Programming with permissions in Mezzo

François Pottier
INRIA
francois.pottier@inria.fr

Jonathan Protzenko
INRIA
jonathan.protzenko@ens-lyon.org

Abstract

We present Mezzo, a typed programming language of ML lineage. Mezzo is equipped with a novel static discipline of duplicable and affine permissions, which controls aliasing and ownership. This rules out certain mistakes, including representation exposure and data races, and enables new idioms, such as gradual initialization, memory re-use, and (type)state changes. Although the core static discipline disallows sharing a mutable data structure, Mezzo offers several ways of working around this restriction. One of them, a dynamic ownership control mechanism which we baptize “adoption and abandon”, is new.

1. Introduction

Programming with mutable, heap-allocated data structures is difficult. In many typed imperative programming languages, including Java, C#, and ML, the type system keeps track of the structure of objects, but not of how they are aliased. As a result, a programming mistake can cause undesired sharing, which in turn leads to breaches of abstraction, invariant violations, race conditions, and so on. Furthermore, the fact that sharing is uncontrolled implies that the type of an object must never change. This forbids certain idioms, such as delayed initialization, and prevents the type system from keeping track of the manner in which objects change state through method calls. In order to work around this limitation, programmers typically use C# and Java’s null pointer, or ML’s option type. This implies that a failure to follow an intended protocol is not detected at compile time, but leads to a runtime error. In short, there is a price to pay for the simplicity of traditional type systems: the bugs caused by undesired sharing, or by the failure to follow an object protocol, are not statically detected.

This paper presents the design of a new programming language, Mezzo, which attempts to address these issues. One motivating principle behind the design of Mezzo is that one should be able to express precise assertions about the current state of an object or data structure. The type system should keep track of state changes and forbid using an object in a manner that is inconsistent with its current state. An example is a socket that moves from state “ready”, to “connected”, then to “closed”. The “close” function, for instance, should be invoked only if the socket is currently in the state “connected”, and changes its state to “closed”. Another example is a collection, which must not be accessed while an iterator exists, but can be used again once iteration is over.

Although state and state change play an important role in many programs, no mainstream programming language builds these notions into its static discipline. External tools must be used, such as typestate checking tools [13, 5, 6] or tools for constructing proofs of programs, based for instance on separation logic [4, 20] or on the Spec# methodology [3]. Instead, we explore the possibility of reasoning about state within the type system. This has well-known potential benefits. A property that is expressed as a type is checked at compile time, and can thus go without a verification tool. Beyond that, a type system can serve as a strong foundation for performing proofs of programs.

Obviously, if two “principals” separately think that “the socket s is currently connected”, and if one of them decides to close this socket, then the other will be left with an incorrect belief about s. Thus, precise reasoning about state and state changes requires that information about a mutable object (or data structure) be recorded in at most “one place” in the type system. In Mezzo, this place is a permission. Permissions keep track not only of the structure of data, as does a traditional type system, but also of must-alias and must-not-alias (i.e. equality and disjointness) information. Like a separation logic assertion [28], a permission has an ownership reading: to have access to a description of a part of the heap is to own this part of the heap. Because “to describe is to own”, we need not explicitly annotate types with owners, as done in Ownership Types [10] or Universe Types [14].

We do not think of the “type” of an object and of its “state” as two distinct notions: a permission describes both at once. Whereas previous work on permissions [5] distinguishes between a fixed type structure and “permissions” that evolve with time, in Mezzo, both “type” and “state” can change over time. This yields greater expressiveness: for instance, gradual initialization and memory re-use become possible. This also yields greater simplicity and conciseness: for instance, when we write polymorphic code that manipulates a list, a single type variable a denotes not only “what” the list elements are (e.g., sockets) but also in what “state” they are and to what extent we “own” them.

The choices described above form our basic design premises. Mezzo can be viewed as an experiment, whose aim is to determine to what extent these choices are viable. Beyond these decisions, we strive to make the language as simple as possible. Mezzo is a high-level programming language: we equip it with first-class functions, algebraic data types, and require a garbage collector. We could have chosen classes and objects instead of (or in addition to) algebraic data types; this could be a topic for future research.

We equip Mezzo with a simple distinction between duplicable permissions (for immutable data) and exclusive permissions (for mutable data). Although more advanced varieties of permissions exist in the literature, including read-only views of mutable data and fractional permissions [7], we wish to evaluate how far one can go without these advanced notions; if desired, they could in principle be added to Mezzo.
By default, *Mezzo*’s permission discipline imposes a restrictive aliasing regime: the mutable part of the heap must form a forest. *Mezzo* offers several mechanisms for evading this restriction. One, adoption and abandon, is new. It allows arbitrary aliasing patterns within a region of the heap and achieves soundness via dynamic checks. We describe it in detail in §7. The second mechanism is Boyland’s nesting [7]. It can be viewed as a form of adoption and abandon that requires no runtime checks but is (for many purposes) less powerful. The last mechanism is locks in the style of concurrent separation logic [25, 18, 19, 8].

*Mezzo*’s static discipline has been formally defined and mechanically proved sound[1]. The formalization, which is available online [26], includes adoption and abandon, but does not (at present) cover nesting, locks, or concurrency. The statement of soundness guarantees that “well-typed programs do not go wrong”, except possibly when the dynamic check involved in the “abandon” operation fails. In a concurrent extension of *Mezzo*, it would in addition guarantee that “well-typed programs are data-race-free”.

A prototype type-checker has been implemented and is publicly available [27]. Several small libraries, totaling a few thousand lines of code, have been written, and are also available online [27]. They include immutable data structures (lists), mutable data structures (lists, doubly-linked lists, binary search trees, hash tables, resizable arrays, and FIFO queues, see §7), persistent data structures implemented via imperative means (suspensions, persistent arrays), and a few algorithms (memoization; graph search). At the time of this writing, an interpreter is available, and a simple compiler (which translates *Mezzo* down to untyped OCaml) is being developed.

The paper begins with a motivating example (§2), which cannot be type-checked in ML, and which serves to informally illustrate *Mezzo*’s permission discipline. Then, we define the syntax of types, permissions, and expressions (§3) and informally explain the ownership reading of permissions for immutable and mutable data (§4). We present the typing rules (§5) and introduce a few syntactic conventions that make the surface language more palatable (§6). We explain adoption and abandon, illustrate them with a second example (§7), and discuss nesting and locks more briefly (§8). Finally, we explain where *Mezzo* lies in the design space and compare it with some of the previous approaches found in the literature (§9).

## 2. Mezzo by example

Figure 1 presents code for the concatenation of two immutable lists. This example showcases several of *Mezzo*’s features, and allows us to explain the use of permissions. We review the code first (§2.1), then briefly explain how it is type-checked (§2.2 and §2.3).

### 2.1 Code

Our purpose is to write code that concatenates two immutable lists $xs$ and $ys$ to produce a new immutable list. The traditional, purely functional implementations of concatenation have linear space overhead, as they implicitly or explicitly allocate a reversed copy of $xs$. Our implementation, on the other hand, is written in destination-passing style, and has constant space overhead. Roughly speaking, the list $xs$ is traversed and copied on the fly. When the end of $xs$ is reached, the last cell of the copy is made to point to $ys$.

The append function (Figure 1, line 23) is where concatenation begins. If $xs$ is empty, then the concatenation of $xs$ and $ys$ is $ys$ (line 27). Otherwise (line 29), append allocates an unfinished, mutable cell $dst$ (line 30). This cell contains the first element of the final list, namely $xs$.head. It is in an intermediate state: it cannot be considered a valid list, since its tail field contains the unit value $()$. It is now up to appendAux to finish the work by constructing the concatenation of $xs$.tail and $ys$ and writing the address of that list into $dst$.tail. Once appendAux returns, $dst$ has become a well-formed list (this is indicated by the postcondition “$dst \not\in list$” on line 10) and is returned by append.

The function appendAux expects an unfinished, mutable cell $dst$ and two lists $xs$ and $ys$. Its purpose is to write the concatenation of $xs$ and $ys$ into $dst$.tail, at which point $dst$ can be considered a well-formed list. If $xs$ is nil (line 12), the tail field of $dst$ is made to point to $ys$. Then, $dst$, a mutable $MCons$ cell, is “frozen” by a tag update instruction and becomes an immutable $Cons$ cell. (This instruction compiles to a no-op.) If $xs$ is a $Cons$ cell (line 15), we allocate a new destination cell $dst’$, let $dst$.tail point to it, freeze $dst$, and repeat the process via a tail-recursive call.

This example illustrates several important aspects of *Mezzo*.

#### Expressiveness

In a traditional typed programming language, such as Java or OCaml, list concatenation in destination-passing style is possible, but its result must be a mutable list, because an immutable list cell cannot be gradually initialized.

#### State change

The call appendAux($dst$, $xs$, $ys$) changes the “type” of $dst$ from “unfinished, mutable list cell” to “well-formed, immutable list”. This type-changing update is sound because one must be the “unique owner” of the mutable cell $dst$ for this call to be permitted.

