a Hoare triple: P and T can be thought of as a pre- and postcondition.

The permission interpretation judgement $R; K \Vdash P$ states that the permission P is valid. If one ignores the parameter R, one can take this to mean that P is "true": in particular, the permission $v \mathbin{@} T$ is "true" if v has type T. We revisit this intuitive interpretation later on (§VI-B).

The typing judgement is inductively defined by two groups of rules, which appear in figures 5 and 7. If a derivation uses only the rules in the first group (except possibly under λ), then it is a *canonical derivation*, and we write $R; K; P \Vdash v : T$. (In that case, the term must be a value v.) If a derivation uses the rules in both groups, we write $R; K; P \vdash t : T$.

Let us first describe the "canonical" typing rules (figure 5). With the exception of Cut, each of these rules introduces one of the type constructors in figure 2. Thus, these rules "give meaning" to the type constructors. Function separately extends the kind environment with the binding x : value and the precondition P with the assumption $x \mathbin{@} T$. The requirement that P be duplicable stems from our decision to consider function types duplicable (§III-E). Because a function can be invoked arbitrarily many times, it must not capture a non-duplicable permission. Block and Dynamic concern memory locations. We defer their explanation to §VI-C. Fold states that v admits the type τ if and only if it admits some structural unfolding of τ. Singleton states that v is one (and the only) inhabitant of the singleton type $=v$. Frame is analogous to the frame rule of separation logic [2]. If one ignores the parameters R_1 and R_2, Cut states that if t is well-typed under the composite precondition $P_1 * P_2$ and if P_1 happens to be "true" then t is well-typed under just P_2.

Atomic (figure 6) states, roughly, that the permission $v \mathbin{@} T$ is "true" if the value v has type T under an empty precondition. (A cut, in the style of the typing rule Cut, is allowed.) The type derivation for v must be canonical. This is an important technical point. An arbitrary type derivation can make use of the permission subsumption relation (§V-B), whose soundness will be established relative to the interpretation of permissions (lemma A.9). Because the interpretation of permissions rests upon canonical type derivations only, we avoid a circularity.

The non-canonical rules (figure 7) are (a) syntax-directed rules for constructs that are not values; (b) non-syntax-directed rules that cannot occur in canonical derivations.

Application is standard, except in the manner in which the requirements about the sub-terms 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 formulation makes Cut the only typing rule whose conclusion involves a join of two resources $R_1 \star R_2$.

New states that "new $i\ \{\vec{v}\}$" produces a memory location of structural type "$m\ i\ \{\vec{T}\}$ adopts $\bot$". The mode m is arbitrary: the programmer chooses whether to create an immutable or mutable block. (In Mezzo, the programmer provides the name of a data constructor, out of which m and i are deduced.) The notation $\vec{v} \mathbin{@} \vec{T}$ is a short-hand for an iterated conjunction and is defined only if the sequences $\vec{v}$ and $\vec{T}$ have the same length. The clause adopts $\bot$ reflects the fact that a newly allocated block does not have any adoptees.

FUNCTION
$$\frac{K \vdash T : \mathsf{type} \qquad P \textit{ is duplicable} \qquad \widehat{R};\ K, x : \mathsf{value}; P * x \mathbin{@} T \vdash t : U}{R;\ K; P \diamond \lambda x.t : T \to U}$$

BLOCK
$$\frac{\begin{array}{c} R_1 \star R_2 = R \\ R_1;\ \varnothing \Vdash \vec{v} \mathbin{@} \vec{T} \qquad R_2 \vdash \ell \textit{ adopts } \vec{\ell'} \dashv R_2' \\ R_{21}' \star R_{22}' = R_2' \\ R_{21}';\ \varnothing \Vdash \vec{\ell'} \mathbin{@} U \qquad \varnothing \vdash U : \mathsf{type} \qquad R_{22}'(\ell) = m\ \langle i \mid \mathit{null} \mid \vec{v} \rangle \end{array}}{R;\ K; P \diamond \ell : m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U}$$

DYNAMIC
$$\frac{\ell < \textit{limit } R}{R;\ K; P \diamond \ell : \mathsf{dynamic}}$$

FOLD
$$\frac{R;\ K; P \diamond v : m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U \qquad \textit{mode}(\tau) = m \qquad \tau \textit{ has a branch } i\ \{\vec{T}\} \qquad \textit{adopts}(\tau) = U}{R;\ K; P \diamond v : \tau}$$

SINGLETON
$$\frac{K \vdash v : \mathsf{value}}{R;\ K; P \diamond v : {=}v}$$

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

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

Fig. 5. Typing rules: canonical fragment; $\diamond$ stands for one of $\Vdash$ (canonical) and $\vdash$ (non-canonical)

ATOMIC
$$\frac{K \vdash v : \mathsf{value} \qquad R_1;\ K; P \Vdash v : T \qquad R_2;\ K \Vdash P}{R_1 \star R_2;\ K \Vdash v \mathbin{@} T}$$

EMPTY
$$R;\ K \Vdash \mathsf{empty}$$

STAR
$$\frac{R_1;\ K \Vdash P_1 \qquad R_2;\ K \Vdash P_2}{R_1 \star R_2;\ K \Vdash P_1 * P_2}$$

EXISTS
$$\frac{K, x : \mathsf{value} \vdash P : \mathsf{perm} \qquad K \vdash v : \mathsf{value} \qquad R;\ K \Vdash [v/x]P}{R;\ K \Vdash \exists x.P}$$

Fig. 6. The interpretation of permissions

APPLICATION
$$\frac{K \vdash v : \mathsf{value} \qquad R;\ K; Q \vdash t : T}{R;\ K; (v \mathbin{@} (T \to U)) * Q \vdash v\ t : U}$$

