The implementation of the *Mezzo* type-checker

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

Abstract

We present the implementation of *Mezzo*, a programming language based on the notion of *permission* that provides strong guarantees about aliasing, ownership and mutable state. The unique features of *Mezzo* make its implementation challenging, both from a formal and a technical perspective. In this paper, we formalize the core operations used for type-checking, and give corresponding algorithms. Two central operations, *subtraction* and *merging*, are detailed. Both operations are closely related to shape analysis and frame inference.

1. Introduction

Mezzo is a strict, functional programming language in the spirit of ML [9]. *Mezzo* is based on the concept of permission, which blends together the *type* of an object as well as the *state* the object is in.

The type system of *Mezzo* differs significantly from that of ML. Writing a type-checker for the language thus poses several new challenges. The purpose of this paper is to show how we managed to design a type-checker that could account for the many novel features of the *Mezzo* type system.

One could think of *Mezzo* as separation logic turned into a type system. Our main contribution is a presentation of type-checking that is capable of dealing with the *frame inference* problem (computing the portions of the heap that are left untouched by a call to a sub-routine) and the *join* problem (finding a description of the heap that subsumes two, more precise descriptions), within the context of ML. This implies dealing with arbitrary, higher-order quantifiers; with duplicable and non-duplicable portions of the heap; with subtyping relations. These issues, to the best of our knowledge, have not yet been dealt with in the literature. Before discussing the type-checking of *Mezzo*, we offer a quick introduction to help the reader become acquainted with the core concepts of the language. We use the example of the `map` function, which will be recurring throughout the present paper. We refer the reader to a more exhaustive document [9] for a more thorough introduction to the language[1].

[1] The reader familiar with *Mezzo* already should know that in the present document, we omit the syntactic conventions of the surface syntax of *Mezzo*. Therefore, the `consumes` keyword is not mentioned, and neither is the name introduction construct. This means that, in the present paper, any permission found in a function's argument will be consumed from the caller. We also do not mention the adoption/abandon mechanism, and discard mode constraints from the discussion.

1.1 Introduction to permissions

The syntax of types is detailed in Figure 1, while the syntax of expressions is omitted, as it forms a fairly standard dialect of ML. Types have kinds: permissions (`empty`, $p * q$, x @ t) have kind `perm`, regular types (arrow, constructor, tuple) have kind `type`. The only types with kind `term` are program variables x. We informally refer to some of the type-checking rules; these are discussed more formally in §2.

Let us start by introducing the definition of *immutable* lists.

```
data list a =
  | Nil
  | Cons { head: a; tail: list a }
```

If `e1` is an expression that creates a list of elements with type `a`, writing "`let x = e1 in e2`" generates a new *permission* "`x @ list a`", which is available in `e2`. Permissions do not exist at runtime, and are purely an artefact of the type system.

At any program point, a *current permission* is available. When execution starts, the current permission is `empty`. As program execution advances, the current permission evolves, possibly becoming the *conjunction* of several permissions.

Permissions have an ownership reading: the permission "`x @ list a`" not only grants its owner the right to use `x` as a list, but it also embodies the *ownership* of the list. Whoever owns "`x @ list a`" owns a *list spine*, made up of a succession of `Cons` cells, as well as the elements themselves. Here, `a` is a type variable; the ownership reading of the elements depends on the actual instantiation of `a`.

Permissions may disappear, for instance when calling a function. Here is the type of the `map` function. Square brackets denote universal quantification.

```
val map: [a, b] ((a -> b), list a) -> list b
```

The `map` function requires a pair of a function and a list. What happens whenever one wishes to call the `map` function is described in rule Application (Figure 3). Calling `map` with `(f, l)` requires exhibiting the permissions "`f @ a -> b`" and "`l @ list a`". The signature of `map` indicates that the function does not hand back these permissions: they are *consumed*. The caller will, however, obtain a fresh permission "`ret @ list b`", where `ret` denotes the return value of the function.

In *Mezzo*, permissions are either *duplicable* or *exclusive*. A duplicable permission denotes shared, read-only knowledge, while an exclusive permission denotes uniquely-owned, read-write knowledge. The unique-owner property allows the owner of a mutable object to change its type, which is safe as the system guarantees no one else knows about the object. A third mode, *affine*, exists, which is a superset of the two.

Duplicable permissions can be copied at any time. Function types, for instance, are duplicable. Back to the `map` example, this means that the caller will not lose the permission "`f @ a -> b`", since it can save a copy of it before calling `map`. What about "`l @ list a`"?

Determining whether "`l @ list a`" may be copied or not is non-trivial, and depends on what the type variable `a` happens to be. If `l` is a list of duplicable elements, e.g. if we have "`l @ list int`", then this permission denotes a read-only fragment of the heap. It is thus duplicable. If, however, `l` is a list of exclusive elements, e.g. if we have "`l @ list (ref int)`", then the permission is *affine*[2].

1.2 Type-checking by example

Let us consider the implementation of the `map` function. We give a high-level overview of the type-checking of the function. This allows subsequent sections to pick specific steps of the type-checking to illustrate the discussion.

```
1  val rec map [a, b] (
2    f: (a -> b),
3    l: list a
4  ): list b =
5  match l with
6  | Nil ->
7      l
8  | Cons { head = h; tail = t } ->
9      let h' = f h
10     and t' = map f t in
11     Cons { head = h'; tail = t' }
12 end
```

Type-checking in *Mezzo* is performed in a forward manner similar to symbolic execution. One starts by assuming the precondition or, in *Mezzo* lingo, by adding the function's arguments to the *current permission*, and then moves forward, updating the permission as type-checking progresses through the program. Thus, at line 5, permissions for `f`, `l` and `map` are available (FUNCTION):

```
f @ a -> b * l @ list a * map @ [a, b] (...) -> list b
```

Functions must be annotated with the type of their arguments, as well as their return type. Functions can only close over duplicable permissions. A function type is therefore duplicable. For the sake of brevity, we will from there on omit the permissions for `f` and `map`. The reader can safely assume that they remain available throughout the whole body of `map`. We thus have:

```
l @ list a
```

When the type-checker enters the `Nil` branch of the `match` construct (line 6), the permission is refined to:

```
l @ Nil
```

`Nil` is a structural type that asserts that `l` is a block in the heap whose tag is `Nil`, that has zero fields.

Naming `ret` the return value of the function, we obtain after line 7:

```
l @ Nil * ret = l
```

This is our first encounter of the *singleton type*. Having the permission "`x @ =y`" means that `x` and `y` are equal; in particular, if they are pointers, they point to the same object in the heap. We write "`x = y`", as syntactic sugar.

[2] `l` itself does not denote a read-write portion of the heap, as the list cells are immutable; the permission "`l @ list a`" is therefore neither duplicable nor exclusive: it is affine.

Similarly, entering the `Cons` branch (line 8), the type-checker refines `l @ list a` into the following permission:

```
l @ Cons { head: a; tail: list a }
```

This permission is expanded (DECOMPOSEBLOCK).

```
l @ Cons { head = h; tail = t } * h @ a * t @ list a
```

Just like the "`ret = l`" example above, we write "`head = h`" to signify "`head: =h`"[3]

The two function calls at lines 9 and 10 consume permissions from the environment (APPLICATION, FRAME): calling `f` consumes "`h @ a`", which is *subtracted* from the environment, while a permission for `h'` is returned. Similarly, calling `map` consumes "`t @ list a`" which is traded for a new permission for `t'`. The two operations are performed in sequence, which gives in the end:

```
l @ Cons { head = h; tail = t } *
h' @ b * t' @ list b
```

The return value, which we call again `ret`, is then created:

```
l @ Cons { head = h; tail = t } *
h' @ b * t' @ list b *
ret @ Cons { head = h'; tail = t' }
```

The type-checker now needs to type-check the `match` construct. Two distinct permissions are available, one for the `Nil` branch, and another one for the `Cons` branch. The type-checker needs to apply subsumption rules on both permissions, until they match. This is called the merge operation, and constitutes the topic of §4. Here, the type-checker sees that `ret` is either `Nil` or `Cons`. It thus decides to reconcile the two permissions for `ret` as "`ret @ list b`".

One may object that the type-checker is doing unnecessary work. Indeed, the return type of the `map` function constitutes a type annotation that tells us already what we should seek to obtain. We could skip the merge operation (reconciling the two permissions) altogether, and merely assert that `ret @ list b` is a permission that is available in both branches. Even though this is actually what our type-checker does in this particular case, type annotations are not always available. We thus discuss the general procedure.

The merge operation decided that we should try to obtain "`ret @ list b`". This implies subtracting "`ret @ list b`" from each branch, which consumes permissions. As the constructor `Nil` is duplicable, we are left, in the `Nil` branch, with:

```
l @ Nil * ret = l
```

while the `Cons` branch now only contains:

```
l @ Cons { head = h; tail = t } *
ret @ Cons { head = h'; tail = t' }
```

The permission "`h' @ b * t' @ list b`" is gone, for `b` is a type variable which is, lacking any other information, considered non-duplicable, or affine. The permission for `ret` is retained, though, as it denotes a `Cons` block in the heap which is immutable, hence duplicable.