#### Ownership transfer

In fact, the call appendAux($dst$, $xs$, $ys$) also changes the “type” of $xs$ and $ys$ from “immutable list” to “un-

---

1 The formalization concerns a slightly lower-level language, Core Mezzo. In Core Mezzo, fields are numbered, whereas in *Mezzo* they are named and field names can be overloaded. At present, Core Mezzo is missing some of the features of *Mezzo*, including parameterized algebraic data types and mode constraints. We hope to add them in the future.

Figure 1. Tail-recursive concatenation of immutable lists
known”. Indeed, the postcondition of \texttt{appendAux} guarantees nothing about \texttt{x} and \texttt{ys}. In other words, the caller gives up the permission to use \texttt{x} and \texttt{ys} as lists, and in return gains the permission to use \texttt{dat} as a list. In other words, the ownership of the list elements is transferred from \texttt{x} and \texttt{ys} to \texttt{dat}. This is required for soundness. We do not know what the list elements are (they have abstract type \texttt{a}). They could be mutable objects, whose “unique ownership” property must not be violated\(^2\).

\subsection{2.2 Permissions}

Permissions do not exist at runtime: they are purely an artefact of the type system. An atomic permission \texttt{x \@ t} represents the right to use the program variable \texttt{x} at type \texttt{t}. Two permissions \texttt{P}_1 and \texttt{P}_2 can be combined to form a composite permission \texttt{P}_1 \ast \texttt{P}_2. The conjunction \texttt{\ast} is separating [28] at mutable memory locations and requires agreement at immutable locations (§4.1). The empty permission, a unit for conjunction, is written \texttt{Nil}. Most of the time, the manner in which permissions evolve and flow is implicit. It must be made explicit in a few places: in particular, every function type must include explicit pre- and postconditions.

Let us continue our discussion of the concatenation example (Figure 1). We explain in an informal manner how the function \texttt{append} is type-checked. This allows us to illustrate how permissions are used and how they evolve.

The typing rules appear in Figure 4; the permission subsumption rules appear in Figure 6. In the following, we refer to some of these rules, but defer their detailed explanation to §5.

The append function is defined at line 23. At the beginning of the function’s body, by the typing rule \texttt{FUNCTION}, permissions for the formal arguments are available. Thus, the current permission is:

\begin{verbatim}
 xs \@ list a \ast ys \@ list a
\end{verbatim}

This permission represents the right to use \texttt{x} and \texttt{ys} as lists of elements of type \texttt{a}.

This permission soon evolves, thanks to the \texttt{match} construct, which examines the tag carried by \texttt{x}. By the typing rule \texttt{MATCH}, as we learn that \texttt{x} is a \texttt{Nil} cell, we replace our permission about \texttt{x} with a more precise one, which incorporates the knowledge that the tag of \texttt{x} is \texttt{Nil}. At line 27, the current permission becomes:

\begin{verbatim}
 xs \@ Nil \ast ys \@ list a
\end{verbatim}

\texttt{xs \@ Nil} is a \textit{structural permission}: it asserts that \texttt{x} points to a memory block whose tag is \texttt{Nil} (and which has zero fields). Similarly, at line 29, the current permission becomes:

\begin{verbatim}
 xs \@ Cons \{ head : a; tail : list a \} \ast ys \@ list a
\end{verbatim}

The structural permission for \texttt{x} asserts that \texttt{x} points to a memory block that carries the tag \texttt{Cons} and has a head field of type \texttt{a} and a \texttt{tail} field of type \texttt{list a}.

At this stage, the type-checker performs an implicit operation. It applies the permission subsumption rule \texttt{DECOMPOSEBLOCK}. This causes fresh names \texttt{hd} and \texttt{tl} to be introduced for the head and \texttt{tail} fields of this structural permission. This yields the following conjunction:

\begin{verbatim}
 xs \@ Cons \{ head : (=hd); tail : (=tl) \} \ast hd \@ a \ast tl \@ list a \ast ys \@ list a
\end{verbatim}

\footnote{We later note (§4.1) that if at a call site the variable \texttt{a} is instantiated with a duplicable type, say \texttt{int}, then the permissions \texttt{xs \@ list int} and \texttt{ys \@ list int} are considered duplicable, so they can in fact be duplicated prior to the call \texttt{appendAux (dat, xs, ys)}, hence are not lost.}

This is our first encounter of a singleton type, which we write \texttt{=hd}. A permission of the form \texttt{x \@ y} asserts that the variables \texttt{x} and \texttt{y} denote the same value. In particular, if they denote memory locations, this means that \texttt{x} and \texttt{y} point to the same object: this is a \textit{must-alias} constraint. We write \texttt{x \@ y} for \texttt{x \@ \texttt{=} y}. Similarly, in the structural permission above, the fact that the head field has type \texttt{=hd} means that the value of this field is \texttt{hd}. We write \texttt{head \=} \texttt{hd} for \texttt{hd} (\texttt{:=hd}).

By the typing rules \texttt{NEW} and \texttt{LET}, when the cell \texttt{dat} is allocated (line 30), a permission for \texttt{dat} appears, namely:

\begin{verbatim}
 dst \@ MCons \{ head : hd; tail : () \}
\end{verbatim}

We now see how singleton types help reason about sharing. At this point, we have three permissions that mention \texttt{hd}. We know that \texttt{hd} is stored in the head field of \texttt{xs}; we know that \texttt{hd} is stored in the head field of \texttt{dat}; and we have a permission to use \texttt{hd} at type \texttt{a}.

We do not need a borrowing convention [24] in order to fix which of \texttt{xs} or \texttt{dst} owns \texttt{hd}. Instead, the system knows that the object \texttt{hd} is accessible via two paths, namely \texttt{xs} and \texttt{dat} and \texttt{hd}, and can be used under either name. This use of singleton types is taken from Alias Types [29].

By the typing rules \texttt{READ} and \texttt{APPLICATION}, in order to call \texttt{appendAux (dat, xs.tail, ys)} (line 32), we need the following conjunction of permissions. It is the precondition of \texttt{appendAux}, suitably instantiated:

\begin{verbatim}
 dst \@ MCons \{ head : a; tail : () \} \ast tl \@ list a \ast ys \@ list a
\end{verbatim}

Are we able to satisfy this requirement? The answer is positive. The subsumption rules \texttt{EXISTSI NTR O} and \texttt{DECOMPOSEBLOCK} allow combining the permissions \texttt{MCons \{ head \=} hd; tail : () \} and \texttt{hd \@ a} (both of which are present) to obtain the first conjunct above. The second and third conjuncts above are present already.

By \texttt{APPLICATION}, the precondition of \texttt{appendAux} is consumed (taken away from the caller). After the call, the postcondition of \texttt{appendAux} is added to the current permission, which is then:

\begin{verbatim}
 xs \@ Cons \{ head : hd; tail : tl \} \ast dst \@ list a
\end{verbatim}

The conjunct that concerns \texttt{x} is of no use, and is in fact silently discarded when we reach the end of the Cons branch within \texttt{append}.

The conjunct that concerns \texttt{dst} is used to check that this branch satisfies \texttt{append}’s advertised return type, namely \texttt{list a}. Similarly, in the Nil branch, the permission \texttt{ys \@ list a} shows that a value of appropriate type is returned. In conclusion, \texttt{append} is well-typed.

\subsection{2.3 To loop or to tail call?}

In-place concatenation (that is, melding) of mutable lists can also be implemented by a tail-recursive function. The pattern is analogous to that of Figure 1, but the code is simpler, because the first list is not copied, and “freezing” is not required.

These algorithms are traditionally viewed as iterative and implemented using a \texttt{while} loop. Berdine et al.’s iterative formulation of mutable list melding [4], which is proved correct in separation logic, has a complex loop invariant, involving two “list segments”, and requires an inductive proof that the concatenation of two list segments is a list segment. In contrast, in the tail-recursive approach, the “loop invariant” is the type of the recursive function (e.g., \texttt{appendAux} in Figure 1). This type is reasonably natural and does not involve list segments.

How do we get away without list segments and without inductive reasoning? The trick is that, even though \texttt{appendAux} is tail-recursive, which means that no code is executed after the call by \texttt{appendAux} to itself, a \textit{reasoning step} still takes place after the call. Immediately before the call, the current permission can be written as follows:

\begin{verbatim}
 xs \@ Cons \{ head : hd; tail : tl \} \ast
\end{verbatim}
The call “appendAux (dst’, xs.tail, ys)” consumes the last three permissions and produces instead dst’ @ list a. The first two permissions are “framed out”, i.e., implicitly preserved. After the call, we have:

\[
\begin{aligned}
x s @ \text{Cons} \{ \text{head} = \text{hd}; \text{tail} = \text{tl} \} \ast \\
\text{dst} @ \text{Cons} \{ \text{head} = a; \text{tail} = \text{dst}' \} \ast \\
\text{ys} @ \text{list} \ a 
\end{aligned}
\]

The data constructor \( \text{MCons} \{ \text{head} = a; \text{tail} : () \} \ast \text{tl} @ \text{list} \ a \ast \text{ys} @ \text{list} \ a \)

Dropping the first permission and combining the last two yields:

\[
\begin{aligned}
\text{dst} @ \text{Cons} \{ \text{head} = a; \text{tail} : \text{list} \ a \} \\
\end{aligned}
\]

which can be folded back to \( \text{dst} @ \text{list} \ a \), so appendAux satisfies its postcondition. The framing out of a permission during the recursive call, as well as the folding step that takes place after the call, are the key technical mechanisms that allow us to avoid the need for list segments and inductive reasoning. In short, the code is tail-recursive, but the manner in which one reasons about it is recursive.