NEW
$$\frac{K \vdash \vec{v} : \mathsf{seq\ value} \qquad K \vdash \vec{T} : \mathsf{seq\ type}}{R;\ K; \vec{v} \mathbin{@} \vec{T} \vdash \mathsf{new}\ i\ \{\vec{v}\} : m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ \bot}$$

READ
$$\frac{K \vdash v : \mathsf{value} \qquad T \textit{ is duplicable} \qquad P = v \mathbin{@} (m\ i\ \{\vec{T}[f := T]\}\ \mathsf{adopts}\ U)}{R;\ K; P \vdash v.f : T \mid P}$$

WRITE
$$\frac{K \vdash v_1 : \mathsf{value} \qquad K \vdash v_2 : \mathsf{value}}{\begin{array}{l} R;\ K; (v_1 \mathbin{@} (X\ i\ \{\vec{T}[f := T_1]\}\ \mathsf{adopts}\ U)) * (v_2 \mathbin{@} T_2) \vdash v_1.f \leftarrow v_2 : () \mid \\ \qquad (v_1 \mathbin{@} (X\ i\ \{\vec{T}[f := T_2]\}\ \mathsf{adopts}\ U)) \end{array}}$$

SWITCH
$$\frac{\begin{array}{c} K \vdash v : \mathsf{value} \qquad K \vdash T : \mathsf{type} \\ \textit{mode}(\tau) = m \qquad \textit{adopts}(\tau) = U \\ \textit{for every } i \textit{ and } \vec{T} \textit{ such that } \tau \textit{ has a branch } i\ \{\vec{T}\}, \\ R;\ K; P * (v \mathbin{@} (m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U)) \vdash \vec{t}(i) : T \end{array}}{R;\ K; P * (v \mathbin{@} \tau) \vdash \mathsf{switch}\ (v)\ \vec{t} : T}$$

WRITETAG
$$\frac{K \vdash v : \mathsf{value}}{\begin{array}{l} R;\ K; v \mathbin{@} (X\ i_1\ \{\vec{T}\}\ \mathsf{adopts}\ U) \vdash \mathsf{tag\ of}\ v \leftarrow i_2 : () \mid \\ \qquad (v \mathbin{@} (m\ i_2\ \{\vec{T}\}\ \mathsf{adopts}\ U)) \end{array}}$$

GIVE
$$\frac{K \vdash v_1 : \mathsf{value} \qquad K \vdash v_2 : \mathsf{value} \qquad T_2 \textit{ adopts } T_1}{\begin{array}{l} R;\ K; (v_1 \mathbin{@} T_1) * (v_2 \mathbin{@} T_2) \vdash \mathsf{give}\ v_1\ \mathsf{to}\ v_2 : () \mid \\ \qquad (v_2 \mathbin{@} T_2) \end{array}}$$

TAKE
$$\frac{K \vdash v_1 : \mathsf{value} \qquad K \vdash v_2 : \mathsf{value} \qquad T_2 \textit{ adopts } T_1}{\begin{array}{l} R;\ K; (v_1 \mathbin{@} \mathsf{dynamic}) * (v_2 \mathbin{@} T_2) \vdash \mathsf{take}\ v_1\ \mathsf{from}\ v_2 : () \mid \\ \qquad ((v_1 \mathbin{@} T_1) * (v_2 \mathbin{@} T_2)) \end{array}}$$

FAIL
$$\frac{K \vdash T : \mathsf{type}}{R;\ K; P \vdash \mathsf{fail} : T}$$

SUBLEFT
$$\frac{K \vdash P_1 \le P_2 \qquad R;\ K; P_2 \vdash t : T}{R;\ K; P_1 \vdash t : T}$$

SUBRIGHT
$$\frac{R;\ K; P \vdash t : T_1 \qquad K \vdash T_1 \le T_2}{R;\ K; P \vdash t : T_2}$$

EXISTSELIM
$$\frac{R;\ K, x : \mathsf{value}; P \vdash t : T}{R;\ K; \exists x.P \vdash t : T}$$

Fig. 7. Typing rules: non-canonical fragment

SUB
$$\frac{K, x : \mathsf{value} \vdash x \mathbin{@} T \le x \mathbin{@} U}{K \vdash T \le U}$$

SEQSUBNIL
$$K \vdash \epsilon \le \epsilon$$

SEQSUBCONS
$$\frac{K \vdash T \le U \qquad K \vdash \vec{T} \le \vec{U}}{K \vdash T; \vec{T} \le U; \vec{U}}$$