Continuing the *reconciliation*, the type-checker infers that "`f @ a -> b * map @ ...`" is a valid conjunction for both branches, meaning that the final conjunction obtained after the `match` construct is:

```
ret @ list b * f @ a -> b * map @ ...
```

The final step consists in matching the final permission above with the expected return type for the function. This is, again,

[3] In practice, the type-checker introduces automatic names for the `head` and `tail` fields when expanding the permission, then performs pattern-matching, which results in additional equations that identify the user-provided names with the type-checker names.

$$
\begin{array}{rll}
\kappa ::= & \mathsf{type} \mid \mathsf{term} \mid \mathsf{perm} \mid \kappa \to \kappa & \text{kind} \\
T, t, P ::= & & \text{type or permission} \\
& X & \text{variable } (a, x, \ldots) \\
& t \to t & \text{function type} \\
& (\vec{t}) & \text{tuple type} \\
& \mathsf{A}\,\{\vec{f} : \vec{t}\} & \text{structural type} \\
& T\,\vec{T} & n\text{-ary type application} \\
& \forall (X : \kappa)\; T & \text{universal quantification} \\
& \exists (X : \kappa)\; T & \text{existential quantification} \\
& {=}x & \text{singleton type} \\
& (t \mid P) & \text{type/permission conjunction} \\
& x \mathbin{@} t & \text{atomic permission} \\
& \mathsf{empty} & \text{empty permission} \\
& P * P & \text{permission conjunction} \\
D ::= & & \text{algebraic data type definition} \\
& \mathsf{data\ mutable?}\ d\ (\vec{X} : \vec{\kappa}) = \vec{b} & \\
& \quad \mathsf{adopts}\ t & \\
b ::= & \mathsf{A}\,\{\vec{f} : \vec{t}\} & \text{algebraic data type branch}
\end{array}
$$

Figure 1. Syntax of types and permissions

achieved by performing the *subtraction* of "`ret @ list b`". The remaining permission is discarded; in this case, the subtraction merely serves as a way to assert that a particular permission is available.

1.3 Challenges for a type-checker

The example of the `map` function highlights several salient points.

- The representation of a permission is not unique; one may "unfold" a structural permission (tuple, constructor) by introducing singleton types; one may also float out quantifiers (ExistsStar) or permissions (MixStar). We start by introducing *prefixed* permissions, along with a normalized representation (§2) of a permission.
- The discussion mentions in several occasions that the type-checker needs to perform a *subtraction*, an operation that is required for implementing the typing rules from Figure 3. This implies recombining permissions using rule Subsumption. Which subsumption rules should be applied? In what order? There is a wide range of options for subsumption. We present an algorithm formulation of subtraction along with an implementation strategy (§3).
- Subtraction allows one to type-check function calls (extracting a permission, obtaining a remainder), function bodies (checking that the return type is satisfied), but we need an additional operation to type-check disjunctions, that is, `match` and `if-then-else` expressions. This is the `merge` operation. We present an algorithm that leverages subtraction (§4).

2. The representation of permissions

Subsumption rules and typing rules are shown in Figure 2 and Figure 3. Our typing judgements are of the form $K; P \vdash e : t$, meaning that under kinding environment K, using permission P, expression e is shown to have type t.

This formulation gives no clue as to how one can effectively type-check a *Mezzo* program. Type-checking a function call requires a combination of Application and Frame: how does one compute the part that is framed out? Similarly, the Let rule mentions $x \mathbin{@} t_1$, but t_1 will most certainly be of the form $(t_1' \mid Q)$ where Q is a permission big enough to type-check e_2 with type t_2. How does one know the value of t_2 or Q?

$$
\begin{array}{rll}
q ::= & & \text{quantifier} \\
& \forall (x : \kappa) & \text{universal (rigid) quantification} \\
& \exists (x : \kappa) & \text{existential (flexible) quantification} \\
& \exists (x : \kappa = \tau) & \text{instantiated existential} \\
\mathcal{V} ::= & \mathsf{nil} \mid q, \mathcal{V} & \text{a prefix is a list of quantifiers}
\end{array}
$$

Figure 4. Syntax of prefixes

2.1 Why is type-checking difficult?

We need to adopt a presentation where we carry a current permission P, and where a type-checking step takes P and returns P', as this will form the basis of an algorithm. Finding the right representation for P is difficult, though. Indeed, a "good" representation shall satisfy quite a few constraints, which we review below.

- The existence of possibly several permissions for the same variable requires *exploration* of a solution space.
- Our conjunction operator $*$, unlike that of separation logic, works on non-exclusively owned portions of the heap. Indeed, some permissions can be duplicated, while some others cannot. When subtracting a permission, whether it will be consumed and disappear depends on whether it is duplicable or not.
- We mention in §1.3 the various, equivalent ways of representing a permission. Function types can also be represented in various equivalent ways: for instance,

  $$x \mathbin{@} \forall a, \forall (y : \mathsf{term}).(=y \mid y \mathbin{@} a) \to () \equiv x \mathbin{@} \forall a, a \to ()$$

  Indeed, first-class functions are central in ML-like languages. We thus need to compare function types during the subtraction, something that other works do not need to do. We cannot use a syntactic criterion, as we often want powerful reasoning when comparing function types (for instance, implicit η-expansion).
- We manipulate quantifiers, both universal and existential. They may stem from user-provided quantifiers (polymorphic function types, one-shot functions), or be introduced by the translation of the name introduction construct found in the surface syntax [9]. They may appear in arbitrary order, which requires careful treatment to ensure they are properly introduced. Moreover, quantifiers may refer to variables that have kind `term` (e.g. x) `type` (e.g. t) or `perm` (e.g. $x \mathbin{@} t$). A quantifier at kind `term` refers to a program variable; this is akin to first-order logic: we're quantifying over the program variable x that may appear in $x \mathbin{@} t$. A quantifier at kind `type` or `perm` is different; such a quantifier may refer to an entire permission $x \mathbin{@} t$, not just x. This would be the equivalent of second-order logic.

Per the first item, we know that we are bound to use a conjunction; a map from variables to types would not work. The second item encourages us to use an expanded representation. Indeed,

$$l \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\} * h \mathbin{@} a * t \mathbin{@} \mathsf{list}\ a$$

is more precise than:

$$l \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} : a; \mathsf{tail} : \mathsf{list}\ a\}$$

The former conjunction tells us that the permission for l is duplicable, while the latter is more conservative and tells us that the permission for l is affine. The last two items highlight the difficulty of dealing with singleton types and quantifiers.

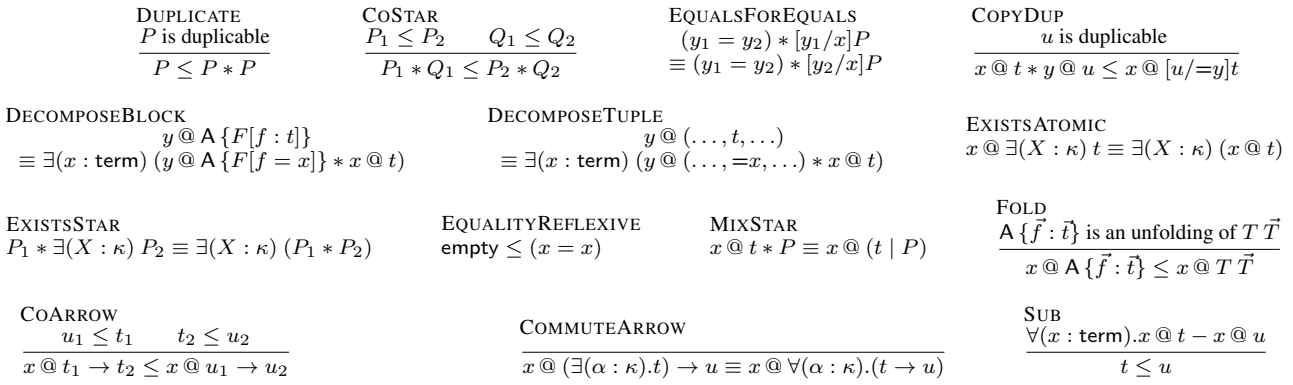

Figure 2. Selected permission subsumption rules

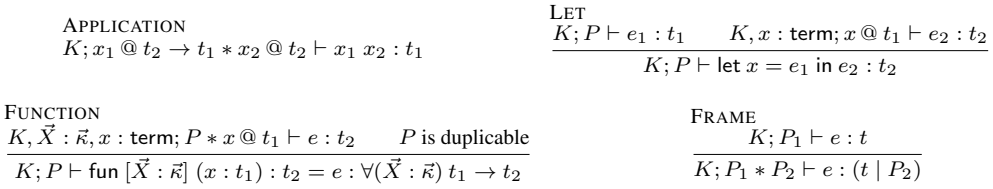

Figure 3. Selected typing rules

2.2 Prefixes

We quantify over the free variables of a permission using a *prefix* $\mathcal{V}$. The syntax of prefixes is presented in Figure 4. The type variables are introduced with a kind κ, which, if missing, is assumed to be type.