Minamide [22] proposes a notion of “data structure with a hole,” or in other words, a segment, and applies it to the problem of concatenating immutable lists. Walker and Morrisett [34] offer a tail-recursive version of mutable list concatenation. Their code is formulated in a low-level typed intermediate language, as opposed to a surface language. The manner in which they avoid reasoning about list segments is analogous to ours. There, because the code is formulated in continuation-passing style, the reasoning step that takes place after the recursive call amounts to composing the current continuation with a coercion. Maeda et al. [21] study a slightly different approach, also in the setting of a typed intermediate language, where separating implication offers a way of defining list segments. Our approach could be adapted to an iterative setting by adopting a new proof rule for while segments. Our approach could be adapted to an iterative setting by

In Figure 2, we use the meta-variables \( T \) and \( X \) to stand for types and variables of arbitrary kind; we use \( t \) and \( P \) to suggest that a type has kind type and perm, respectively; we use \( a \) and \( x \) to suggest that a variable has kind type and term, respectively. The full definition of the kind system appears in Appendix A.2.2.

The \textbf{structural} type \( A \{ \vec{f} : t \} \) describes a block in the heap whose tag is currently \( A \) and whose fields \( \vec{f} \) currently have the types \( \vec{t} \). An example, taken from §2, is \( \text{MCons} \{ \text{head} = a; \text{tail} : () \} \ast \text{tl} @ \text{list} \ a \ast \text{ys} @ \text{list} \ a \).

The data constructor \( A \) must reference a previously defined algebraic data type, and the fields \( \vec{f} \) must match the definition of \( A \). The types \( \vec{t} \), however, need not match the types that appear in the definition of \( A \). For instance, in the definition of \( \text{MCons} \), the type of the tail field is \text{mlist} \ a, not \( () \). This implies that the above structural type cannot be folded to \( \text{mlist} \ a \); the tail field must be updated first. A structural type may include a clause of the form \( \text{adapts} \ t \), whose meaning is explained later on (§7). If omitted, adapts \( \perp \) is the default.

An example of a type application \( T \vec{T} \) is list int. We sometimes refer to this as a \textbf{nominal} type, as opposed to a structural type.

The universal and existential types are in the style of System F. A (base) kind annotation is mandatory; if omitted, type is the default. The bottom type \( \perp \) and the top type unknown can be defined as \( \forall a.a \) and \( \exists a.a \), respectively.
The conjunction of a type and a permission is written \((t \mid P)\). Because permissions do not exist at runtime, a value of this type is represented at runtime as a value of type \(t\). Such a conjunction is typically used to express function pre- and postconditions. The type \((t \mid P)\) is abbreviated as \(\langle t\rangle\).

Algebraic data types are defined using the keyword data. These definitions are analogous to Haskell’s and OCaml’s. Each branch is explicitly named by a data constructor and carries a number of named fields. If the definition is prefixed by the keyword mutable, then the tag and all fields are considered mutable, and can be modified via tag update and field update instructions; otherwise, they are considered immutable. Examples appear at the top of Figure 1. Like a structural type, an algebraic data type definition may include an adopts clause; if omitted, adopts \(\perp\) is the default.

3.2 Expressions

The syntax of expressions (Figure 3) forms a fairly standard \(\lambda\)-calculus with tuples and algebraic data structures.

A function definition must be explicitly annotated with the function’s type parameters, argument type, and return type. One reason for this is that the argument and return type serve as pre- and postconditions and in general cannot be inferred. Furthermore, we have System \(F\)-style polymorphism. Explicit type abstractions are built into function definitions. Type applications must in principle be explicit as well. The current prototype allows omitting them and performs a limited form of local type inference, which is outside the scope of this paper.

4. Ownership, modes, and extent

We wrote earlier (§1) that “to have a permission for \(x\) can be understood informally as “to own \(x\)”. Roughly speaking, this is true, but we must be more precise, for two reasons. First, we wish to distinguish between mutable data, on which we impose a “unique owner” policy, and immutable data, for which there is no such restriction. For this reason, types and permissions come in several flavors, which we refer to as modes (§4.1). Second, in a permission of the form \(x \subseteq t\), the type \(t\) describes the extent to which we own \(x\). If \(x\) is a list cell, do we own just this cell? The entire spine? The spine and the elements? The answer is given by the type \(t\). For instance (§4.2), \(x \subseteq \text{Cons}\{\text{head} = \text{hd}; \text{tail} = \text{tl}\}\) represents the ownership of just the cell \(x\), because the singleton types \(=\text{hd}\) and \(=\text{tl}\) denote the ownership of an empty heap fragment. On the other hand, \(x \subseteq \text{Cons}\{\text{head} : a; \text{tail} : \text{list}\ a\}\) gives access to the entire list spine. (Because list is an immutable algebraic data type, this is read-only, shared access.) It further gives access to all of the list elements, insofar as the type \(a\) allows this access. In this example, \(a\) is a variable: one must wait until \(a\) is instantiated to determine what the elements are and to what extent we own them.

4.1 Modes

A subset of the permissions are considered duplicable, which means that they can be implicitly copied (DUPLICATE, Figure 6). Copying a permission for an object \(x\) means that \(x\) may be shared: it may be used via different pointers, or by different threads simultaneously. Thus, a duplicable permission does not represent unique ownership; instead, it denotes shared knowledge. Because the system does not control with whom this knowledge is shared, this knowledge must never be invalidated, lest some principals be left with an outdated version of the permission. Therefore, a duplicable permission denotes shared, permanent knowledge. The permissions that describe read-only, immutable data are duplicable: for instance, \(x \subseteq \text{Cons}\{\text{head} = \text{hd}; \text{tail} = \text{tl}\}\) and \(x \subseteq \text{list}\ \text{int}\) are duplicable.

A subset of the permissions are considered exclusive. An exclusive permission for an object \(x\) represents the “unique owner-ship” of \(x\). In other words, such a permission grants read-write access to the memory block at address \(x\) and guarantees that no-one else has access to this block. The permissions that describe mutable memory blocks are exclusive: for instance, \(x \subseteq \text{MCons}\{\text{head} = \text{hd}; \text{tail} = \text{tl}\}\) is exclusive. An exclusive permission is analogous to a “unique” permission in other systems [5] and to a separation logic assertion [28].

Predicates of the form “\(t\) is duplicable” and “\(t\) is exclusive” are part of a more general form of predicates which we call facts. We are able to compute the fact for any given type, as well as an optimal fact for any given data type, such as “a list is duplicable as long as its elements are duplicable”. The details are provided in §B.

No permission is duplicable and exclusive. Some permissions are neither duplicable nor exclusive, “\(x \subseteq \text{list}\ (\text{ref} \ \text{int})\”, which describes an immutable list of references to integers, is such a permission. It must not be duplicated: this would violate the “unique owner” property of the list elements. It is not exclusive: the list cell at \(x\) is an immutable object, and this permission does not guarantee exclusive access to this cell. Another example is “\(x \subseteq a\)”. Because \(a\) is a type variable, one cannot assume that this permission is duplicable (or exclusive)\(^3\).

Every permission is affine. One can implicitly drop a permission that one does not need.

The language is designed so that the type-checker (and the programmer!) can always tell what mode \(a\) permission \(P\) satisfies: duplicable, exclusive, or neither (hence, affine). Modes form a lattice, whose top element is “affine”, and where “duplicable” and “exclusive” are incomparable; a “bottom” mode is added, which only an inconsistent permission can satisfy. Because algebraic data types are recursively defined, their mode analysis requires a fixed point computation, whose details are given in §B.

If \(t\) and \(u\) are exclusive types, then the conjunction \(x \subseteq t \ast y \subseteq u\) implies that \(x\) and \(y\) are distinct addresses. In other words, conjunction of exclusive permissions is separating. On the other hand, if \(t\) and/or \(u\) are duplicable, \(x\) and \(y\) may be aliases. Conjunction is not in general separating. Conjunction of duplicable permissions requires agreement between the two conjuncts. The reader is referred to the draft paper that accompanies the type soundness proof [26] for a formal definition of the semantics of conjunction.

4.2 Extent

Every type \(t\) has an ownership reading: that is, the permission \(x \subseteq t\) represents certain access rights about \(x\). However, the extent of these rights (or, in separation logic terminology, their footprint) depends on the type \(t\).

A singleton type \(a\), for instance, has empty extent. Indeed, the permission \(x \subseteq a\) indicates that \(x\) is a pointer, let alone dereferencing it.