Fig. 8. Subsumption: types and sequences of types

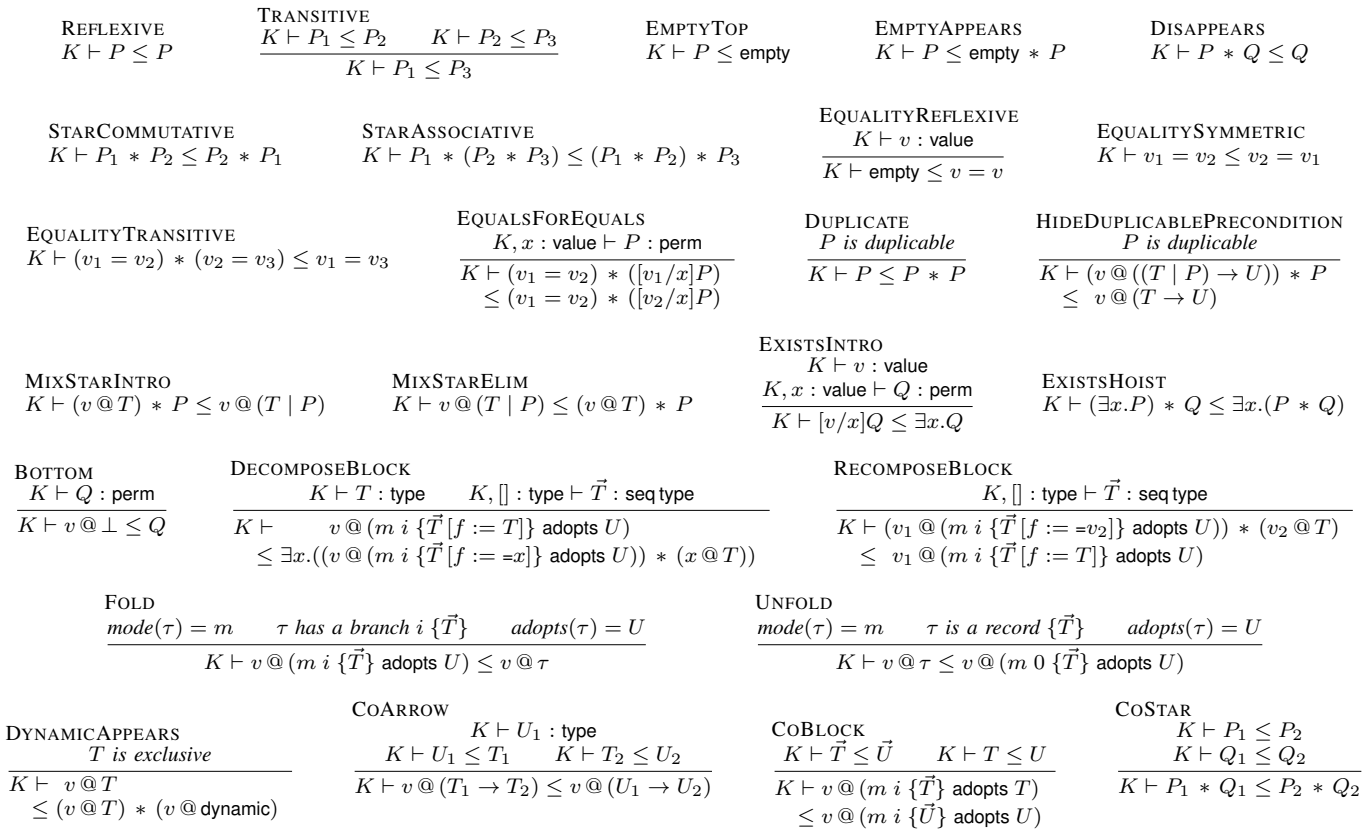

Fig. 9. Subsumption: permissions

A READ expression "$v.f$" requires v to be the address of a block whose field f has type T. (We write $\vec{T}[f := T]$ for a sequence of types whose f-th element is T.) This block may be immutable or mutable: m is unconstrained. Because reading a field creates a new copy of its content, the type T must be duplicable. No permission is consumed or altered: the pre- and postcondition are identical.

A WRITE instruction "$v_1.f \leftarrow v_2$" requires v_1 to be the address of a block whose field f has type T_1. An exclusive permission for v_1 is required, which ensures that "nobody else" has read or write access to this block. This allows changing the type of the field f to T_2, which, according to the precondition, is the type of v_2. The permission $v_2 \mathbin{@} T_2$ is consumed[5].

The SWITCH construct "$\mathsf{switch}\ (v)\ \vec{t}$" requires $v \mathbin{@} \tau$, where τ is an algebraic data type. The last premise checks that every branch is well-typed. (We write $\vec{t}(i)$ for the i-th element of the sequence $\vec{t}$.) A *permission refinement* step takes place: in the i-th branch, $v \mathbin{@} \tau$ is replaced with the more precise permission "$v \mathbin{@} (m\ i\ \{\vec{T}\}\ \mathsf{adopts}\ U)$", an unfolding of τ that incorporates the knowledge that the block's tag is i.

Tag update, "$\mathsf{tag\ of}\ v \leftarrow i_2$", is analogous to field update. An exclusive permission for v is required and is updated so as to record that the block now has tag i_2. Furthermore, WRITETAG allows a mode change, from X to an arbitrary m, so that, if the programmer so wishes, the block becomes immutable.

Adoption, "$\mathsf{give}\ v_1\ \mathsf{to}\ v_2$", requires the permissions $v_1 \mathbin{@} T_1$ and $v_2 \mathbin{@} T_2$. GIVE's premise "T_2 *adopts* T_1" means that objects of type T_2 may adopt objects of type T_1 (§III-F). $v_1 \mathbin{@} T_1$ is consumed: the ownership of the adoptee becomes implicitly bundled with that of the adopter. At any time, $v_2 \mathbin{@} T_2$ represents the ownership of v_2 and of its adoptees.

Abandon, "$\mathsf{take}\ v_1\ \mathsf{from}\ v_2$", requires $v_1 \mathbin{@} \mathsf{dynamic}$, which guarantees that v_1 is the address of a block, and implies that it is safe to read its *adopter* field. (This permission is not produced by give, but by subsumption; DYNAMICAPPEARS, §V-B.) Abandon also requires $v_2 \mathbin{@} T_2$, which represents the ownership of v_2. These permissions do not imply that v_1 is currently one of v_2's adoptees, which is why one checks, at runtime, that v_1's *adopter* field contains the address v_2. The success of this check, together with TAKE's premise "T_2 *adopts* T_1", implies that v_1 has type T_1. The value *null* is written to v_1's *adopter* field, so v_1 is no longer considered adopted by v_2. Accordingly, in the postcondition, we recover the permission $v_1 \mathbin{@} T_1$.