A prefix $\mathcal{V}'$ is more precise than $\mathcal{V}$ if $\mathcal{V}'$ can be obtained from $\mathcal{V}$ by:

- *instantiating* existential quantifiers,
- *inserting* existential quantifiers,
- *appending* arbitrary quantifiers.

Type-checking initially starts with the permission empty, then threads it through various program points. Permissions are added (when entering function bodies, or when a function call returns) and removed (when exiting function bodies, or performing a function call). Thus, the type-checker carries a prefixed permission $\mathcal{V}.P$.

We define $\mathsf{restrict}(\mathcal{V}', \mathcal{V})$ to be the smallest prefix of $\mathcal{V}'$ that contains all the variables quantified in $\mathcal{V}$. We see its usage in §3.3.

2.3 Normal form for a permission

We maintain the following invariants.

Existential quantifiers Existential quantifiers are all hoisted into the prefix.

Expanded form All structural permissions (tuples, constructors) are *expanded*, meaning that their fields are all singleton types. All function types have a domain that is a singleton type; structural permissions in the domain of function types are also expanded.

No hidden permissions All permissions nested inside structural types using $(t \mid P)$ are floated out into the outer conjunction.

No redundant permissions Some conjunctions, such as $x \mathbin{@} \mathsf{Nil} * x \mathbin{@} \mathsf{list}\ a$, are redundant. We simplify these.

Let us now explain how these invariants are enforced.

Existential quantifiers The existence of EXISTSATOMIC allows us to move existential quantifiers into the prefix. Along with rules DECOMPOSEBLOCK and DECOMPOSETUPLE, any existentially-quantified type that was found "inside" a tuple or a constructor will be assigned to a fresh name, thus making EXISTSATOMIC applicable. The reason why existential quantifiers must be hoisted out is the topic of §3.5.

Expanded form Combining EQUALITYREFLEXIVE, COSTAR, MIXSTAR and EXISTSINTRO allows one to derive the following subtyping rule, which we call DECOMPOSE:

$$\text{DECOMPOSE}\quad t \equiv \exists(x : \mathsf{term}).((=x \mid x \mathbin{@} t))$$

Intuitively, one can always give a name to the portion of the heap denoted by type t. Rules DECOMPOSEBLOCK and DECOMPOSETUPLE allow us to enforce the expanded form invariant on structural types, while DECOMPOSE allows the invariant to be respected in the domain of functions, in combination with COARROW.

This invariant is important for two reasons. First, having it simplifies the application of rules TUPLE and CONSTRUCTOR, as t is already of the form $=x$, meaning that we don't have to introduce an extra variable y. Second, it reveals "implicit" existential quantifiers which are then moved into the prefix. Again, the technical discussion on why it is important to eagerly move existentials into the prefix takes place in §3.5.

No hidden permissions Decomposition, along with MIXSTAR, allows enforcing this invariant. Having this invariant simplifies the reasoning all throughout the present paper, as we don't have to consider the case where the permission we're looking for is "stashed" inside another one.

No redundant permissions The algorithm applies a last set of rules to further simplify P. The following conjunction is not in-

consistent:

$$x \mathbin{@} \mathsf{list}\ a * x \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\}$$

It can be simplified into:

$$x \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\} * h \mathbin{@} a * t \mathbin{@} \mathsf{list}\ a$$

Another is example is

$$x \mathbin{@} (=y_1, =y_2) * x \mathbin{@} (=z_1, =z_2)$$

which is simplified into

$$x \mathbin{@} (=y_1, =y_2) * y_1 = z_1 * y_2 = z_2$$

A similar rule exists for constructors.

All the subsumption rules used for normalization are *reversible*, meaning that we do not lose information when applying them. Applying a non-reversible rule (e.g. FOLD) is a bad idea, as it will *lose* information about the current permission.

Once the normalization procedure is in place, one can implement the *addition* of a permission, which boils down to applying the normalization rules above. Additions take place when entering function bodies, and when returning from function calls.

2.4 Data structures of the type-checker

The type checker of *Mezzo* is written in OCaml. Representing a current permission P is done using the following data structures.

Persistent union-find We use a union-find structure to keep track of equalities between variables, that is, permissions of the form $x = y$. The union-find is persistent, as it facilitates backtracking and branching on disjunctions. Moreover, the union-find maps the representative of an equivalence class x to a list of types, which represent the permissions available for x.

Floating permissions Permissions that are not of the form $x \mathbin{@} t$ are not attached to a program variable; they are abstract. We dub them "floating permissions", and store them in a separate list.

Flexible variables Flexible variables (introduced later on, in §3.3) are implemented using unification variables. A flexible variable is a globally unique atom. We keep a persistent map of these atoms to a structure containing the variable's level as well as its instantiation, if any. This structure allows inserting existential quantifiers at arbitrary position (FLEX-TUPLE-L).

Levels In order to efficiently check the non-occurrence premise of a rule such as INSTANTIATION, we use levels [10]. This prevents illegal instantiations, and allows for an efficient implementation of the "is local" check during merges (§4). The $\mathsf{restrict}(\mathcal{V}', \mathcal{V})$ operation is implemented by restricting the flexible variables of $\mathcal{V}'$ to the maximum level of $\mathcal{V}$.

3. The subtraction operation

The basic building block of a type-checking algorithm is the *subtraction* operation, as mentioned at numerous occasions in §1.1. A subtraction extracts Q from P, computing the remainder R. This operation is used when exiting function bodies, to check that the return type is satisfied (the remainder R is then discarded), and when calling functions, to compute the part of the current permission that is *consumed* by the function call, and the remainder (the frame). This problem is known as *frame inference* in separation logic.

3.1 Subtraction examples

Subtractions can be found in the `map` example. At line 9, for instance, a call to `f` takes place. Rule APPLICATION requires a permission for `f` that is a function type; we happen to possess one, namely "`f @ a -> b`". Consistently with §1.1, we omit this permission, for brevity. The function is called with argument `h`. We thus need to obtain "`h @ a`" from the current permission; in other words, we need to perform the following subtraction:

```
  l @ Cons { head = h; tail = t } *
  h @ a * t @ list a
- h @ a
```

Will "`h @ a`" disappear? As the type variable `a` is abstract, we have to be conservative: `a` could be anything, and denote uniquely-owned data. We thus take a conservative stance, and assume "`h @ a`" is not a duplicable permission. It is affine, and we cannot save a copy of it. Thus, the subtraction *consumes* the permission, leading to the following remainder:

```
  l @ Cons { head = h; tail = t } * t @ list a
```

(This subtraction is followed by the *addition* of "`h' @ b`" to the current permission, as per the return type of the function `f`. Addition is a much easier operation which we briefly mentioned in §2.3.)

A more sophisticated subtraction takes place when type-checking the whole `match` construct. The type-checker decides (and the reason for this is the topic of §4) to extract "`ret @ list b`" from both branches. Thus, the following subtraction takes place in the `Cons` branch:

```
  l @ Cons { head = h; tail = t } *
  ret @ Cons { head = h'; tail = t' }
  h' @ b * t' @ list b *
- ret @ list b
```

The type-checker uses a syntactic criterion and figures out that if we could show that `ret` is a well-formed `Cons` cell, this would be enough to justify that it is also a list (FOLD). The type-checker hence tries to perform the following subtraction:

```
  l @ Cons { head = h; tail = t } *
  ret @ Cons { head = h'; tail = t' }
  h' @ b * t' @ list b *
- ret @ Cons { head: b; tail: list b }
```

Applying DECOMPOSEBLOCK and floating up existential quantifiers through EXISTSSTAR, this amounts to showing that:

```
∃h'', ∃t'' (
  l @ Cons { head = h; tail = t } *
  ret @ Cons { head = h';  tail = t'  }
  h'  @ b * t'  @ list b *
- ret @ Cons { head = h''; tail = t'' } *
  h'' @ b * t'' @ list b
)
```

One can instantiate `h''` and `t''` with `h'` and `t'` respectively. The subtraction then becomes straightforward. Again, the type variable `b` being abstract, the permission "`h' @ b * t' @ list b`" disappears. Conversely, "`ret @ Cons { head = h'; tail = t' }`" is duplicable[4]: a copy of the permission can be saved through DUPLICATE. The final remainder of the subtraction is thus:

```
  l @ Cons { head = h; tail = t } *
  ret @ Cons { head = h';  tail = t'  }
```