A structural type such as \(\text{Cons}\{\text{head} = \text{hd}; \text{tail} = \text{tl}\}\) has an extent of one memory block. The permission \(x \subseteq \text{Cons}\{\text{head} = \text{hd}; \text{tail} = \text{tl}\}\) gives us (read-only, shared) access to the block at address \(x\), and guarantees that its head and tail fields contain the values \(\text{hd}\) and \(\text{tl}\), respectively, but (as per the semantics of singleton types) guarantees nothing about \(\text{hd}\) and \(\text{tl}\).

What is the extent of a “deep” composite type, such as the structural type \(\text{Cons}\{\text{head} : a; \text{tail} : \text{list}\ a\}\) or the nominal type \(\text{list}\ a\)? What does it mean to own a list? In order to answer these questions, one must understand how a composite permission is decomposed into a conjunction of more elementary permissions.

A structural permission, such as \(x \subseteq \text{Cons}\{\text{head} : a; \text{tail} : \text{list}\ a\}\), can be decomposed by introducing a fresh name for each of

\(^3\) Mode allows the programmer to explicitly assume that a type variable \(a\) is duplicable, or exclusive. This mechanism is not treated in this paper.
the values stored in the fields. (See DecomposeBlock in Figure 6.) The result is a more verbose, but logically equivalent, permission:
\[
\exists h, l. (\text{xs} @\text{Cons} \{\text{head} = h; \text{tail} = t\}) \leftrightarrow h \& \alpha \& t \& \alpha \& \text{list}\ a
\]
The meaning and extent of the original structural permission is now clearer: it grants access to the cell at \text{xs} and access to the first list element (to the extent dictated by the type \alpha) and access to the rest of the list.

The meaning of a nominal permission, such as \text{xs} @ list \alpha, is just the disjunction of the meanings of its unfoldings, namely \text{xs} @ Nil and \text{xs} @ Cons \{\text{head} : \alpha; \text{tail} : \alpha\}.

If \alpha is (instantiated with) an exclusive type, then we find that \text{xs} @ list \alpha implies that the list elements are pairwise distinct, and grants read-only access to the list spine and exclusive access to the list elements.

5. Type-checking

5.1 The typing judgment

The typing judgment takes the form \(K; P \vdash e : t\). It is inductively defined in Figures 4 and 5. The kind environment \(K\) maps variables to kinds. This judgment means that, by consuming the permission \(P\), the expression \(e\) produces a value of type \(t\). It is analogous to a Hoare logic or separation logic triple, where \(P\) is the precondition and \(t\) is the postcondition.

The typing rules require many sub-expressions to be variables. For instance, the rule \text{Read} cannot handle a field access expression of the form \(e.f\): instead, it requires \(x.f\). This requirement is met by first performing a monadic transformation, which introduces extra let constructs. Furthermore, the pattern matching rules (Figure 5) cannot handle deep patterns: they require shallow patterns. Again, this requirement is met by introducing extra let constructs. We omit the details of these transformations. For brevity, we omit the side conditions that concern the freshness of variables and the well-kindness of the user-provided type annotations.

\text{Var} is the axiom rule. It is worth noting that, in conjunction with the subsumption rule \text{EqualityRefl} (Figure 6), it allows proving that \(x\) has type \(\equiv x\), even in the absence of any hypothesis about \(x\).

\text{Let} corresponds to the sequence rule of separation logic.

\text{Function} states that a duplicable permission \(P\) that exists at the function definition site is also available within the function body. Requiring \(P\) to be duplicable allows us to consider every function type duplicable. Thus, a function can be shared without restriction and can be invoked as many times as desired, provided of course that one is able to satisfy its precondition. If one wishes to write a function that captures a non-duplicable permission \(P\), and can be invoked at most once, this is still possible. Indeed, the type \(t_1 \rightarrow t_2\) of "one-shot" functions can be defined as:
\[
\exists (p : \text{perm}) \cdot (t_1 \rightarrow t_2) \rightarrow p
\]
This is a conjunction of a function whose precondition is \(p\) and of one copy of \(p\). Because \(p\) is abstract, it is considered affine. Hence, at most one call is possible, after which \(p\) is consumed and the function becomes unusable.

\text{Application} corresponds to the rule for procedure calls in separation logic. The caller gives up the permission \(x_2 @ t_2\), which is consumed, so to speak, and in return gains a permission for the result of the function call, at type \(t_1\). In other words, because types have an ownership reading, a function type \(t_1 \rightarrow t_2\) describes not only the shape of the function’s arguments and results, but also the side effects that the function may perform, as well as the transfers of ownership that occur from the caller to the callee and back.

\text{New} uses a structural type to describe the newly-allocated memory block in an exact manner. \text{Tuple} is analogous.

\text{Read} requires a structural permission \(x @ A \{F[f : t]\}\), which guarantees that \(x\) points to a memory block that contains a field named \(f\), and allows us to dereference \(x.f\). (We overload field names: there could exist multiple data constructors that have a field named \(f\). There can be at most one permission of this form, though, which allows disambiguation to take place.) This permission indicates that the value stored in the field \(f\) has type \(t\). \text{Read} concludes that the field access expression \(x.f\) has type \(t\), and that the structural permission \(x @ A \{F[f : t]\}\) is preserved. There is a catch: because the type \(t\) occurs twice in this postcondition, clearly we must require \(t\) to be duplicable, or the rule would be unsound. Fortunately, this is not a problem: by using DecomposeBlock (Figure 6; also explained earlier, see §2.2 and §4.2), it is possible to arrange for \(t\) to be a singleton type, which is duplicable.

Like \text{Read}, \text{Write} requires a structural permission, of the form \(x_1 @ A \{F[f : t]\}\). It checks that this permission is exclusive, i.e., the data constructor \(A\) is associated with a mutable algebraic data type. This ensures that we have write access. In fact, since we have exclusive access to \(x_1\), a strong (type-changing) update is sound. The structural permission is changed to \(x_1 @ A \{F[f = t_2]\}\), where \(t_2\) is the type of \(x_2\). Without loss of generality, one may take \(t_2\) to be the singleton type \(\text{=t}_2\). This allows the type-checker to record that \(x_1.f\) and \(x_2\) are now aliases. If desired, the permissions \(x_1 @ A \{F[f = x_2]\}\) and \(x_2 @ t_2\) can later be combined by DecomposeBlock to yield \(x_1 @ A \{F[f = t_2]\}\).

Because DecomposeBlock, read from right to left, involves a loss of information, it is typically applied by the type-checker only "on demand", i.e., to satisfy a function postcondition or a type annotation.

\text{Match} is used to type-check a case analysis construct. Each branch is type-checked independently. We currently do not check that the case analysis is exhaustive, but are planning to add this feature in the future. The premise relies on a judgment of the form \(K; P \vdash \text{let} p = x \in e : t\). This is not a new judgment; it is an ordinary typing judgement, but, for clarity, the typing rules that have a conclusion of this form are isolated in Figure 5. Although these rules may appear somewhat daunting, they are in fact quite straightforward. \text{LetTuple} checks that \(x\) is a tuple, i.e., we have a permission of the form \(x @ (t_1, \ldots, t_n)\). If that is the case, then matching \(x\) against the tuple pattern \((x_1, \ldots, x_n)\) is permitted, and gives rise to a conjunction of permissions of the form \(x_1 @ t_1\). Because the permission for \(x\) is not lost, the types \(t_i\) are duplicated, so they are required to be duplicable. Again, this requirement causes no loss of generality, since one can arrange to introduce singleton types before the choice. \text{LetDataMatch} is analogous to \text{LetTuple}, but concerns a (mutable or immutable) memory block. \text{LetDataMismatch} concerns the situation where the pattern, which mentions the data constructor \(B\), will clearly not match \(x\), which is statically known to have the tag \(A\). In that case, the branch is dead code, and is considered well-typed. \text{LetDataUnfold} refines a nominal permission, such as \(x @ \text{list} \alpha\), by replacing it with a structural one, such as \(x @ \text{Cons} \{\text{head} : \alpha; \text{tail} : \text{list} \alpha\}\), obtained by unfolding the algebraic data type and specializing it with respect to the data constructor that appears in the pattern. We omit the exact definition of unfolding.

\text{WriteTag} type-checks a tag update instruction, which modifies the tag carried by a memory block. Like \text{Write}, it requires an exclusive permission for this block. It further requires the new tag \(B\) to carry the same number of fields as the previous tag \(A\). (Thus, the block does not have to be enlarged or shrunk.) The structural permission is updated in a straightforward way. The types \(t\) of the fields do not change. The names of the fields change

\footnote{We write \(F[f : t]\) for a sequence of field/type pairs within which the pair \(f : t\) occurs. The adopts clause, if there is one, is irrelevant.}
from \( \vec{f} \) to \( \vec{f}' \), where the sequences of fields are ordered in the same way as in the (user-provided) definitions of A and B. It is worth noting that A and B need not belong to the same algebraic data type: thus, a memory block can be re-used for a completely new purpose. Furthermore, the tag B may be associated with an immutable algebraic data type: in that case, the block is frozen, that is, becomes forever immutable. This feature is exploited in the concatenation of immutable lists (Figure 1, line 19).

\textbf{Give} and \textbf{Take} are explained later on (§7).