FAIL states that fail is well-typed in every context. The static discipline does not prevent this runtime failure.

The subsumption rules, SUBLEFT and SUBRIGHT, correspond to the consequence rule of Hoare logic. The subsumption

[5]One can derive a variant of WRITE where T_2 is required to be duplicable and the permission $v_2 \mathbin{@} T_2$ is not consumed. One can also derive a variant of WRITE where no permission about v_2 is required and T_2 is taken to be the singleton type $=v_2$.

relations $K \vdash P_1 \leq P_2$ and $K \vdash T_1 \leq T_2$ are described further on (§V-B). ExistsElim is a left elimination rule for the existential quantifier.

B. Subsumption

There are three subsumption judgements:

1) for permissions, $K \vdash P \leq Q$;
2) for types, $K \vdash T \leq U$;
3) for sequences of types, $K \vdash \vec{T} \leq \vec{U}$.

The last two judgements are trivial: subsumption of types is defined in terms of subsumption of permissions, and subsumption of sequences is defined pointwise (figure 8). Thus, we review only permission subsumption (figure 9). Reflexive and Transitive state that subsumption is a pre-order. EmptyTop and Disappears state that every permission is affine (i.e., can be discarded). EmptyAppears and Disappears state that empty is a unit for $*$. StarCommutative and StarAssociative state that $*$ is commutative and associative. EqualityReflexive, EqualitySymmetric, EqualityTransitive, and EqualsForEquals state that equality of values is an equivalence relation and satisfies Leibniz' principle. Recall that "$v_1 = v_2$" is sugar for "$v_1 \,@\, {=}v_2$" (§III-C). Duplicate states that a permission that is syntactically considered duplicable (§III-E) can in fact be duplicated. HideDuplicablePrecondition states that if the function v has precondition P, which is duplicable and available, then one may pretend that v has no precondition. This allows a closure to capture a duplicable permission *after* it has been constructed, whereas the typing rule Function of figure 5 allows a closure to capture a duplicable permission *when* it is constructed. MixStarIntro and MixStarElim introduce and eliminate the type constructor "$T \mid P$". ExistsIntro introduces the existential quantifier. (ExistsElim is a typing rule; see figure 7.) ExistsHoist hoists an existential quantifier out of a conjunction. Bottom states that $\bot$ is the least type. DecomposeBlock introduces a fresh name for a field. Informally, it says, "if $v.f$ has type T, then $v.f$ must be some value x such that x has type T". When T is not duplicable, this subsumption rule *must* be applied before the field can be read: indeed, the typing rule Read (figure 7) is restricted to duplicable types. Because the singleton type ${=}x$ is duplicable, Read *can* be applied after DecomposeBlock has been used. RecomposeBlock is the reverse of DecomposeBlock. Informally, it says, "if $v_1.f$ is v_2 and if v_2 has type T, then $v_1.f$ has type T". Fold is analogous to the typing rule by the same name (figure 5). Unfold unfolds an algebraic data type definition. It is restricted to the case where τ is a record type, i.e., there is only one branch. If there are multiple branches, no subsumption rule can unfold τ; a switch construct must be used instead. DynamicAppears states that if v has type T and T is exclusive, then v has type dynamic in addition to T. This rule is usually applied immediately before type-checking an instruction of the form "give v to ...". The adoption instruction consumes the permission $v \,@\, T$, but $v \,@\, \mathsf{dynamic}$ remains and can later be used to justify a take instruction. CoArrow, CoBlock, and CoStar state that subtyping is a congruence. Functions are contravariant in their domain and covariant in their codomain. Structural types are covariant in their fields and in the type of their adoptees. (This holds both for immutable and mutable blocks.) Conjunction is covariant. A rule stating that "$T \mid P$" is covariant in T and P can be derived.

This ends the definition of Core Mezzo. As a sanity check, we have proved that Core Mezzo subsumes the simply-typed λ-calculus, so it is not vacuous: see the appendix (§B).

VI. Soundness

In order to express the global invariant that is enforced by the type and permission system, we must provide typing judgements not just for source programs, but for programs under execution. This is done in four steps, as follows. We introduce a notion of *instrumented heap fragment*, or *resource* (§VI-A). We parameterize our judgements over a resource (§VI-B). We introduce rules for type-checking memory locations (§VI-C). Finally, we define a typing judgement for configurations and prove that the system is sound (§VI-D).

A. Instrumented heap fragments, or resources

An *instrumented heap fragment* R is a total function of an initial segment of the natural numbers to instrumented blocks. An *instrumented block* is N, $D\,b$, or $X\,b$, where b is a block. When R maps an address ℓ to N, this means that ℓ is not part of the heap fragment R, i.e., "we know nothing" about ℓ. When R maps ℓ to $D\,b$, this means that "we know, and everyone knows, that there is a block b" at address ℓ, and "everyone must preserve this fact", i.e., the block is immutable. When R maps ℓ to $X\,b$, this means that "we know there is a block b" at ℓ, and "no-one else knows anything" about ℓ, i.e., we can mutate this block in arbitrary ways without violating someone else's assumptions. In the last two cases, we say that ℓ is *in the domain* of R. The letters D and X respectively stand for "duplicable" and "exclusive" and correspond to the modes m that appear as part of structural types (figure 2).