3.2 Formally defining subtraction

The result of a subtraction of Q from P under $\mathcal{V}$ is a remainder R along with a more precise prefix $\mathcal{V}'$. We write:

$$\mathcal{V}.(P - Q) = \mathcal{V}'.R$$

[4] The permission denotes an immutable block, whose fields are singleton types, which are themselves duplicable.

$$\textsc{Subsumption}\ \frac{P \le P' \qquad Q' \le Q \qquad \mathcal{V}.P' - Q' = \mathcal{V}'.R}{\mathcal{V}.P - Q = \mathcal{V}'.R} \qquad \textsc{Empty}\ \mathcal{V}.P - \mathsf{empty} = \mathcal{V}.P \qquad \textsc{Star}\ \frac{\mathcal{V}.P - Q = \mathcal{V}'.R \qquad \mathcal{V}'.R - Q' = \mathcal{V}''.R'}{\mathcal{V}.P - Q * Q' = \mathcal{V}''.R'}$$

$$\textsc{Arrow}\ \frac{\mathcal{V}, \forall(y : \mathsf{term}).\overline{P} * y \mathbin{@} t_1' - y \mathbin{@} t_1 = \mathcal{V}'.P' \qquad \mathcal{V}', \forall(z : \mathsf{term}).P' * z \mathbin{@} t_2 - z \mathbin{@} t_2' = \mathcal{V}''.P''}{\mathcal{V}.P * x \mathbin{@} t_1 \to t_2 - x \mathbin{@} t_1' \to t_2' = \mathsf{restrict}(\mathcal{V}'', \mathcal{V}).P}$$

$$\textsc{App}\ \frac{\mathcal{V}_i, \forall(y : \mathsf{term}).\overline{P} * y \mathbin{@} u_i \overset{v_i}{\leftrightarrows} y \mathbin{@} u_i' = \mathcal{V}'.P' \qquad \mathcal{V}_{i+1} = \mathsf{restrict}(\mathcal{V}', \mathcal{V}_i)}{\mathcal{V}_0.P * x \mathbin{@} t\ \vec{u} - x \mathbin{@} t\ \vec{u}' = \mathcal{V}_n.P}$$

$$\textsc{Forall-L}\ \frac{\mathcal{V}, \exists(\alpha : \kappa).P - Q = \mathcal{V}'.R}{\mathcal{V}.(\forall(\alpha : \kappa).P) - Q = \mathcal{V}'.R}$$

$$\textsc{Forall-R}\ \frac{\mathcal{V}, \forall(\alpha : \kappa).P - Q = \mathcal{V}'.R}{\mathcal{V}.P - (\forall(\alpha : \kappa).Q) = \mathcal{V}'.R} \qquad \textsc{Exists-L}\ \frac{\mathcal{V}, \forall(\alpha : \kappa).P - Q = \mathcal{V}'.R}{\mathcal{V}.(\exists(\alpha : \kappa).P) - Q = \mathcal{V}'.R} \qquad \textsc{Exists-R}\ \frac{\mathcal{V}, \exists(\alpha : \kappa).P - Q = \mathcal{V}'.R}{\mathcal{V}.P - (\exists(\alpha : \kappa).Q) = \mathcal{V}'.R}$$

$$\textsc{Unknown}\ \mathcal{V}.P - x \mathbin{@} \mathsf{unknown} = \mathcal{V}'.P$$

$$\textsc{VariablePerm}\ \frac{}{\mathcal{V}.P * p - p = \mathcal{V}.P} \qquad \textsc{Variable}\ \frac{}{\mathcal{V}.P * x \mathbin{@} \alpha - x \mathbin{@} \alpha = \mathcal{V}.P}$$

$$\textsc{Tuple}\ \frac{\mathcal{V}_i, \forall(y : \mathsf{term}).P_i * y \mathbin{@} t_i - y \mathbin{@} t_i' = \mathcal{V}_{i+1}.P_{i+1}}{\mathcal{V}_0.P_0 * x \mathbin{@} (\vec{t}) - x \mathbin{@} (\vec{t'}) = \mathcal{V}_n.P_n}$$

$$\textsc{Constructor}\ \frac{\mathcal{V}_i, \forall(y : \mathsf{term}).P_i * y \mathbin{@} t_i - y \mathbin{@} t_i' = \mathcal{V}_{i+1}.P_{i+1}}{\mathcal{V}_0.P_0 * x \mathbin{@} \mathsf{A}\,\{\vec{f} : \vec{t}\} - x \mathbin{@} \mathsf{A}\,\{\vec{f} : \vec{t'}\} = \mathcal{V}_n.P_n}$$

$$\textsc{SubstituteFlexible}\ \frac{\mathcal{V}, \exists(\alpha : \kappa = \tau), \mathcal{V}'.[\tau/\alpha]P - [\tau/\alpha]Q = \mathcal{V}''.R}{\mathcal{V}, \exists(\alpha : \kappa = \tau), \mathcal{V}'.P - Q = \mathcal{V}''.R}$$

$$\textsc{Instantiation}\ \frac{\mathcal{V}, \exists(\alpha : \kappa = \tau), \mathcal{V}'.P - Q = \mathcal{V}''.R \qquad \mathcal{V}' \# \tau}{\mathcal{V}, \exists(\alpha : \kappa), \mathcal{V}'.P - Q = \mathcal{V}''.R} \qquad \textsc{ExistsIntro}\ \frac{\mathcal{V}, \exists(\alpha : \kappa), \mathcal{V}'.P - Q = \mathcal{V}''.R}{\mathcal{V}, \mathcal{V}'.P - Q = \mathcal{V}''.R}$$

Figure 5. The rules of subtraction

$$\textsc{Variance-Co}\ \frac{\mathcal{V}.P * x \mathbin{@} t - x \mathbin{@} t' = \mathcal{V}'.P'}{\mathcal{V}.P * x \mathbin{@} t \overset{\mathsf{co}}{\leftrightarrows} x \mathbin{@} t' = \mathcal{V}'.P'} \qquad \textsc{Variance-Contra}\ \frac{\mathcal{V}.P * x \mathbin{@} t' - x \mathbin{@} t = \mathcal{V}'.P'}{\mathcal{V}.P * x \mathbin{@} t \overset{\mathsf{contra}}{\leftrightarrows} x \mathbin{@} t' = \mathcal{V}'.P'}$$

$$\textsc{Variance-Inv}\ \frac{\mathcal{V}.P * x \mathbin{@} t - x \mathbin{@} t' = \mathcal{V}'.P' \qquad \mathcal{V}'.P' * x \mathbin{@} t' - x \mathbin{@} t = \mathcal{V}''.P''}{\mathcal{V}.P * x \mathbin{@} t \overset{\mathsf{inv}}{\leftrightarrows} x \mathbin{@} t' = \mathcal{V}''.P''} \qquad \textsc{Variance-Bi}\ \mathcal{V}.P * x \mathbin{@} t \overset{\mathsf{bi}}{\leftrightarrows} x \mathbin{@} t' = \mathcal{V}'.P'$$

Figure 6. Variance-dependent subtraction

where P and Q may themselves be prefixed. This is analogous to the frame inference problem, for which several algorithms have been proposed [2, 6, 8]. Calcagno *et al.* write this as $\delta \vdash H{*}?\text{frame}$ [4] when assuming an existing procedure for frame inference.

In the algorithmic presentation of the subtraction, we often wish to focus the subtraction on a particular variable x. We write:

$$\mathcal{V}.\big(P * x \mathbin{@} t - x \mathbin{@} t'\big) = \mathcal{V}'.R$$

We sometimes need to keep only the duplicable parts of a permission. If $P = p_1 * \ldots * p_n$, this merely consists in dropping all permissions p_i that do not satisfy the "is duplicable" predicate. We write $\overline{P}$.

3.3 Flexible variables *vs.* rigid variables

In a subtraction $\mathcal{V}.(P - Q)$, one can think of P as our current hypothesis, and of $\mathcal{V}$ and Q as our goal. Thus, Q is, like V, in positive position, while P is in negative position. Consider:

$$\forall(x : \mathsf{term}).\big(x \mathbin{@} \exists a.a - x \mathbin{@} \exists b.b\big)$$

In the example above, the quantifier $\exists a$ is in negative position, meaning that it turns into a universal when moved into the outer prefix. The $\exists b$ quantifier, being in positive position, moves unchanged into the outer prefix. We obtain:

$$\forall(x : \mathsf{term}), \big(\forall a, \exists b.x \mathbin{@} a - x \mathbin{@} b\big)$$

We say that a is a *rigid* variable, while b is an *uninstantiated flexible variable*. This subtraction problem admits the following solution, where b is *instantiated*:

$$\forall(x : \mathsf{term}), \forall a, \exists(b = a).\mathsf{empty}$$

The treatment of flexible and rigid variables is a central issue and constitutes the topic of §3.5.