\textbf{Sub} is analogous to Hoare’s rule of consequence. It relies on permission subsumption, \( P_1 \leq P_2 \), defined in Figure 6 and discussed further on (§5.2), and on subtyping, \( t_1 \leq t_2 \), defined as \( x \oplus t_1 \leq x \oplus t_2 \) for a fresh \( x \).

\textbf{Frame} is analogous to the frame rule of separation logic.

5.2 The permission subsumption judgment

The rules that define the subsumption judgment appear in Figure 6. We comment a subset of them. Since \( x = y \) is sugar for \( x \oplus y \), the rule \texttt{EQUALITYREFLEXIVE} can be understood as a claim that \( x \) inhabits the singleton type \( \equiv x \). \texttt{EQUALSFOREQUALS} shows how equations are exploited: if \( y_1 \) and \( y_2 \) are known to be equal, then they are interchangeable. (We write \( \equiv \) for subsumption in both directions.) \texttt{DUPLICATE} states that a permission that is syntactically considered duplicable can in fact be duplicated. \texttt{MIXSTAR} introduces and eliminates \( (t \mid P) \). \texttt{WEaken} states that every permission is affine. \texttt{EXISTSIINTRO} introduces an existential permission; \texttt{EXISTSIATOMIC} converts between an existential permission and an existential type. When read from left to right, \texttt{DECOMPOSEBLOCK}, which was discussed earlier (§2.2, §4.2), introduces a fresh name \( x \) for the value stored in \( y \). When read from right to left, it forgets such a name. (In that case, it is typically used in conjunction with \texttt{EXISTSIINTRO}.) \texttt{FOLD} folds an algebraic data type definition, turning
a structural type into a nominal type. Unfolding is normally performed by case analysis (see \textsc{letDataUnfold} in Figure 4), but in the special case where an algebraic data type has only one branch (i.e., it is a record type), it can be implicitly unfolded by \textsc{unfold}. \textsc{dynamicAppears} is explained later on (§7).

6. Surface syntax

The internal syntax, which we have been using so far, can be fairly verbose. To remedy this, we introduce two syntactic conventions, which rely on the \textit{name introduction} construct and on the consumes keyword (Figure 2). Two transformations eliminate these constructs, so as to obtain a type expressed in the internal syntax. This section contains an informal discussion that gives the intuition for these transformations. The two transformations are formalized and discussed in A.4.

6.1 The name introduction form

The construct \( x : t \) allows introducing a name \( x \) for a component of type \( t \). This allows writing "dependent function types", such as \( (x_1 : t_1) \to (x_2 : t_2) \), where by convention \( x_1 \) is bound within \( t_1 \) and \( x_2 \) is bound within \( t_2 \). This is desugared by quantifying \( x_1 \) universally above the arrow and quantifying \( x_2 \) existentially in the right-hand side of the arrow.

As an example, consider the type of \( :: \): the function that writes a reference. This function expects a pair of a reference \( x \) whose content has type \( a \) and of a value of type \( b \), which it stores into \( x \). At the end, \( x \) has become a reference whose content has type \( b \). The variable \( x \) must be mentioned in the pre- and postcondition. In the internal syntax, the type of \( :: \) is:

\[
\forall a, b. \forall (x : \text{term}) \quad ((x : a) \times (x : \text{ref } a), b) \to (x : \text{ref } b)
\]

Thanks to the name introduction form, instead of planning ahead and quantifying \( x \) in front of the function type, one names the first argument "\( x \)" on the fly. Thus, in the surface syntax, one writes:

\[
\forall a, b. \text{c} (x : \text{ref } a, \text{c} a b) \to (x : \text{ref } b)
\]

This is not significantly shorter, because of the \textit{consumer} keyword, which must be used in the surface syntax, as explained below. In actual use, though, the comfort afforded by this feature is critical.

6.2 The consumes annotation

Often, a permission is required \textit{and returned} by a function, in which case it is unpleasant to have to write this permission twice, in the precondition and postcondition. Drawing inspiration from Sing# [15], we adopt the convention that, in the surface syntax, by default, the permission for the argument is required and returned, i.e., it is not consumed.

For instance, the type of the list \textit{length} function, which in the internal syntax is:

\[
\forall a. \forall (x : \text{term}) (\text{length } x | x \times \text{list } a) \to \text{(int } | x \times \text{list } a)
\]

can in the surface syntax be written in a much more pleasant form:

\[
\forall a. \text{list } a \to \text{int}
\]

The type list \( a \) is mentioned once, instead of twice, and as a side effect, the need to name the argument \( x \) vanishes.

When a permission is consumed, though, we need a way of indicating this. This is the purpose of the consumes keyword. When a component is marked with this keyword, the permission for this component is required and not returned. This keyword only makes sense in the left-hand side of an arrow.

Because internal syntax and surface syntax interpret the function type differently, a translation is required, regardless of whether consumes is used. Consider a function type of the form \( t \to u \), where \( t \) does not contain any name introduction forms. Let \( t_1 \) stand for \( [\tau/\text{consumes } \tau] t \), i.e., a copy of \( t \) where the consumes keyword is erased. Let \( t_2 \) stand for \( [\tau/\text{consumes } \tau] t \), i.e., a copy of \( t \) where every component marked with this keyword is replaced with \( \top^\tau \). Then, the translation of this function type is \( (x : t_1) \to (u | \top) \).

\footnote{Here, we write \( \top \) for unknown or empty, depending on whether the consumes keyword is applied to a type or a permission.}
val retrieve : [a] bag a -> option a
val insert : [a] (consumes a, bag a) -> ()
val create : [a] () -> bag a

abstract bag a

In the internal syntax, functions take the form

\[ \text{fun } x \text{ : type } \forall (x : \text{term}) \Rightarrow (x \circ (a, \text{bag } a)) \rightarrow (x \circ (\text{unknown}, \text{bag } a)) \]

6.3 Function definitions

In the internal syntax, functions take the form fun \((x : t_1) : t_2 = e\), where one variable, namely \(x\), is bound in \(e\). In the surface syntax, instead, functions take the form fun \(t_1 : t_2 = e\). The argument type \(t_1\) is interpreted as a pattern, and the names that it introduces are considered bound in \(e\). An example is fun \((x : \text{int}, y : \text{int}) : \text{int} = x + y\), where \((x : \text{int}, y : \text{int})\) is the type of the argument, \(\text{int}\) is the type of the result, and \(x\) and \(y\) are bound in the function body, which is \(x + y\).

7. Adoption and abandon

The permission discipline that we have presented so far has limited expressive power. It can describe immutable data structures with arbitrary sharing and tree-shaped mutable data structures. However, because mutable memory blocks are controlled by exclusive permissions, it cannot describe mutable data structures with sharing.

7.1 Overview

In order to illustrate this problem, let us imagine how one could implement a “bag” abstraction. A bag is a mutable container, which supports two operations: inserting a new element and retrieving an arbitrary element.

We would like our implementation to offer the interface in Figure 7. There, bag is presented as an abstract type. Because it is not explicitly declared duplicable, it is regarded as affine. Hence, a bag “has a unique owner”, i.e., is governed by a nonduplicable permission. The function create creates a new bag, whose ownership is transferred to the caller. The type of insert indicates that insert(x, b) requires the permissions "x ∈ t" and "b ∈ bag t", for some type t, and returns only the latter. Thus, the caller gives up the ownership of x, which is “transferred to the bag”. Conversely, the call “let o = retrieve b in ...” produces the permission “o ∈ option a”, which means that the ownership of the retrieved element (if there is one) is “transferred from the bag to the caller”.

To implement bags, we choose a simple data structure, namely a mutable singly-linked list. One inserts elements at the tail and extracts elements at the head, so this is a FIFO implementation. One distinguished object b, “the bag”, has pointers to the head and tail of the list, so as to allow constant-time insertion and extraction. (We use “object” as a synonym for “memory block”.)

This data structure is not tree-shaped: the last cell in the list is accessible via two distinct paths. In order to type-check this code, we must allow the ownership hierarchy and the structure of the heap to differ. More specifically, we would like to view the list cells as collectively owned by the bag b. That is, we wish to keep track of just one exclusive permission for the group formed by the list cells, as opposed to one permission per cell.

We use the name b as a name for this group. When a cell c joins the group, we say that b adopts c, and when c leaves the group, we say that b abandons c. In other words, the bag b is an adopter, and the list cells c are its adoptees. In terms of ownership, adopter and adoptees form a unit: the exclusive permission that controls b also represents the ownership of the group, and is required by the adoption and abandon operations.

Adoption requires and consumes an exclusive permission for the cell c that is about to be adopted: the ownership of c is transferred to the group. Conversely, abandon produces an exclusive permission for the cell c that is abandoned: the group relinquishes the ownership of c.

Abandon must be carefully controlled. If a cell could be abandoned twice, two permissions for it would appear, which would be unsound. Due to aliasing, though, it is difficult to statically prevent this problem. Instead, we decide to record at runtime which object is a member of which group, and to verify at runtime that abandon is used in a safe way.

Figure 7. An interface for bags

Figure 8. A FIFO implementation of bags
7.2 Details

Let us now explain in detail the dynamic semantics of adoption and abandon (what these operations do) as well as their static semantics (what the type-checker requires).