Two instrumented blocks can be combined by a partial function $\star$, which is defined as follows:

$$N \star N = N \qquad N \star (X\,b) = X\,b$$
$$(D\,b) \star (D\,b) = D\,b \qquad (X\,b) \star N = X\,b$$

This function is extended pointwise to resources. It tells whether two partial views of the memory are compatible with each other. A total function $\widehat{\cdot}$, which maps an instrumented block to its "duplicable fragment", is defined as follows:

$$\widehat{N} = N \qquad \widehat{(D\,b)} = D\,b \qquad \widehat{(X\,b)} = N$$

A partial order $\lhd$ on instrumented blocks is defined as the least reflexive relation that satisfies $N \lhd (D\,b)$. This relation describes how an inactive principal's view of memory may evolve due to actions by other principals. The function $\widehat{\cdot}$ and the relation $\lhd$ are also extended to resources. Thus equipped, resources form a *monotonic separation algebra* [5].

In order to justify the soundness of adoption and abandon, we must further restrict our notion of resource. We impose two properties that every instrumented heap fragment must satisfy.

First, *a duplicable block cannot be adopted*. In an instrumented block $D\,b$, the block b must be of the form $\langle\, i \mid \mathit{null} \mid \vec{v} \,\rangle$, i.e., its adopter pointer must be *null*. Second, every instrumented heap fragment must be *closed under adopter edges*. That is, if R maps ℓ to $X\langle\, i \mid p \mid \vec{v} \,\rangle$, then the adopter pointer p must be either *null* or in the domain of R. In other words, a resource R cannot exhibit a dangling adopter pointer. The functions $\star$ and $\widehat{\cdot}$ preserve these two properties: so, resources, thus restricted, still form a monotonic separation algebra.

By imposing these restrictions, we promise to never split the heap in a way that would place an adopter and its adoptee in distinct heap fragments. This is exploited to justify that "if we own a block, then we own all of its adoptees too". This idea plays a key role in the proof of subject reduction for take.

B. Parameterizing the judgements with a resource

The judgements of figures 5, 6, and 7 are parameterized with a resource R. A judgement about a program depends upon the view of the heap that is granted to this program. The interpretation of permissions (figure 6) is closely related to the interpretation of assertions in separation logic: a heap fragment R justifies an assertion P. The function $\star$ is used in CUT, ATOMIC, and STAR. The function $\widehat{\cdot}$ is used in FUNCTION.

C. Rules for type-checking memory locations

The rules BLOCK and DYNAMIC in figure 5 assign types to memory locations. They define the meaning of the structural type "$m\ i\ \{\vec{T}\}$ adopts U" and of the type dynamic.

BLOCK is quite complex, because a structural type represents at the same time the ownership of: (a) a block, (b) the values stored in the block's fields, and (c) the block's adoptees. This rule can be read as follows. With respect to R, the address ℓ admits the type "$m\ i\ \{\vec{T}\}$ adopts U" if:

1) the heap fragment R can be split into R_1 and R_2, where:
2) R_1 justifies that the values $\vec{v}$ have types $\vec{T}$;
3) there exists a list $\vec{\ell'}$ of addresses such that[6], in the heap fragment R_2, the adoptees of ℓ are exactly the members of $\vec{\ell'}$, and if we set their *adopter* fields to *null*, yielding an updated heap fragment R'_2, then R'_2 can be split into R'_{21} and R'_{22}, where:
4) R'_{21} justifies that every member of $\vec{\ell'}$ has type U;
5) R'_{22} maps the address ℓ to an instrumented block which has mode m, tag i, no adopter, and fields $\vec{v}$.

DYNAMIC states that all valid memory locations (and only them) have type dynamic. This explains why, when we have a permission of the form x @ dynamic, it is safe to treat x as a pointer and read its *adopter* field.

D. Statement of soundness

We write $R \vdash t : T$ as a short-hand for $R; \varnothing; \mathsf{empty} \vdash t : T$. Let us furthermore write "h *and* R *agree*" when the heap h and the instrumented heap R agree (in the obvious sense).

[6]We omit the definition of the predicate $R_2 \vdash \ell$ *adopts* $\vec{\ell'} \dashv R'_2$. The text that follows is an informal paraphrase of it.

Definition VI.1 *The configuration h / t is well-typed, and we write $\vdash h$ / t, if and only if there exist a resource R and a type T such that h and R agree and $R \vdash t : T$ holds.* ◇

The well-typedness of configurations is an invariant ("subject reduction") and rules out stuck configurations ("progress"). Thus, well-typed programs do not go wrong:

Theorem VI.2 (Type soundness) *Let t be a well-typed source program: that is, assume void $\vdash t : T$, where void is the empty resource. Then, the configuration $\emptyset$ / t either eventually reduces to an answer or diverges.* ◇

A more detailed outline appears in an appendix (§C).

Theorem VI.2 offers an end-to-end guarantee. It does not state how this guarantee is obtained, and does not state that "the ownership policy is properly enforced". Nevertheless, this is a strong result: if the ownership policy could somehow be subverted, then, because the owner of an object can change its type, type soundness would be compromised as well. If desired, subject reduction (lemma A.11) offers a more detailed statement. It guarantees that, at every time, there exists a type derivation, which has tree structure and can be viewed as the "ownership hierarchy" of the program under execution.

VII. 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 a few of them and contrast them with Mezzo.

Ownership Types [6] and its descendants restrict aliasing. Every object is owned by at most one other object, and an "owner-as-dominator" principle is enforced: every path from a root to an object x must go through x's owner. Universe Types [7] impose a slightly different principle, "owner-as-modifier". 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 meant to support program verification, as it allows the owner to impose an object invariant. Permission systems [8]–[10] annotate pointers 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.