3.4 An algorithmic presentation of type-checking

The subtraction operation (Figure 5) provides a set of rules that an algorithm can choose to apply; the rules make it explicit which part of a permission is preserved, and which permission is returned. The subtraction rules work at a lower level than the type-checking rules; yet, they leave plenty of non-determinism, meaning that an actual algorithm needs an implementation strategy in order to know which rules should be applied (§3.5).

Rule Subsumption is central, as it allows powerful recombinations of a permission on both sides of the subtraction (§3.6, §2.3).

Rules Empty and Star are standard. The $*$ conjunction being commutative and associative, rule Star leaves the order of subtractions unspecified. The algorithm is free to apply an efficient strategy (§3.6).

Rules Unknown and Variable are standard. There is no subtraction rule for singleton types as it can be derived from EqualityReflexive and Subsumption.

Rules Forall-L, Forall-R, Exists-L and Exists-R deal with quantifiers; they are discussed in §3.5. The right-hand side of a subtraction preserves the nature of the quantifier (it refers, like the prefix, to the *goal* that we're trying to prove); the left-hand side, however, flips quantifiers (it is our current *hypothesis*).

$$\textsc{Flex-Tuple-L}\ \frac{\mathcal{V}, \exists(\vec{\beta} : \vec{\kappa})\exists(\alpha : \kappa = (\vec{\beta}))), \mathcal{V}'.P * x \mathbin{@} (\vec{t}) - x \mathbin{@} \alpha = \mathcal{V}''.R}{\mathcal{V}, \exists(\alpha : \kappa), \mathcal{V}'.P * x \mathbin{@} (\vec{t}) - x \mathbin{@} \alpha = \mathcal{V}''.R}$$

$$\textsc{Flex-Constructor-L}\ \frac{\mathcal{V}, \exists(\vec{\beta} : \vec{\kappa})\exists(\alpha : \kappa = \mathsf{A}\,\{\vec{f} : \vec{\beta}\})), \mathcal{V}'.P * x \mathbin{@} \mathsf{A}\,\{\vec{f} : \vec{t}\} - x \mathbin{@} \alpha = \mathcal{V}''.R}{\mathcal{V}, \exists(\alpha : \kappa), \mathcal{V}'.P * x \mathbin{@} \mathsf{A}\,\{\vec{f} : \vec{t}\} - x \mathbin{@} \alpha = \mathcal{V}''.R}$$

$$\textsc{Flex-Default}\ \frac{\mathcal{V}, \exists(\alpha : \kappa = t), \mathcal{V}'.P * x \mathbin{@} t - x \mathbin{@} \alpha = \mathcal{V}''.R}{\mathcal{V}, \exists(\alpha : \kappa), \mathcal{V}'.P * x \mathbin{@} t - x \mathbin{@} \alpha = \mathcal{V}''.R}$$

Figure 7. Instantiation of flexible variables

Rules TUPLE and CONSTRUCTOR descend into structural permissions. The remainders are *chained*, ensuring that a permission consumed in the i-th subtraction is no longer available in subsequent subtractions.

Rule APP is somehow complex; let us review it. First of all, we compare the parameters of the type applications according to the standard notion of *variance*. A type can be either *covariant* or *contravariant* in is i-th parameter. A type that is neither is *invariant*. A type that is both is *bivariant*, meaning it makes no use of its parameter. The $\overset{v_i}{\leftrightarrows}$ symbol allows the rule to pick the appropriate operation from Figure 6 according to v_i, the variance of the i-th parameter.

APP takes the duplicable restriction of the current permission P before comparing the arguments of the type application. This is crucial, as performing:

$$\mathcal{V}.y \mathbin{@} \mathsf{ref\ int} * x \mathbin{@} \mathsf{list}\ (=y) - x \mathbin{@} \mathsf{list}\ (\mathsf{ref\ int})$$

should be forbidden. The list may have length two, meaning we would need *two* copies of $y \mathbin{@} \mathsf{ref\ int}$, which is not duplicable!

As a first approximation, let us then write the premise as:

$$\mathcal{V}.\overline{P} * y \mathbin{@} u_i \overset{v_i}{\leftrightarrows} y \mathbin{@} u_i' = \mathcal{V}'.P'$$

The resulting P' should be discarded. Indeed, performing

$$\mathcal{V}.x \mathbin{@} \mathsf{list}\ (a \mid Q) - x \mathbin{@} \mathsf{list}\ a$$

should *not* give Q as a remainder. Indeed, the type $(a \mid Q)$ may not be available, as x may be Nil; therefore, Q may be not available at all!

This is still not precise enough. If α is a flexible variable, our approximation allows the following subtraction to succeed.

$$\forall(x : \mathsf{term}), \exists\alpha.x \mathbin{@} t\ \alpha\ \alpha - x \mathbin{@} t\ \mathsf{int}\ ()$$

Indeed, the approximation does not carry the flexible variable instantiations from one premise to another. This is the reason why we need to number our prefixes, and use `restrict` to import the flexible variable instantiations from one prefix onto the next.

ARROW is perhaps the most powerful rule. It first starts by taking the restriction of P to its duplicable bits. A closure in *Mezzo* can only capture *duplicable* variables (function types are duplicable); the corresponding function comparison rule behaves the same way. It then subtracts the domains; this is a contravariant position, meaning the order of the subtraction is changed. Then, it uses the remainder to compare codomains. Any information contained in t_1' is then carried over, which amounts to performing an η-expansion. A complete example that highlights the η-expansion rule is available in §3.8.

The resulting environment is then discarded, and the flexible variable instantiations are retained.

3.5 Implementation: introducing quantifiers

The order in which we introduce quantifiers in a prefix $\mathcal{V}$ is of particular importance. Consider, for instance, the subtraction from §3.3. We luckily chose the right order for introducing quantifiers: the $\forall a.\exists b$ prefix allows picking $b = a$ via INSTANTIATE, which solves the subtraction. If we were, however, to introduce b first, the prefix would be $\exists b.\forall a$, meaning that the instantiation could not succeed. Indeed, the two quantifiers cannot commute, meaning that the freshness premise from INSTANTIATE is not satisfied.

Quantifier introduction via normalization We thus need to *eagerly introduce universal (rigid) quantifiers*. This is the *raison d'être* of the normalization. By hoisting all possible existential quantifiers out of our current hypothesis P, we are able to introduce them as universal variables in the prefix $\mathcal{V}$ (§2.3).

One may wonder why normalization introduces singleton types in the domain of arrows. Let us consider the following subtraction problem, which we already mentioned in §2.1. This subtraction should succeed (we omit the `term` kind annotation, for clarity); however, performing it naïvely leads to a failure.

$$\dfrac{\dfrac{\dfrac{\text{no proof}}{\forall f, \exists x, \forall y.\big(y = x * y \mathbin{@} a - y \mathbin{@} a\big)}}{\forall f, \exists x.\big(f \mathbin{@} (=x \mid x \mathbin{@} a) \to () - f \mathbin{@} a \to ()\big)}\ \textsc{Function}}{\forall f.\big(f \mathbin{@} \forall x.(=x \mid x \mathbin{@} a) \to () - f \mathbin{@} a \to ()\big)}\ \textsc{Forall-L}$$

Indeed, finding an x such that for any y, we have $x = y$ is impossible, however powerful the type-checker may be. The y quantifier was introduced *too late*. We have to perform extra work in order to introduce y earlier; this can be achieved by introducing a singleton type in the arrow's domain. The type-checker properly takes care of it, using a combination of DECOMPOSE, COARROW and COMMUTEARROW. The example above can thus be type-checked, as shown in Figure 8.

Quantifier introduction using rules Normalization is not enough, because it only extrudes *existential* quantifiers. Rule EXISTS-R (resp. FORALL-L) introduces a flexible variable. At this stage, we should introduce all possible rigid variables. Failing to do that would lose instantiation options for the flexible variable. The type-checker thus aggressively searches for rigid variables, that is, existential quantifiers on the left-hand side (resp. universal quantifiers on the right-hand side), before appending an existential variable to the prefix.