Adopter fields  We maintain a pointer from every adoptee to its adopter. Within every object, there is a hidden "adopter" field, which contains a pointer to the object's current adopter, if it has one, and null otherwise. This information is updated when an object is adopted or abandoned. In terms of space, the cost of this design decision is one field per object. It is possible to lessen this cost by letting the programmer declare that certain objects cannot be adopted and don’t need this field.

The type dynamic  The permission “c @ dynamic” guarantees that c is a pointer to a memory block (as opposed to, say, an integer value, or a function value) and grants read access to the field c.adopter. This can be used to verify the identity of c’s adopter. In other words, “c @ dynamic” can be viewed as a permission to perform a dynamic group membership test. It is a duplicable permission. It appears spontaneously when c is known to be a (mutable) object; this is stated by the rule DYNAMICAPPEARS in Figure 6.

In the bag implementation, shown in Figure 8, the head and tail fields of a non-empty bag object, as well as the next field of every cell object, have type dynamic (lines 2 and 6). Because dynamic is duplicable, sharing is permitted: for instance, the pointers b.head and b.tail might happen to be equal.

Adopts clauses  When a cell c is adopted, the exclusive permission that describes it, namely “c @ cell a”, disappears. Only “c @ dynamic” remains. As a result, the information that c is a cell is lost; the type-checker can no longer tell how many fields exist in the object c and what they contain. When the bag b later abandons c, we would like the permission “c @ cell a” to re-appear. How can the type-checker recover this information?

Fortunately, when b abandons c, the type-checker has access to the type of b. Thus, provided the type of the adopter determines the type of its adoptees, this problem is solved.

For an object b of type t to serve as an adopter, where t is an algebraic data type, we require that the definition of t contain the clause “adopts u” and that t and u be exclusive types. This is illustrated in Figure 8, where the definition of “bag a” says “adopts cell a” (line 7).

Because the type of the adoptees must not be forgotten when an algebraic data type is unfolded, structural permissions also carry an adopts clause. In the case of bags, for instance, the permission “b @ bag a” is refined by the match constructs of lines 16 and 29 into either “b @ Empty { head, tail: () } adopts cell a” or “b @ NonEmpty { head, tail: dynamic } adopts cell a”, and, conversely, either of these permissions can be folded back to “b @ bag a”.

We write that “t adopts u” if either t is an algebraic data type whose definition contains the clause “adopts u” or t is a structural type that contains the clause “adopts u”.

Adoption  The syntax of adoption is “give c to b”. This instruction requires two permissions “c @ u” and “b @ t”, where t adopts u (Give, Figure 4). At the program point that follows this instruction, the permission “b @ t” remains available, but “c @ u” has been consumed. Fortunately, not everything about c is forgotten. The permission “c @ dynamic”, which is present before the adoption instruction because “c @ u” spontaneously gives rise to “c @ dynamic”, remains present after adoption.

The runtime effect of this operation is to write the address b to the field c.adopter. The exclusive permission “c @ u” guarantees that this field exists and that its value, prior to adoption, is null.

In the bag implementation (Figure 8), adoption is used at the beginning of insert (line 15), after a fresh cell c has been allocated and initialized. This allows us to maintain the (unstated) invariant that every cell that is reachable from b is adopted by b.

Abandon  The syntax of abandon is “take c from b”. This instruction requires “b @ t” and “c @ dynamic”, where t adopts “u”, for some type u (Take, Figure 4). After this instruction, “b @ t” remains available. Furthermore, the permission “c @ u” appears.

The runtime effect of this operation is to check that the field c.adopter contains the address b and to write null into this field, so as to reflect the fact that b abandons c. If this check fails, the execution of the program is aborted.

In the bag implementation (Figure 8), abandon is used near the beginning of retrieve, at line 33. There, the first cell in the queue, b.head, is abandoned by b. This yields a permission at type “cell a” for this cell. This permission lets us read b.head.elem and b.head.next and allows us to produce the permission “x @ a”, where x is the value found in b.head.elem.

Abandon and adoption are also used inside insert, at lines 22 and 24. There, the bag b is non-empty, and the cell b.tail must be updated in order to reflect the fact that it is no longer the last cell in the queue. However, we cannot just go ahead and access this cell, because the only permission that we have at this point for this cell is at type “dynamic”. Instead, we must take the cell out of the group, update it, and put it back. This well-parenthesized use of take and give is related to Fähndrich and Deloc’s “focus” [16] and to Singh’s “expose” [15]. We allow writing “taking b.tail from b begin ... end” as sugar for such a well-parenthesized use.

7.3 Discussion

To the best of our knowledge, adoption and abandon are new. Naturally, the concept of group, or region, has received sustained interest in the literature [11, 12, 16, 31]. Regions are usually viewed either as a dynamic memory management mechanism or as a purely static concept. Adoption and abandon, on the other hand, offer a dynamic ownership control mechanism, which complements our static permission discipline.

Adoption and abandon are a very flexible mechanism, but also a dangerous one. Because abandon involves a dynamic check, it can cause the program to encounter a fatal failure at runtime. In principle, if the programmer knows what she is doing, this should never occur. There is some danger, but that is the price to pay for a simpler static discipline. After all, the danger is effectively less than in ML or Java, where a programming error that creates an undesired alias goes completely undetected—until the program misbehaves in one way or another.

One might wonder why the type dynamic is so uninformative: it gives no clue as to the type of the adoptee or the identity of the adopter. Would it be possible to parameterize it so as to carry either information? The short answer is negative. The type dynamic is duplicable, so the information that it conveys should be stable (i.e., forever valid). However, the type of the adoptee, or the identity of the adopter, may change with time, through a combination of strong updates and give and take instructions. Thus, it would not make sense for dynamic to carry more information.

That said, we believe that adoption and abandon will often be used according to certain restricted protocols, for which more information is stable, hence can be reflected at the type level. For instance, in the bag implementation, a cell only ever has one adopter, namely a specific bag b. In that case, one could hope to work with a parameterized type dynamic b, whose meaning would be “either this object is currently not adopted, or it is adopted by b”.

Ideally, dynamic’ would be defined on top of dynamic in a library module, and its use would lessen the risk of confusion.
We now describe two other mechanisms, namely nesting and locks.

Adoption and abandon is not the only way of sharing mutable data.

8. Other means of permitting sharing

As promised, this permission can be folded back to \( b \oplus \text{bag} a \) so that, when \text{insert} completes, we are able to return \( b \oplus \text{bag} a \), as promised.

8. Other means of permitting sharing

Adoption and abandon is not the only way of sharing mutable data. We now describe two other mechanisms, namely nesting and locks.

8.1 Nesting

Nesting is irreversible, such a witness is duplicable.

Once \( P \) has been nested in \( x \), whoever has exclusive ownership of \( x \) may decide to temporarily recover \( P \). This is done via two symmetric operations, say “focus” and “defocus”, which in the presence of \( P \prec x \) convert between \( x \otimes t \) and \( P \ast (P \rightarrow x \otimes t) \) (where the type \( t \) is arbitrary, but must be exclusive). The permission \( P \rightarrow x \otimes t \) means that \( P \) has been “carved out” of \( x \). While this is the case, \( x \otimes t \) is temporarily lost: in order to recover it, one must give up \( P \). Thus, it is impossible to simultaneously carve two permissions out of \( x \).

Nesting subsumes Fähndrich and DeLine’s adoption and focus [16]. We view it as a purely static cousin of adoption and abandon. Adoption is more flexible in several important ways: it allows accessing two adoptees at the same time, and allows abandoning an object forever. Nesting has advantages over adoption and abandon: it cannot fail at runtime; it has no time or space overhead; one may nest a permission, whereas one adopts an object; and nesting is heterogeneous, i.e., an object \( x \) can nest multiple distinct permissions, whereas, in the case of adoption and abandon, all adoptees of \( x \) must have the same type.

Nesting can be axiomatized in \textit{Mego} as a library, whose interface appears in Figure 9. In principle, this requires extending the meta-theoretic proof of type soundness; we have not yet done so. We believe that, if applicable, nesting is preferable to adoption. However, adoption and abandon is more widely applicable.

In the case of bags (§7), retrieve takes a cell out of the group in order to extract the element that it contains. If one chooses to use nesting instead of adoption and abandon, then one cannot permanently take the cell out of the group. Thus, the cell must remain in the group; but, in that case, one must take the ownership of the element away from the cell. This forces one to allow a cell to possibly contain no element, hence re-introduces the need for a dynamic check.

8.2 Locks

Dynamically-allocated locks in the style of concurrent separation logic [25, 18, 19, 8] are another dynamic mechanism for mediating access to a permission. A new lock, of type lock \( P \), where \( P \) is an arbitrary permission, is created via a function \text{new}. The functions \text{acquire} and \text{release} both take the lock as an argument; \text{acquire} produces the permission \( P \), which \text{release} consumes. The type lock \( P \) is duplicable, so an arbitrary number of threads can share the lock and simultaneously attempt to acquire it. Within a critical section, delimited by \text{acquire} and \text{release}, the “lock invariant” \( P \) is available, whereas, outside of it, it is not. The “invariant” \( P \) can in fact be broken within the critical section, provided it is restored when one reaches the end of the section.