The systems mentioned so far are refinements (restrictions) of a traditional type discipline. Separation logic [2] departs from this approach and obeys a principle that we dub "owner-as-asserter". (In O'Hearn's words, "ownership is in the eye of the asserter" [11].) Objects are described by logical assertions. To assert is to own: if one knows that "x is a linked list", then one may read and write the cells that form this list, and nobody else may. Whereas the previously mentioned systems combine structural descriptions (i.e., types) with owner or permission annotations, separation logic assertions are at once structural descriptions and claims of ownership.

Mezzo obeys "owner-as-asserter": permissions describe and claim ownership. The key motivation for this decision is to

allow strong updates: an object's description can be changed by its owner. This enables gradual initialization, memory reuse, and typestate tracking. In the future, piggybacking logical assertions on permissions could facilitate program verification.

Mezzo differs from separation logic in that it distinguishes between immutable and mutable data, supports first-class functions and algebraic data types, and can be type-checked without producing proof obligations.

A strength of Ownership Types is their ability to describe a container that does not own its (mutable) elements. In Mezzo, this is expressed by assigning a weak type (namely, dynamic) to an element while it is in the container and recovering stronger information via a dynamic check (a take instruction) when it is taken out of the container. Although this approach may seem inelegant, we believe that it is imposed by our strong interpretation of "owner-as-asserter": since the container does not own its elements, it cannot record any information about them, not even their type. dynamic is the most precise information that one can hope for about an object that one does not own.

On the other hand, a situation that Mezzo can easily describe is the action of taking an element of type T out of a container; processing it, thereby changing its type to U; then inserting it into another container. This can be expressed using containers that own their elements, and does not require dynamic. Ownership or Universe Types cannot express uniqueness or ownership transfer.

The use of singleton types to keep track of equations, and the idea that pointers can be copied, whereas permissions are affine, are inspired by Alias Types [12]. Linear [13] and affine [14] type systems support strong updates and often view permissions (or "capabilities") as ordinary values, which hopefully the compiler can erase. By offering an explicit distinction between permissions and values, we guarantee that permissions are erased, and we are able to make the flow of permissions mostly implicit. Through algebraic data types and through the type constructor $T \mid P$, we retain the ability to tie a permission to a value, if desired.

Adoption & abandon are inspired by adoption & focus [15] and by nesting [9]. The common purpose of all three mechanisms is to let one permission govern a group of objects. Adoption [15] and nesting are irreversible and static, whereas Mezzo's adoption can be undone, which makes it much more flexible, but incurs a runtime cost and introduces potential runtime failures. Because nesting is forever, nesting facts [9], which witness an adopter-adoptee pair, are duplicable. Somewhat analogously, Mezzo's dynamic is duplicable, but cannot mention the identity of the adopter or the type of the adoptee, as neither is stable. The identity of the adopter can be ascertained at runtime, and determines the type of the adoptee.

Regions [5], [12], [13], [15] have been widely used as a technical device that allows a type to indirectly refer to a value or set of values. In Mezzo, types refer to values directly. This simplifies the meta-theory and the programmer's view.

We are considering extending Mezzo with support for shared-memory concurrency. Many abstractions (threads, locks, channels, tasks, etc.) can be axiomatized in a way that ensures that well-typed programs are data-race-free. Fractional permissions [9] could in principle be added as well. In Mezzo, a mutable block can be made immutable forever; fractional permissions would allow reversing this transition.

Gordon *et al.* [10] ensure data-race freedom in a simple extension of C#. They qualify 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. A salient feature is the absence of an alias analysis, which simplifies the system considerably. This comes at a cost in expressiveness: mutable global variables, as well as shared objects protected by locks, are disallowed.

From the author's previous work [5], we borrow some ideas, such as the notion of a monotonic separation algebra. The absence of regions and of an instrumented semantics represent significant technical simplifications. The notion of hidden state is currently absent in Mezzo; one way of introducing it would be to extend Mezzo with dynamically-allocated locks.

Acknowledgements

The author would like to extend his warmest thanks to Jonathan Protzenko, co-designer and implementor of Mezzo.

References

[1] F. Pottier and J. Protzenko, "Mezzo," Jan. 2013, http://gallium.inria.fr/~protzenk/mezzo-lang/.

[2] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *IEEE Symposium on Logic in Computer Science (LICS)*, 2002, pp. 55–74.

[3] F. Pottier and J. Protzenko, "Programming with permissions in Mezzo," Oct. 2012, unpublished. http://gallium.inria.fr/~fpottier/publis/pottier-protzenko-mezzo.pdf.

[4] F. Pottier, "A Coq proof of type soundness for Mezzo," Jan. 2013, http://gallium.inria.fr/~fpottier/mezzo/mezzo.tar.gz.

[5] ——, "Syntactic soundness proof of a type-and-capability system with hidden state," *Journal of Functional Programming*, vol. 23, no. 1, pp. 38–144, 2013.

[6] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998, pp. 48–64.

[7] W. Dietl and M. Peter, "Universes: Lightweight ownership for JML," *Journal of Object Technology*, vol. 4, no. 8, pp. 5–32, 2005.

[8] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 301–320.

[9] J. T. Boyland, "Semantics of fractional permissions with nesting," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 6, 2010.

[10] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and reference immutability for safe parallelism," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012, pp. 21–40.

[11] P. W. O'Hearn, "Resources, concurrency and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1–3, pp. 271–307, 2007.

[12] F. Smith, D. Walker, and G. Morrisett, "Alias types," in *European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 1782. Springer, 2000, pp. 366–381.