Exploration Eagerly introducing all possible rigid variables before introducing an existential seems to be enough to ensure we never make mistakes about the order of our quantifiers. This is not the case. Consider the following subtraction:

$$(\exists\alpha, \forall\beta.(\alpha, \beta)) - (\exists\alpha', \forall\beta'.(\alpha', \beta'))$$

At first, the type-checker chooses to apply EXISTS-L to make sure the rigid variable α is introduced first. The type-checker is then confronted with a choice between FORALL-L and EXISTS-R which both introduce a flexible variable. The former choice leads to the prefix $\forall\alpha\exists\beta\exists\alpha'\forall\beta'$ and, ultimately, a failure, while the latter leads to the prefix $\forall\alpha\exists\alpha'\forall\beta\exists\beta'$, which admits a trivial solution. The type-checker has no way to figure out in advance which choice is the right one, and tries both solutions.

$$\textsc{Subsumption}\ \frac{\vdots}{\forall f.\big(f \mathbin{@} \forall x.(=x \mid x \mathbin{@} a) \to () - f \mathbin{@} \forall y.(=y \mid y \mathbin{@} a) \to ()\big)} \qquad \frac{\dfrac{\dfrac{}{a \equiv \exists x.(=x \mid x \mathbin{@} a)}\ \textsc{Decompose}}{a \to () \equiv (\exists x.(=x \mid x \mathbin{@} a)) \to ()}\ \textsc{CoArrow}}{a \to () \equiv \forall x.(=x \mid x \mathbin{@} a) \to ()}\ \textsc{CommuteArrow}$$
$$\frac{}{\forall f.\big(f \mathbin{@} \forall x.(=x \mid x \mathbin{@} a) \to () - f \mathbin{@} a \to ()\big)}\ \textsc{Subsumption}$$

Figure 8. Singleton types need to be introduced in the domain of arrows

3.6 Implementation: picking rules to apply

When confronted with a subtraction $P - Q$, the type-checker breaks down Q into a conjunction of atomic permissions of the form $x \mathbin{@} t$ or p, where p is a type variable at kind perm. The latter can be easily subtracted (or fail to be) through VariablePerm. The former pose more difficulties. First of all, the type-checker is unable to subtract $x \mathbin{@} t$ if x is a flexible variable, meaning that a situation where all permissions have a flexible left-hand side is a failure. Assuming a certain $x \mathbin{@} t$ exists, where x is rigid, the type-checker is faced with a choice. Indeed, there may exist several permissions for x: for instance, $x \mathbin{@} =x$ is always available, and it may happen that multiple function types are available for the same variable x.

The type-checker will thus explore all possible solutions, for each permission $x \mathbin{@} t'$ present in P. This may seem expensive, but there is usually one "useful" permission for a given variable, which we try first, using a syntactic criterion; situations where multiple "useful" permissions are available happen, but pertain to the realm of guru *Mezzo* code.

Once a permission is focused and the subtraction is of the form $P' * x \mathbin{@} t' - x \mathbin{@} t$, the type-checker relies on syntactic criteria to pick which rule to apply. Rules such as Tuple, Constructor, App are triggered according to the shape of t. Some situations require the use of Subsumption. For instance, when confronted with a nominal type (e.g. list a) and a concrete type (e.g. Cons), the rule is used in conjunction with Fold, as follows:

$$\frac{\dfrac{\dfrac{}{\mathsf{Nil} \le \mathsf{list}\ a}\ \textsc{Fold} \qquad \dfrac{}{\forall (y : \mathsf{term}).y \mathbin{@} \mathsf{Nil} - y \mathbin{@} \mathsf{Nil}}\ \textsc{Constructor}}{\forall (y : \mathsf{term}).y \mathbin{@} \mathsf{Nil} - y \mathbin{@} \mathsf{list}\ a}\ \textsc{Subsumption}$$

The Subsumption rule also allows one to save a copy of a permission at any time, using Duplicate. In practice, and if possible, we save a copy of the permission when focusing on it.

3.7 Implementation: instantiation choices

The way Instantiate is formulated leaves plenty of choice for the instantiation of a flexible variable. An implementation should thus have a strategy for instantiating flexible variables. The solution space is very large, and our algorithm explores only a subset of it: it is in no way complete. The behavior of our algorithm is governed by the rules found in Figure 7.

Rules Flex-Tuple-L and Flex-Constructor-L express the fact that a flexible variable should never directly instantiate to a tuple type or a constructor type. This ensures that a subtraction such as:

$$\exists \alpha.x \mathbin{@} \mathsf{int} * y \mathbin{@} \mathsf{int} * (=x, =y) - \alpha$$

always contains $\alpha = (\mathsf{int}, \mathsf{int})$ as a solution. Lacking these two rules, the only solution considered by the algorithm would be $\alpha = (=x, =y)$ which is often problematic (§4). These rules are prioritized over Flex-Default.

There are a variety of other situations where the algorithm applies a specific strategy for instantiating flexible variables. We do not detail these, as the discussion is quite technical, and the strategy likely to change in the future.

3.8 A complete subtraction example

We mentioned earlier (§3.2) that the type-checker is able to perform:

$$(\forall (y : \mathsf{term}).=y \to =y) - (\forall a.a \to a) = \mathsf{empty}$$

A perplexed *Mezzo* user may want to write the following η-expansion to convince themselves of the fact (comments denote the current permission):

```
fun (f: [y] (=y -> =y)): [a] (a -> a) =
  fun [a] (y: a): a =
    (* f @ [y] (=y -> =y) * y @ a *)
    f y
    (* ret @ =y * y @ a *)
```

In essence, a singleton type does not carry any useful information; it merely reveals the existence of a variable, by giving a pointer to it, and does not carry any ownership of the heap. Therefore, a function that takes a pointer and returns it, without the ability to do anything with it, necessarily preserves its argument untouched.

We provide a complete derivation for this subtraction in Figure 9. The key steps are highlighted in red. The original subtraction problem is at the bottom. Moving up a few lines, we have performed normalization, by introducing a singleton type in the domain of the function on the right. We have also introduced quantifiers in the right order. We are left with two functions to compare. The rule for functions first compares domains, which gives us $x \mathbin{@} a$ as a remainder. This remainder is carried onto the comparison of codomains, which uses this very permission to succeed, giving the empty result.

As a side-note, proving the converse subtraction below is trivial, as it is just a matter of instantiating a with $=y$.

$$(\forall a.a \to a) - (\forall (y : \mathsf{term}).=y \to =y) = \mathsf{empty}$$

4. The merge operation

When introducing the language informally (§1.2), we mentioned that in order to type-check the `match` construct, the type-checker is confronted with the following permission, from the `Nil` branch:

```
l @ Nil * ret = l
```

and the following permission, from the `Cons` branch:

```
l @ Cons { head = h; tail = t } *
h' @ b * t' @ list b *
ret @ Cons { head = h'; tail = t' }
```

We went on explaining that the type-checker "magically" knew that it should seek to obtain "`ret @ list b`", a permission that is valid in both branches. Solving this problem in general is called the *merge operation*.

Figure 9. A complete type-checking example

The merge operation is what allows one to effectively type-check a `match` or an `if-then-else` expression. As *Mezzo* positions itself as a type-checker, not a static analyzer or a program prover, it is compulsory that disjunctions in control-flow be solved on the spot. Indeed, a type system shall be predictable; having errors that mention a particular path in the control-flow is no intuitive behavior, and postponing errors until the end of a function body is not acceptable either.

4.1 Why is merging difficult?

The merge problem does not admit, in general, a principal solution. Let us see a few examples.

First example We consider a first example where both branches of an `if-then-else` expression return a tuple. Please note that `t` is exclusive, meaning only one copy of "`x @ T`" is available.

```
data mutable t = T

val z =
  if ... then begin
    let x = T in
    (x, x)
  end else begin
    (T, T)
  end
```

The merge above can be solved by either "`z @ (T, unknown)`" or "`z @ (unknown, T)`", where `unknown` denotes the $\top$ type, i.e. the supertype of all types. Lacking any other annotation, the type-checker has no way of deciding which solution should be preferred over the other, as they are incomparable. It is to be remarked that if the type `t` were to be duplicable, "`z @ (T, T)`" would be a principal solution.

Second example Here is a slightly more complex example, where sharing occurs:

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

This time, the type-checker can pick several solutions:

- picking "`x @ T * z @ (=u, =u)`" will preserve the permission for `x` while still retaining the fact the two components of the tuple are pointers to the same block in the heap (`u` is existentially quantified);
- picking "`x @ unknown * z @ (=u, =u) * u @ T`" loses the permission for `x` in favor of a "better" one for `z`;
- other, inferior solutions exist, such as "`z @ (T, unknown)`" or "`z @ (unknown, T)`", which would fail to preserve sharing information.

The first two solutions are incomparable, as none of them implies the other. The last two solutions, however, are derivable from the second one. Again, if `t` were to be duplicable, this situation would admit a principal solution, namely "`x @ T * z @ (=u, =u) * u @ T`".

We conjecture that a principal solution always exists (and is always found by our algorithm) if the merge problem only contains duplicable data, and does not contain any data type with more than one branch.

Third example Merging is even more difficult when nominal types come into play.

```
val z =
  let y = ... in
  if ... then
    Some { contents = y }
  else
    None
```

Let us consider the case where `y` refers to duplicable data, and assume for the sake of example that we have "`y @ int`". This merge problem admits a principal solution, namely "`z @ option (=y) * y @ int`". One can derive "`z @ option int`" using CopyDup.

In the case that `y` refers to non-duplicable data, e.g. if we have "`y @ ref int`", then again, this problem does not admit a principal solution, as "`z @ option (=y) * y @ ref int`" and "`z @ option (ref int)`" are not comparable[5].