Locks introduce a form of \textit{hidden state} into the language. Because the permission \( l \otimes \text{lock} P \) is duplicable, it can be captured by a closure. As a result, it becomes possible for a function to perform a side effect, even though its type does not reveal this fact (the pre- and postcondition are empty). \textit{Mego}’s modest library for memoization exploits this feature.

Locks can be used to encode “weak” (duplicable) references in the style of ML and duplicable references with affine content in the style of Alms [32], both of which support arbitrary sharing.

\begin{figure}[h]
\centering
\begin{verbatim}
abstract nests (x : term) (p : perm) : perm
fact duplicable (nests x p)

val nest: [p : perm, a]
  exclusive a =>
  (x: a | consumes p) =>
  (! nests x p)

abstract punched (a : type) (p : perm) : type

val focus: [p : perm, a]
  exclusive a =>
  (consumes x: a | nests x p) =>
  (! x @ punched a p \&\& p)

val defocus: [p : perm, a]
  (consumes (x: punched a p \&\& p)) =>
  (! x @ a)
\end{verbatim}
\caption{A simplified axiomatization of nesting}
\end{figure}
Locks can be axiomatized in Mezzo as a library, whose interface appears in Figure 10. Again, this requires extending the proof of type soundness; we have not yet done so. We view locks as complementary to adoption and abandon and nesting. In a typical usage scenario, a lock protects an adoptor, which in turn controls a group of adoptees (or of nested permissions). This allows a group of objects to be collectively protected by a single lock. It should be noted that (we believe) adoption and abandon are sound in a concurrent setting.

9. Related work

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

Ownership Types [10] 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 [14] 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 [5, 7, 17] 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 [28] departs from this approach and obeys a principle that we dub “owner-as-asseter”. (In O’Hearn’s words, “ownership is in the eye of the asserter” [25].) 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 as- signed assertions are at once structural descriptions and claims of ownership.

Mezzo follows the “owner-as-asseter” principle. In the future, this should allow us to annotate permissions with logical assertions and use that as a basis for the specification and proof of Mezzo programs. A tempting research direction is to translate Mezzo into \( \Pi^* [30] \). This purely functional programming language is equipped with affine values, with powerful facilities for expressing program specifications and proofs, and with a notion of proof erasure. Although our permission discipline is partly inspired by separation logic [28], it is original in several ways. It presents itself as a type system, as opposed to a program logic. This makes it less expressive than a program logic, but more pervasive, in the sense that it can (and must) be used at every stage of a program’s development, without proof obligations. It distinguishes between immutable and mutable data, supports first-class functions, and takes advantage of algebraic data types in novel ways.

As far as we know, Ownership or Universe Types cannot express uniqueness or ownership transfer. Müller and Rudich [23] extend Universe Types with these notions. They rely on the fact that each object maintains, at runtime, a pointer to its owner. The potential analogy with our adopt\( \alpha \)er fields deserves further study.

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 [29]. Linear [1] and affine [32] 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.

Regions [29, 16, 1] 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.

Gordon et al. [17] ensure data-race freedom in an 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.

Plaid [2] and Mezzo exhibit several common traits. A Plaid object does not belong to a fixed class, but can move from one “state” to another: this is related to Mezzo’s tag update. Methods carry state pre- and postconditions, which are enforced via permissions [5]. Plaid is more ambitious in that states are organized in an extensible hierarchy, whereas algebraic data types are flat and closed.

10. Conclusion and future work

Mezzo is a high-level functional and imperative programming language where the traditional concept of “type” is replaced with a more powerful concept of “permission”. Distinguishing between duplicable, exclusive, and affine permissions allows reasoning about state changes. We strive to achieve a balance between simplicity and expressiveness by marrying a static discipline of permissions and a novel dynamic form of adoption and abandon. By adding other mechanisms for controlling sharing, such as nesting and locks, we augment the expressiveness of the language and emphasize that the permission discipline is sufficiently powerful to express these notions. Mezzo is type-safe: well-typed programs cannot go wrong (but an abandon operation can fail). We have carried out a machine-checked proof of type safety [26]. In the future, we would like to extend Mezzo with support for shared-memory concurrency. We believe that, beyond locks (§8.2), many abstractions (threads, channels, tasks, etc.) can be axiomatized so as to guarantee that well-typed code is data-race-free.

A. Surface syntax

So far, we have been fairly imprecise regarding the syntax of Mezzo. The examples shown in sections 2 and 7 use special syntactic conventions, while the formal definition of the typing rules (§5) does not take them into account. This section clarifies the rules for the syntax of Mezzo.

We saw in various examples that the user is allowed to write name introductions, consumes keywords, and benefits from a special convention for function types; conversely, the internal syntax has no such convention for function types, and makes no use of the other two constructs. Thus, there is a difference between the syntax that the user manipulates, which we call the external syntax (or surface syntax), and the representation that Mezzo internally uses, which we call the internal syntax.

Section 6 briefly detailed the differences between the two syntaxes, as well as the procedure one can use to convert from the former to the latter. We now formally define the syntax of Mezzo. We separate external constructs from internal constructs; we introduce kind-checking rules; we define what it means for a type in the surface syntax to be well-formed; we show how to translate the constructs from the surface syntax into constructs from the inter-
A.1 Differentiating the two syntaxes

As section 6 explained, the interpretation of an arrow type differs, depending on whether one uses the surface syntax or the internal syntax. To account for that difference, we clarify what we mean when writing an arrow type: an arrow type is either the external variant, or the internal variant. These two arrows are now denoted by different symbols (Figure 11).

Therefore, we do not see the external and internal syntaxes as two separate syntactical categories; rather, they are both restrictions of the general syntax of types. The external arrow, the name introduction, and the consumes keyword (all underlined) may only appear in the external syntax, while the internal arrow may only appear in the internal syntax. As the name implies, the internal arrow is not exposed to the user. All other constructs may be used freely both in the external and internal syntaxes.

Thus, translating from the external syntax to the internal version amounts to removing all the underlined constructs, replacing them with other constructs.

Function expressions need to be translated as well, as an expression of the form fun \( t_1 : t_2 \rightarrow e \) contains an implicit function type \( t_1 \rightsquigarrow t_2 \), meaning the function declaration benefits from the same syntactical conventions as the corresponding external function type. Furthermore, the external syntax for anonymous functions admits a type as its argument, which is then interpreted as a pattern. For instance, \( (x : t, y : u) \) is a tuple type, which we interpret later on as a pattern binding names \( x \) and \( y \). In order to clarify this as well, we introduce a syntax for internal anonymous functions (Figure 11), which is closer to the one found in the \( \lambda \)-calculus. We write an internal function as \( \lambda(x : t_1) : t_2 \), where \( x \) is the name of the argument, bound in the function body \( e \), \( t_1 \) the type of argument \( x \) and \( t_2 \) the return type for the function.

A.2 Binding rules

A.2.1 Environments

To keep track of the names that are available in the current context, we introduce naming environments. They are denoted as \( \Gamma \), and are made up of lists of pairs \((x, \kappa)\) where \( x \) is the name of the binder and \( \kappa \) its kind. Adding a variable into an environment masks the variable definition of the previous variable. Concatenating environments with masking is done using a semicolon, as in \( \Gamma_1; \ldots; \Gamma_n \).

When merging several environments, we may wish to assert that the environments \( \Gamma_i \) do not bind the same names. We use \( \bigcup \Gamma_i \) for that purpose.

A.2.2 Non-lexical scope

We want to define the kinding rules for the entire syntax of Mezzo; that is, we do not want a separate set of rules for both the external and the internal syntax. Thus, the kinding rules should operate on both syntaxes; in particular, we need to define kinding rules on the external syntax. This means that we ought to clarify the binding rules that govern the usage of our special name introduction construct \( x : t \).

Our rules for binding names are non-standard, in the sense that we separate the name introduction and the binding point: our.binders are not lexically-scoped.

As an example, consider the type of the :: function, which assigns a value into a reference (as seen in §6). Our references have a unique owner, which makes it possible for this function to change the type of the reference. As a consequence, the post-condition of the function needs to mention the function argument.

\[
\forall a, b, (\text{consumes } x : \text{ref } a, \text{consumes } b) \rightsquigarrow (| x \oplus \text{ref } b)
\]

In this arrow type, the name \( x \) is made available both on the left-hand side of the function, and in the right-hand side; \( x \) is not lexically-scope. We say that the left-hand side of this function type introduces the name \( x \), and that the binding point for the name \( x \) is immediately above the function type.

The user may want to introduce names at arbitrary depth: under a consumes keyword, in the field of a concrete data type, in a tuple... Therefore, even though a name may be introduced in-depth, we want it to be reachable as broadly as possible. For that purpose, we “collect” the names that appear in a type, and make them available in the entire type. In the example above, the name \( x \) is collected and made available everywhere in the function type.