[13] A. Ahmed, M. Fluet, and G. Morrisett, "L^3: A linear language with locations," *Fundamenta Informaticæ*, vol. 77, no. 4, pp. 397–449, 2007.

[14] J. A. Tov and R. Pucella, "Practical affine types," in *ACM Symposium on Principles of Programming Languages (POPL)*, 2011, pp. 447–458.

[15] M. Fähndrich and R. DeLine, "Adoption and focus: practical linear types for imperative programming," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 13–24.

$$\frac{K \vdash v : \text{value}}{K \vdash v : \text{prog}} \qquad \frac{K(x) = \kappa}{K \vdash x : \kappa} \qquad \frac{K, x : \text{value} \vdash t : \text{prog}}{K \vdash \lambda x.t : \text{value}} \qquad \frac{K \vdash v : \text{value} \qquad K \vdash t : \text{prog}}{K \vdash v\ t : \text{prog}}$$

$$K \vdash \ell : \text{value} \qquad \frac{K \vdash \vec{v} : \text{seq value}}{K \vdash \text{new } i\ \{\vec{v}\} : \text{prog}}$$

$$\frac{K \vdash v : \text{value}}{K \vdash v.f : \text{prog}} \qquad \frac{K \vdash v_1 : \text{value} \qquad K \vdash v_2 : \text{value}}{K \vdash v_1.f \leftarrow v_2 : \text{prog}} \qquad \frac{K \vdash v : \text{value} \qquad K \vdash \vec{t} : \text{seq prog}}{K \vdash \text{switch}\ (v)\ \vec{t} : \text{prog}} \qquad \frac{K \vdash v : \text{value}}{K \vdash \text{tag of } v \leftarrow i : \text{prog}}$$

$$\frac{K \vdash v_1 : \text{value} \qquad K \vdash v_2 : \text{value}}{K \vdash \text{give } v_1 \text{ to } v_2 : \text{prog}} \qquad \frac{K \vdash v_1 : \text{value} \qquad K \vdash v_2 : \text{value}}{K \vdash \text{take } v_1 \text{ from } v_2 : \text{prog}} \qquad K \vdash \text{fail} : \text{prog} \qquad \frac{K \vdash T : \text{type} \qquad K \vdash U : \text{type}}{K \vdash T \to U : \text{type}}$$

$$\frac{K \vdash \vec{T} : \text{seq type} \qquad K \vdash U : \text{type}}{K \vdash m\ i\ \{\vec{T}\} \text{ adopts } U : \text{type}} \qquad K \vdash \text{dynamic} : \text{type} \qquad K \vdash \tau : \text{type} \qquad \frac{K \vdash v : \text{value}}{K \vdash =v : \text{type}} \qquad \frac{K \vdash T : \text{type} \qquad K \vdash P : \text{perm}}{K \vdash T \mid P : \text{type}}$$

$$K \vdash \bot : \text{type} \qquad \frac{K \vdash v : \text{value} \qquad K \vdash T : \text{type}}{K \vdash v \mathbin{@} T : \text{perm}} \qquad K \vdash \text{empty} : \text{perm} \qquad \frac{K \vdash P : \text{perm} \qquad K \vdash Q : \text{perm}}{K \vdash P * Q : \text{perm}} \qquad \frac{K, x : \text{value} \vdash P : \text{perm}}{K \vdash \exists x.P : \text{perm}}$$

$$K \vdash \epsilon : \text{seq}\ \kappa \qquad \frac{K \vdash t : \kappa \qquad K \vdash \vec{t} : \text{seq}\ \kappa}{K \vdash t; \vec{t} : \text{seq}\ \kappa}$$

Fig. 10. Kinding rules

APPENDIX

A. Well-kindedness

The well-kindedness judgement $K \vdash t : \kappa$ is inductively defined by the rules in figure 10. The first rule states that value is a subset of prog. The remaining rules are syntax-directed.

B. Derived rules and encoding of the simply-typed λ-calculus

None of the typing rules (§V) 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 (figure 5) only allows proving that x has type $=x$. Fortunately, an axiom rule, presented in figure 11, can be derived from SINGLETON, SUBRIGHT, and the permission subsumption axioms.

We have defined "let $x = u$ in t" as sugar for "$(\lambda x.t)\ u$". Using APPLICATION and FRAME, among others, it is possible to derive a typing rule for this construct (figure 11). The rule states that part of the available permissions (namely P) are used to prove that the term t has type T, while the rest (namely Q), together with the new hypothesis $x \mathbin{@} T$, are used to type-check the term u.

VALUE
$$\frac{K \vdash v : \text{value} \qquad K \vdash T : \text{type}}{\mathit{void}; K; v \mathbin{@} T \vdash v : T}$$

LETFRAME
$$\frac{K \vdash T : \text{type} \qquad K \vdash Q : \text{perm} \qquad \mathit{void}; K; P \vdash t : T \qquad \mathit{void}; K, x : \text{value}; Q * (x \mathbin{@} T) \vdash u : U}{\mathit{void}; K; P * Q \vdash \text{let } x = t \text{ in } u : U}$$

Fig. 11. Derived typing rules for variables and sequencing

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

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

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 $T ::= () \mid T \to T$. 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:

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

This encoding is type-preserving:

Lemma A.1 (Encoding) *If $E \vdash t : T$ holds in the simply-typed λ-calculus, then void; $[\![E]\!]$; $(\!|E|\!) \vdash [\![t]\!] : [\![T]\!]$ holds.* ◇

C. Proof of type soundness

We outline the main steps along the way that leads to the final statement of type soundness (theorem VI.2).

The following two lemmas are syntactic "sanity checks". We formulate them about the typing judgement; there are analogous statements about the other judgements.