As a side-note, if `y` were bound in the `then` branch, the solutions would be similar, except `y` would be existentially-quantified in the resulting permission.

Fourth example Lists pose different difficulties.

```
val z =
  let y = ... in
  if ... then
    Cons { head = y; tail = Nil }
  else
    Nil
```

Here, merging into "`z @ list (=y) * y @ ...`" would be a terrible choice, as the user could only append `y` to the list afterwards!

4.1.1 Three sub-problems

The merge operation poses several difficulties.

- We need to reconcile conflicting heap shapes into a common description. This implies traversing two graphs in parallel, which is similar to what has been done in the field of shape analysis [12]. This problem is predictably and efficiently treated by our algorithm.
- Because of mutable data, the algorithm has to decide, as in the first example, where to "assign" a piece of mutable data. We call this the *exclusive resource allocation problem*. Our algorithm can detect and warn the user about such cases, but lacking any backward analysis, is unable to decide which solution it should pick. Our first two examples would trigger such a warning.
- Folding inductive predicates is also a recurring problem in the literature; is is highlighted by the third and fourth examples. In *Mezzo*, the use of singleton types poses additional difficulties, as it offers multiple solutions for the parameter α when folding two types into, say, list α. Our algorithm is not complete and only considers a limited, yet predictable, subset of choices for the parameter.

4.2 Formal definition of the merge operation

We define a *root* to be a program variable defined before the merge operation, or the return value of the `match` expression. In the fourth example, `y` is a root, as it is bound before the disjunction; `z` is also a root, as it represents the value of the `if-then-else` expression.

We talk about the *original* permission (before the disjunction in control-flow), the *left* and *right* permissions, and the *destination* permission (the result of the merge operation).

[5] The system would need a rule for an η-expansion of data types. We currently do not have this, and do not plan to add it.

We define a *point* to be any location in the heap accessible from a root. There are left points, right points, and destination points.

We assume a *bijection* ψ where $\psi(x) = x_l, x_r$ means that the left point x_l and the right point x_r map onto the destination point x. Initially, $\psi(\rho) = \rho_l, \rho_r$, meaning that the return values from the left and right expressions map onto the return value for the whole `match` expression. We also have $\psi(x) = x, x$ for all other roots, that is, variables that were already defined in the *original* environment.

We define the merge of prefixed, *normalized* environments P_l and P_r to be:

$$(\mathcal{V}_l.P_l \vee V_r.P_r) = \mathcal{V}.R$$

A more useful presentation is one that, just like subtraction, focuses on a particular variable, and represents the working state of the algorithm using R, which stands for permissions that have been merged already.

$$\frac{\psi(x) = x_l, x_r}{(\mathcal{V}_l.P_l * x_l \mathbin{@} t \vee V_r.P_r * x_r \mathbin{@} t) * \mathcal{V}.R = (\mathcal{V}_l.P_l \vee V_r.P_r) * \mathcal{V}.R * x \mathbin{@} t}$$

The difficulties lie, first, in finding the correct ψ function and second, in applying the right subsumption rules on P_l and P_r so as to obtain the same type on both sides of the disjunction.

4.3 Computing the mapping

Computing the ψ function is done by traversing the left and right permission graphs in parallel. One basically follows the structural permissions (tuples and constructors) starting from the roots, and associates a pair of matching points onto a destination point. We ensure that for a pair of matching x_l and x_r, only one destination x corresponds: ψ is really a bijection, meaning that it *preserves sharing*.

The ψ function, once completed, maps every pair of left and right points, reachable through the same path, onto a destination point.

The rules that implement the building of the ψ function are presented in Figure 10. They assume the environments to be normalized, since building the ψ function is only possible if structural types are *expanded* using singleton types.

4.4 Algorithmic presentation of merge

We assume the ψ function has been computed, and initially start with:

$$(\mathcal{V}_l.P_l \vee V_r.P_r) * \mathcal{V}.\mathsf{empty}$$

If $\mathcal{V}_0$ is the prefix of the *original* permission, and if $\mathsf{dom}(\psi)$ is the domain of the ψ function, we define $\mathcal{V}$ to be:

$$\mathcal{V} = \mathcal{V}_0, \exists(\overrightarrow{\mathsf{dom}(\psi)} : \mathsf{term})$$

This is restrictive: a variable that is *local* to the left or right permission, and that has not been mapped by the ψ function, cannot appear in the destination permission. For instance, no meaningful permission will come out of the following merge (`{t} t` is the concrete syntax for $\exists t.t$):

```
(* f @ () -> {t} t *)
val z =
  if ... then
    f ()
  else
    f ()
```

Function `f` returns an existentially-quantified type. Per the restriction above, the algorithm will not know how to merge this. The user can solve the merge by providing a type annotation for `z`.

PSI-TUPLE

$$\frac{\psi(x) = x_l, x_r \qquad x_l \mathbin{@} (\ldots, {=}y_l, \ldots) \qquad \nexists y'.\psi(y') = y_l, y_r \qquad x_r \mathbin{@} (\ldots, {=}y_r, \ldots) \qquad y \text{ fresh}}{\psi(y) = y_l, y_r}$$

PSI-CONSTRUCTOR

$$\frac{\psi(x) = x_l, x_r \qquad x_l \mathbin{@} \mathsf{A}\,\{\ldots, f : {=}y_l, \ldots\} \qquad \nexists y'.\psi(y') = y_l, y_r \qquad x_r \mathbin{@} \mathsf{A}\,\{\ldots, f : {=}y_r, \ldots\} \qquad y \text{ fresh}}{\psi(y) = y_l, y_r}$$

Figure 10. Computing the ψ function

MERGE-TUPLE

$$\frac{\psi(x) = x_l, x_r \qquad \psi(\vec{y}) = \vec{y_l}, \vec{y_r}}{(\mathcal{V}_l.P_l * x_l \mathbin{@} (=\vec{y_l}) \lor \mathcal{V}_r.P_r * x_r \mathbin{@} (=\vec{y_r})) * \mathcal{V}.R = (\mathcal{V}_l.P_l \lor V_r.P_r) * \mathcal{V}.R * x \mathbin{@} (=\vec{y})}$$

MERGE-CONSTRUCTOR

$$\frac{\psi(x) = x_l, x_r \qquad \psi(\vec{y}) = \vec{y_l}, \vec{y_r}}{(\mathcal{V}_l.P_l * x_l \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y_l}\} \lor \mathcal{V}_r.P_r * x_r \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y_r}\}) * \mathcal{V}.R = (\mathcal{V}_l.P_l \lor V_r.P_r) * \mathcal{V}.R * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y}\}}$$

MERGE-APP

$$\frac{\begin{array}{c}\psi(x) = x_l, x_r \\ (\mathcal{V}_l, \exists(y : \mathsf{term}).\overline{P_l} * y \mathbin{@} u_{i,l} \lor \mathcal{V}_r, \exists(y : \mathsf{term}).\overline{P_r} * y \mathbin{@} u_{i,r}) = (\mathcal{V}_l.\overline{P_l} \lor V_r.\overline{P_r}) * \mathcal{V}.y \mathbin{@} u \quad \text{if } t \text{ is covariant in parameter } i \\ u = \forall\alpha.\alpha \quad \text{if } t \text{ is bivariant in parameter } i \\ u_l \in \mathcal{V}_0 \qquad u_r \in \mathcal{V}_0 \qquad u_l = u_r \quad \text{otherwise}\end{array}}{(\mathcal{V}_l.P_l * t\ \vec{u_l} \lor V_r.P_r * t\ \vec{u_r}) * \mathcal{V}.R = (\mathcal{V}_l.P_l \lor V_r.P_r) * \mathcal{V}.R * t\ \vec{u}}$$

MERGE-PERM-VAR

$$\frac{p \in \mathcal{V}_0}{(\mathcal{V}_l.P_l * p \lor V_r.P_r * p) * \mathcal{V}.R = (\mathcal{V}_l.P_l \lor V_r.P_r) * \mathcal{V}.R * p}$$

MERGE-DEFAULT

$$\frac{t \in \mathcal{V}_0 \qquad \psi(x) = x_l, x_r}{(\mathcal{V}_l.P_l * x_l \mathbin{@} t \lor V_r.P_r * x_r \mathbin{@} t) * \mathcal{V}.R = (\mathcal{V}_l.P_l \lor V_r.P_r) * \mathcal{V}.R * x \mathbin{@} t}$$

Figure 11. The merge operation

The reason for this restriction is that merging local types requires packing them under an existential; the results are not predictable, and we risk merging too many types. Indeed, with this strategy, "$x \mathbin{@} \mathsf{int} \lor x \mathbin{@} ()$" would be solved as "$x \mathbin{@} \exists t.t$"!