The collection procedure should only descend into certain types, which we call transparent. Transparent types are conjunctive in nature: structural types (tuples, concrete types) and conjunctions (mode and type, permission and type, permission and permission) are transparent. It is unclear whether some constructs should be transparent or not; there is a design space to explore, and we chose to make some constructs opaque. For instance, names are not collected below quantifiers.

We introduce the BV function (Figure 13), where \( BV(t) \) is the set of names introduced by type \( t \). This function recursively descends into transparent constructs and stops at opaque constructs. Therefore, any traversal of a type will need to operate consistently with \( BV \). More precisely, after traversing an opaque construct, it is mandatory to extend the working environment with the names found below the construct, before resuming the traversal.

The kinding rules naturally operate consistently with \( BV \), through the use of \( \Gamma \vdash t : \kappa \), which first extends \( \Gamma \) with the names found in \( t \) before kind-checking \( t \). As an example, the kind-checking rules for the opaque construct \( \forall(x : \kappa) \) rely on \( \Gamma \), in their premise. In the particular case of the external function type \( t_1 \rightsquigarrow t_2 \), the names that \( t_1 \) introduces are made available in \( t_2 \) as well.

In the rest of the discussion, we assume environments to also contain the names of all the data types, along with their respective kinds (Figure 13).

A.3 Kinding rules

Kind-checking is defined in Figure 12. The relation performs several checks simultaneously:

- it ensures that all types are well-kindled;
- it ensures that all names are bound according to the binding rules of the surface syntax;
- it ensures that consumes annotations only appear in the left-hand side of an arrow type;
it ensures that all type applications are complete; no partial type applications are allowed, as our system explicitly disallows it.

Our well-kindness judgements are of the form $\Gamma \vdash t : \kappa$, meaning that in environment $\Gamma$, the type $t$ satisfies all the above conditions, and has kind $\kappa$. We intentionally restricted the form of rules $K$-ANCHORED and $K$-SINGLETON; since the only types with kind term are variables, we made it explicit that the only well-kindred singleton types and atomic permissions are of the form $\equiv x$ and $x \otimes t$.

In order to ensure that consumes annotations only appear in the left-hand side of an arrow type, we introduce an extra $s$ variable. In the case of the $K$-CONSUMES, the side $s$ has to be left. Initially, the side is right, and changes to left as soon as one sees an external arrow ($K$-EARRROW).

We define a $\perp$ variant, which sets the side to right, meaning that no consumes annotation may appear here, and extends the environment with the names introduced by $t$. As explained before, the kind-checking relation needs to be consistent with the $BV$ function. Therefore, this variant is used whenever an opaque argument is crossed, such as in the premise of $K$-FORALL. The relation $\perp$ is also the entry point of the kind-checking relation, which we use to check top-level data types definitions, as well as types that appear in expressions, for instance in function definitions or type annotations.

One important rule is $K$-ARRROW. Both $t_1$ and $t_2$ are checked in an environment $\Gamma'$ which contains the names introduced by $t_1$. This means that the names introduced in the domain $t_1$ of the function are available in both the domain $t_1$ and the codomain $t_2$, while names introduced in the codomain $t_2$ remain available in the codomain only.

### A.4 Translating the surface syntax

In order to translate from the surface syntax down to the internal syntax, three constructs must be removed: name introductions, consumes annotations and external arrows. We describe a set of transformations that perform the translation from the surface syntax down to the internal syntax.

The translation step assumes that all well-formedness checks (as described in §A.2.2) have been performed.

#### A.4.1 Removing name introductions

We first need to remove all name introductions, that is, constructs of the form $x : t$. The binding checks have been performed in the kind-checking process already; in order to ensure that the translation is faithful to the binding rules implemented by the kind-checking process, we need to make sure that for every call to $\triangleright$, performed...
in the kind-checking rules, we introduce explicit binders, either existential or universal. We introduce our translation $\triangleright$, and a variant $\triangleleft$, defined in rule T-EXTEND-EXISTS, which introduces explicit existential binders to account for the names introduced by a type. For every use of $\triangleright$ in the kinding rules, the translation makes use of $\triangleleft$.

**Lemma 1** (Well-kindness preservation). If $t$ is well-kind, i.e. $\Gamma \vdash t : \kappa$, and $t$ translates to $t'$, i.e. $t \triangleright t'$, then $t'$ is similarly well-kind, i.e. $\Gamma \vdash t' : \kappa$.

The one explicit call to $BV$ in the kinding rules (K-EARROW) is reflected by call to $BV$ in the corresponding translation rule (T-EARROW). This defines names introduced by the domain $t_1$ of a function type $t_1 \triangleleft t_2$ to be universally quantified binders, enclosing an internal arrow.

The rule T-NAMEINTRO describes what one means when using a name introduction: $x : t$ represents both a pointer to an element named $x$, and a permission stating that $x$ has type $t$. As $\triangleright$ is consistent with $BV$, we can assume that the variable $x$ has been introduced via an explicit binder above us.

**Lemma 2** (Name introduction removals). If $t \triangleright t'$, then $t'$ contain no occurrences of the name introduction construct.

As an example, consider a type $(x : t, \equiv x)$. This type annotation describes the type of a tuple whose two components are equal; we name them $x$, and $x$ has type $t$. This type will be translated using $\triangleright$ into:

$$\exists(x : \mathit{term}),(\equiv x \mid x \equiv t_1) \equiv x$$

**A.4.2 Interpreting consumes annotations**

The consumes keyword receives a special treatment. The kind-checking rules made sure consumes keywords only appear in the left-hand side of external arrows. Rule T-EARROW therefore defines how to interpret consumes annotations that appear in the domain of an external arrow. This translation step changes the meaning of function types; hence, it goes from the external arrow $\triangleright$ to the internal arrow $\triangleleft$.

The consumes keyword is all about ownership: intuitively, the ownership of a function argument will be, by default, returned to the caller, except for the sub-parts of the arguments that are marked with a consumes keyword.

Let us now describe the meaning of the rule in greater detail. We write $[t_1/t_2]t$ as “substitute $t_1$ for $t_2$ in $t$”; $\top$ is understood to mean either unknown (which has kind type) or empty (which has kind perm) depending on which one is appropriate. We introduce a name for the argument, so that we can talk about it in the post-condition of the function. We then take a permission for the “full” argument, and return a modified permission for the argument, where all parts marked as being consumed have been “carved out”.

One important property is that $t'$ no longer contains name introductions, as they have been removed by the recursive call to $\triangleright$. This is important: name introductions only make sense with the semantics of the external arrow and have no meaning when using within an internal arrow.

**Lemma 3** (Consumes and external arrow removals). If $t \triangleright t'$, then $t'$ contain no occurrences of the consumes keyword and of the external arrow $\triangleright$.

The consumes keyword can appear at any depth in the type of the argument: a sophisticated function may wish, for instance, to consume the ownership of just a single field of a data structure. As a final example, consider the type of the swap function, that swaps the two components of a mutable pair. The type of this function can be written, with the surface syntax conventions, as $[a, b] \text{ (consumes } x \text{: mpair a b)} \triangleright ([x @ mpair b a], [x @ mpair a b])$, which we believe is concise: yet intuitive notation for the internal type, which we saw earlier:

$$\forall a. \forall b. \forall(x : \mathit{term}) ([x] \equiv x \mid x \equiv mpair a b) \rightarrow ([x @ mpair b a], [x @ mpair a b])$$
Expressions may contain types. Types appear for instance in type applications, which are used for instantiating polymorphic calls; they also appear in type annotations. We extend $\triangleright$ and define a translation for these expressions. The rules, which are to be found in Figure 14, simply perform a translation of the types using $\triangleright$.

A more complex rule is $T\rightarrow\text{FUN}$, which translates an external anonymous function into an internal one. Let us review the various applications, which are used for instantiating polymorphic calls; expressions may contain types. Types appear for instance in type annotations. We extend $\triangleright$ and define a translation for these expressions. The rules, which are to be found in Figure 14, simply perform a translation of the types using $\triangleright$.

Finally, the internal $\lambda$-abstraction takes a single argument which we name $x$; we recover the names that the user provided by binding the pattern $p$ to variable $x$ in the body of the $\lambda$-abstraction.

We trivially extend $\triangleright$ to be the identity for the other constructs in the syntax of expressions.

### B. Modes and facts

We have mentioned previously (§4.1, Figure 4) predicates of the form “is duplicable” or “is exclusive”. We now provide a formal definition of those predicates, as well as a more general form which we call a fact. We define modes and facts, both of which belong to a lattice, and we give details for their computation.

Modes form a lattice shown in Figure 18. Any type can be affine, but affine is a strict superset of duplicable and exclusive (§4.1). We define two mutually exclusive judgements “$t$ is duplicable” and “$t$ is exclusive” (Figure 16, Figure 17). A type that satisfies neither judgement is affine.

The “$t$ is duplicable” judgement is defined co-inductively. Rule $D\rightarrow\text{APP}$ states that a type application is duplicable if and only if all the unfoldings of the type being applied are themselves duplicable. The “$t$ is exclusive” is only defined for a small subset of types, as the only exclusive types are concrete types and type applications that belong to an mutable-defined data type.

### References