The typing judgement is "reasonable" in the sense that it holds only of well-formed programs and (if seeded with a well-formed precondition) produces a well-formed type:

Lemma A.2 (Reason) *If $R; K; P \vdash t : T$ holds, then:*

1) *$K \vdash t$: prog holds;*
2) *$K \vdash P$: perm implies $K \vdash T$: type.* ◇

The typing judgement is preserved by every kind-preserving substitution. This holds at arbitrary kinds: value variables may be replaced with values, type variables may be replaced with types, and so on. When one replaces a variable of kind value with a value v, this value is not required to be well-typed. This is in contrast with the substitution lemma of (say) simply-typed λ-calculus.

Lemma A.3 (Substitution) *Well-typedness is preserved by the substitution of something of kind κ for a variable of kind κ.*

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

A technical detail was glossed over earlier (§VI-A). The function $\star$ was presented as a partial function of two resources to a resource: it is undefined when its arguments represent two incompatible views of the heap. In reality, $\star$ is a total function, which produces an "inconsistent" result when its arguments are incompatible. A new predicate, written "R *is consistent*", identifies the consistent resources.

The following three lemmas are established independently of one another. Again, we formulate them about the typing judgement; there are analogous statements about the other judgements.

The actions of an active principal cannot cause an inactive principal to become ill-typed. In other words, well-typedness is stable in the face of legal interference, as defined by the relation $\lhd$ (§VI-A).

Lemma A.4 (Stability) *Well-typedness is preserved under an evolution of the resources along the relation $\lhd$.*

$$\frac{R_1; K; P \vdash t : T \qquad R_1 \text{ is consistent} \qquad R_1 \lhd R_2}{R_2; K; P \vdash t : T} \quad \diamond$$

The system is affine: extra resources do not hurt.

Lemma A.5 (Affinity) *Well-typedness is preserved under the addition of unnecessary resources.*

$$\frac{R_1; K; P \vdash t : T \qquad R_1 \star R_2 \text{ is consistent}}{R_1 \star R_2; K; P \vdash t : T} \quad \diamond$$

The syntactic notion of duplicable types and permissions, as defined in figure 3, is sound with respect to the semantic notion of a duplicable resource. The statement uses the function $\widehat{\cdot}$, which maps a resource to its duplicable fragment (§VI-A). In this and the following statements, we write $R \Vdash v : T$ as an abbreviation for $R; \varnothing \Vdash v \mathbin{@} T$.

Lemma A.6 (Duplication) *If a canonical type derivation, at a duplicable type, is justified by some resource R, then it is also justified by the duplicable fragment of R.*

$$\frac{R \Vdash v : T \qquad R \text{ is consistent} \qquad T \text{ is duplicable}}{\widehat{R} \Vdash v : T} \quad \diamond$$

We now prove a classification lemma and a decomposition lemma for each type constructor. They extract information out of a canonical typing judgement for a closed value. By way of example, we present the classification and decomposition lemmas for functions; there are analogous lemmas for each of the other type constructors.

Lemma A.7 (Classification) *Among the closed values, only λ-abstractions admit a function type.*

$$\frac{R \Vdash v : T \to U}{\exists x t,\ v = \lambda x.t} \quad \diamond$$

Lemma A.8 (Decomposition) *If a closed function $\lambda x.t$ has type $T \to U$, then t has type U under the precondition $x \mathbin{@} T$.*

$$\frac{R \Vdash \lambda x.t : T \to U \qquad R \text{ is consistent}}{\widehat{R}; \varnothing, x : \mathsf{value}; x \mathbin{@} T \vdash t : U} \quad \diamond$$

This decomposition lemma is slightly stronger than one might expect in view of the typing rule FUNCTION (figure 5). Indeed, the lemma does not mention the fact that the function body could require a duplicable permission P. We are able to establish this strong statement because the permission P, if there is one, can be hidden by appeal to lemma A.6 and by application of the typing rule CUT.

We now come to two key lemmas about subsumption.

Permission subsumption, as axiomatized in figures 8 and 9, is sound with respect to the semantic notion of entailment that naturally arises out of the interpretation of permissions. All of the previous lemmas (except Stability, which is used only in the proof of Subject reduction) are used in this proof.

Lemma A.9 (soundness of subsumption) *The subsumption of permissions is sound:*

$$\frac{\varnothing \vdash P \leq Q \qquad R; \varnothing \Vdash P \qquad R \text{ is consistent}}{R; \varnothing \Vdash Q}$$

and so is the subsumption of types:

$$\frac{\varnothing \vdash T \leq U \qquad R \Vdash v : T \qquad R \text{ is consistent}}{R \Vdash v : U} \quad \diamond$$

This implies that an arbitrary derivation can be turned into a canonical one, i.e., one that does not use the subsumption rules SUBLEFT and SUBRIGHT, except possibly under λ:

Lemma A.10 (Canonicalization) *If a closed value v admits the type T, then there is a canonical derivation of this fact.*

$$\frac{R \vdash v : T \qquad R \text{ is consistent} \qquad \varnothing \vdash v : \mathsf{value}}{R \Vdash v : T} \quad \diamond$$

Thus equipped, we climb the two usual cliffs of the syntactic approach to type soundness: subject reduction and progress.

Lemma A.11 (Subject reduction) *The well-typedness of configurations is preserved by reduction.*

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

Lemma A.12 (Progress) *A well-typed configuration either is an answer or is able to take a reduction step.*

$$\frac{\vdash c}{c \text{ is not stuck}} \quad \diamond$$

Theorem VI.2 is a direct corollary of the last two lemmas.