The rules are presented in Figure 11. We write $t \in \mathcal{V}_0$ to signify that a type makes sense in the left, right, and destination permissions (thus rendering its merge trivial). Tuple and constructor types are translated using the ψ function (MERGE-TUPLE, MERGE-CONSTRUCTOR). Function types, type variables are merged using the default criterion, that is, syntactic equality (MERGE-DEFAULT). MERGE-APP deserves an explanation. We can only merge type applications of the same type. In that case, we recursively merge the parameters of the type application.

- If parameter i is bivariant, the type application will not use its parameter, meaning we can pick $\bot = \forall\alpha.\alpha$ for the resulting u_i.
- If parameter i is covariant, we can recursively merge the type parameters; we bind a fresh rigid variable y and re-use the merge algorithm to compute parameter u_i.
- If parameter i is invariant, we rely on a syntactic equality criterion to compute parameter u_i.
- If parameter i is contravariant, we would need to perform the *intersection* of two types, an operation that is not supported by our type-checker. We approximate, and use syntactic equality again.

4.5 Implementation strategy

Again, these rules provide very little in terms of implementation details. With the contents of Figure 11, we are still at loss for resolving the merge from the third example. Indeed, one needs to apply *subsumption rules* before the rule MERGE-APP can apply.

Difficulties related to merging only appear when merging distinct constructors (e.g. Nil *vs.* Cons), or a constructor with a nominal type (e.g. Cons *vs.* list t). Let us start with the latter, easier case.

Cons.* vs. *nominal We are faced with the following disjunction, where A is a data constructor that belongs to type t (the problem has no solution otherwise).

$$\mathcal{V}_l.P_l * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y}\} \lor \mathcal{V}_r.P_r * x \mathbin{@} t\ \vec{u}$$

We could be subtle here, but for the sake of clarity, we require $\vec{u} \in \mathcal{V}_0$, which implies that all the type application's parameters also make sense in the *left* environment.

We can thus perform, in the left environment, the following subtraction:

$$\mathcal{V}_l.P_l * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y}\} - x \mathbin{@} t\ \vec{u} = \mathcal{V}'_l.P'_l$$

The left permission is weakened, but we gain in exchange a new merge problem, which is solvable using rule MERGE-APP.

$$\mathcal{V}'_l.P'_l * x \mathbin{@} t\ \vec{u} \lor \mathcal{V}_r.P_r * x \mathbin{@} t\ \vec{u}$$

Please note that we *keep* the resulting environment $\mathcal{V}'_l.P'_l$: the subtraction may have consumed permissions and instantiated flexible variables in the left permission, and we need to keep this information for future merges.

Cons.* vs. *cons We are now faced with the following disjunction, where both A and B are data constructors belonging to a common type t.

$$\mathcal{V}_l.P_l * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y_l}\} \lor \mathcal{V}_r.P_r * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y_r}\}$$

Again, subtraction is going to help us. We know that both these types should fold into "$t\ \vec{u}$". The difficulty is we don't know the actual values of $\vec{u}$. We can perform the following subtraction:

$$\mathcal{V}_l, \exists(\vec{\alpha_l} : \vec{\kappa}).P_l * x \mathbin{@} \mathsf{A}\,\{\vec{f} : {=}\vec{y_l}\} - x \mathbin{@} t\ \vec{\alpha_l} = \mathcal{V}'_l.P'_l$$

We introduce flexible variables along with the right kinds to stand for the parameters of the type application. The same operation is

performed on the right-hand side, leading to the following subtraction, which is solvable using MERGE-APP:

$$\mathcal{V}'_l.P'_l * x \mathbin{@} t\ \vec{\alpha_l} \vee \mathcal{V}'_r.P'_r * x \mathbin{@} t\ \vec{\alpha_r}$$

If instantiated, the type-checker automatically substitutes a flexible variable α for its instantiation t. In the case where either one of α_l and α_r is uninstantiated, the type-checker will perform the intuitive instantiation. In the case where both remain flexible, a type annotation is mandatory.

Instantiation choices The quality of the frequent *Cons* vs. *cons* case depends on the choice made by the type-checker for the flexible variables $\vec{\alpha}$. A heuristic is used, which the user can always bypass using a type annotation. It mainly consists in never instantiating a flexible variable with a type that contains singletons, for fear the singletons would refer to a local variable, thus rendering the instantiation choice useless for a merge.

4.6 Detecting exclusive resource allocation conflicts

The exclusive resource allocation conflicts from the first and second examples can be detected easily. When one merges $y_l \mathbin{@} t \vee y_r \mathbin{@} t$ onto $\psi(y) = y_l, y_r$, if t is exclusive, and there exists y' such that $\psi(y') = y_l, y'_r$ (resp. $\psi(y') = y'_l, y_r$), this means that there will be a conflict as to which of y and y' "gets" the exclusive permission from y_l (resp. y_r).

5. Related work

The entire design of Mezzo owns a great deal to separation logic [11]; our type system can be seen as an adaptation of separation logic into a type system for an ML-like language. This is achieved through the use of permissions: a permission *is* analogous to a separation logic assertion. This design choice has numerous consequences.

- Type annotations are mandatory for functions, meaning that the types of the arguments *are* the pre-condition, while the return type of the function is the post-condition. We do not need a separate language for pre- and post-conditions.
- We thus do not distinguish between spatial predicates and logical formulae: a permission embodies both concepts. The permission "`x @ list a`" has a spatial reading, while a permission "`x @ =y`" has a logical reading.
- Algebraic (inductive) data types are central in ML-like languages, and Mezzo makes no exception. Instead of defining separate hard-wired predicates for lists, trees, doubly-linked-lists, Mezzo naturally relies on user-defined data types (such as `list`) for the spatial assertions (such as "`x @ list a`").
- Disjunction is somehow easier to treat. For a pointer language, reasoning of the form $x = \mathsf{null} \vee x \mapsto \mathsf{list}$ takes place, which requires exploring disjunctions and manipulating predicates for *list segments*. Algebraic data types are *tagged* sums, meaning that disjunctions are destructed explicitly via `match` expressions, which make the job of the type-checker easier. Moreover, `while` loops are replaced with *recursive* functions, that express different pre- and post-conditions. Combined with recursive reasoning [9], this alleviates the need for list segments.

Frame inference Calcagno *et al.* perform a symbolic execution of pointer programs with Smallfoot [2, 3]. They introduce a decidable algorithm for inferring *frame axioms*, that is, portions of the current separation logic assertion that are left untouched by a call to a subroutine. They assume unquantified assertions, and only consider a subset of hard-wired inductive predicates.

Nguyen *et al.* [8] allow user-defined inductive predicates. They present an algorithm that, even if not complete, terminates. They allow a limited form of quantification.

Pérez and Rybalchenko [7] have a correct and complete algorithm for a fragment of separation logic with list segments. They deal with linear formulas that contain exactly one spatial predicate, and treat disjunction.

In the setting of Mezzo, the frame inference problem requires dealing with duplicable and non-duplicable parts of the heap, as well as quantifiers. To the best of our knowledge, this extends previous work in the area.

Computing joins What we call the merge problem received comparatively little attention from the program proof and static analysis communities [12, 13, 5]. There are several reasons for this. Indeed, computing the merge, or "join" of two branches amounts to performing an approximation. There are valid reasons not to do that, such as preserving the quality of the solutions, or avoiding the complexity of computing an approximation. In this case, each branch is treated separately, which may indeed lead to an exponential number of branches when one reaches the end of control-flow.

The need for a "join" operation arises in a different context for pointer languages. In pointer languages, loops are central. Computing the shape of a loop requires "joining" shapes that denote successive iterations of the loop. The procedure ultimately reaches a fixed point which then stands for the loop invariant. We do not have this issue as loops in Mezzo are recursive functions, meaning they are annotated.

Proving our type-checker A version of the Smallfoot program analyzer has been proven sound using the Coq proof assistant [1]. We currently do not have any soundness or completeness result for our type-checking algorithms.

6. Conclusion

We have presented the implementation of a type-checker for the Mezzo language. Mezzo is based on permissions and features singleton types, constructor types, first-class functions, which make implementing a type-checker difficult. Type-checking a Mezzo program leverages two central operations, *subtraction* and *merge*. We have shown how to perform both operations in the context of Mezzo, extending previous work in the area.

In the future, we would like to obtain a completeness result for our algorithm, even if on a limited subset of Mezzo programs. We conjecture that our type-checker is complete when the only quantifiers are at kind `term`. We would also like to improve inference, as our algorithm only explores a subset of the solution space.

References

[1] Andrew W. Appel. VeriSmall: Verified Smallfoot shape analysis. volume 7086 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2011.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[4] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)*, pages 289–300, 2009.

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

[6] Dino Distefano and Matthew J Parkinson J. jstar: Towards practical verification for java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

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

[8] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.

[9] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *International Conference on Functional Programming (ICFP)*, 2013. To appear.

[10] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[12] Xavier Rival. Abstract domains for the analysis of programs manipulating complex data-structures (habilitation thesis), 2011.

[13] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.