Contents

1 Getting and using Mezzo

1.1 Installation

Mezzo is only available in source form; there are no pre-compiled binaries available yet. The source code of Mezzo is available online.

Mezzo requires a few ocamlfind packages in order to work properly. We recommend using opam. Please make sure the following packages are installed: `yojson`, `ulex`, `functory`, `pprint`, `fix`, `menhir`.

Once the prerequisites are complete, download the latest snapshot of Mezzo, and extract it.

```
$ tar xjvf mezzo-latest.tar.bz2
$ cd mezzo-*
```

Then, compile Mezzo with:

```
$ make
```

To make sure that Mezzo has been compiled properly, you can run:

```
$ make test
```

For the moment, there is no way to install Mezzo into a system-wide location. In the meanwhile, you can add the Mezzo directory to your `PATH` environment variable.

1.2 The `mezzo` binary

1.2.1 Type-checking a program

Mezzo files end with `.mz`. Open your favorite editor, and edit `ex1.mz`. Put the following contents into the file:

```
val x = 1

val _ =
  print x
```

Next, assuming the `mezzo` binary is available in your `PATH`, run `mezzo ex1.mz`. The output is as follows:

```
val x @ int::int
```

The syntax of Mezzo is reminiscent of ML, which we assume the reader is familiar with. We will introduce syntax on-the-fly, presenting the language constructs as we make progress through this tutorial. Top-level values are introduced using the `val` keyword; intermediate definitions are introduced using the usual `let` keyword.

By default, `mezzo` type-checks the program, then prints the type of the top-level values. Here, `mezzo` judiciously type-checked `x` as being an integer. More precisely, `x` has the `int` type, from the `int` module.

In Mezzo, names are qualified using `::`. By writing `m::x`, you refer to the name `x` from module `m`. The `int` module is bundled with Mezzo; it is found in the `corelib` directory, and exports the `int` type as well as arithmetic operators.

1.2.2 Interpreting a program

Mezzo programs can be interpreted, using the built-in interpreter. Interpreting a program is as simple as passing a `-i` argument ("interpret") to the `mezzo` binary.

```
$ mezzo -i ex1.mz
```

The output will be:

```
1
```

1.2.3 Compiling a program

Compiling a Mezzo program is done by "translating" it into an OCaml program[1].

Programs translated from Mezzo to OCaml rely on a runtime support library. It can be installed by running the following set of commands. These commands will build and install an `ocamlfind` package called `mezzolib`.

```
$ cd mezzolib
$ make && make install
```

Mezzo comes with a set of pre-defined modules for arrays, atomic variables, references, etc. called the "core library". We should translate these first, so that further translated programs can reference and use the functions exported by the core library. This can be achieved via the following set of commands, which will install a new `ocamlfind` package called `mezzocorelib`, which packages together all the core modules into a big OCaml library.

```
$ cd corelib
$ make && make install
```

Back to our original example, we must first translate it. This is done by passing the `-c` argument ("compile") to the `mezzo` binary.

```
$ mezzo -c ex1.mz
```

This creates `mzex1.ml`, the translated version of `ex1.mz`. We now need to compile `mzex1.ml`.

[1] Since the type-system of Mezzo is more powerful than that of OCaml, some programs that Mezzo "understands" will not be "understood" by OCaml. In other words, a Mezzo program will generally not type-check in OCaml. We must therefore bypass the OCaml type-checker: this is done by emitting code that uses `Obj`, the unsafe module of OCaml, which allows one to manipulate the internal representation of objects.

```
$ ocamlbuild -use-ocamlfind -package mezzolib -package mezzocorelib mzex1.byte
```

Barring any errors, the resulting program can be launched.

```
$ ./mzex1.byte
```

Unsurprisingly, the output is:

```
1
```

1.2.4 Using ocamlbuild with .mz files

TODO

- create a big ocamlfind package with: the core library, the standard library, the runtime library, our ocamlbuild plugin (compiler-libs or even lablgtk2 do that)
- create a target install in mezzo's `Makefile`
- explain how to leverage the new -plugin-tag feature of the latest ocamlbuild to allow the user to just write, in their `myocamlbuild.ml`

```
MezzoOcamlbuildPlugin.init ()
```

- and while we're at it, provide a sample project with a ready-made `Makefile` and `myocamlbuild.ml`

1.2.5 Interfacing OCaml with Mezzo

TODO

- explain how one can create a mixed project with OCamlbuild
- put a forward pointer to the section on interaction between OCaml and Mezzo, where we will explain the kind of restrictions we currently have, also explain how to use MezzoLib to convert between a `MezzoLib.t` and an OCaml type.

2 The philosophy of Mezzo

The type system is central in the design of Mezzo. It is novel, sometimes complex, and certainly does require some effort to fully understand. We thus devote a section to the type system itself, before showing how to write programs in the language.

2.1 Founding principles: reasoning about state

Mezzo is a language that helps the programmer reason about *state*. State is a pervasive notion in computer programs. Consider the following pseudo-code:

```
int main () {
  int* x = new int;
  // x is a _valid_ pointer to an integer
  ...
  delete x;
  // x is an _invalid_ pointer that should not be used
}
```

The variable `x` changes *state*: it goes from the "valid pointer" state to the "invalid pointer" state. When `x` is "valid", it can be safely dereferenced. Once `x` is `delete`'d, one must no longer use it. A traditional type system, however, will just assert that `x` is a pointer to an integer: it is up to the programmer to make sure they only use `x` when it is valid. *The type system provides no help for reasoning about the state that* `x` *is in.*

What if we had a type system that allows one to reason about state? This seems simple enough. However, reasoning about state is tricky because of *aliasing*. Imagine we start thinking about state in a *naïve* fashion.

```
int main() {
  int* x = new int;
  // x has type "valid int*"
  ...
  int* y = x;
  // x and y have type "valid int*"
  ...
  delete x;
  // x has type "invalid int*", y has type "valid int*"
  ...
  delete y;
  // crash!
}
```

The type system sees two different *names*, `x` and `y`, and gives each one of them a *type*. Even if we were to track the state change of `x`, because `y` is actually a *synonym* for `x`, the call to `delete y` would still crash the program. We say that `x` and `y` are *aliases*.

Aliasing is closely related to the problem of ownership. If one part of the program sees `x` as a "valid integer", then tracking state changes for `x` only makes sense if no other part of the program sees `x`. Thus, we should be able to say that we "*own*" `x`.

Mezzo blends in several concepts from the literature; it incorporates mechanisms that speak about *aliasing* and *ownership*, so as to obtain a type system that is able to deal with *state*.

In this section, we used examples in a C-like language. Mezzo is a language that draws inspiration from ML; it has first-class functions, pattern matching, and a garbage-collector. The syntax of Mezzo is thus very similar to that of ML, and we will forget about C-like snippets in the remainder of this tutorial.

2.2 Turning this into a type system: an informal presentation of permissions

We introduce the foundational notion of Mezzo, permissions, through an informal presentation of the type system. We anticipate on the following section, which will exhaustively introduce all the constructs in the type system of Mezzo.

2.2.1 My first permission

In a traditional, strongly-typed programming language, such as ML, one may say that "`x` has type `ref int`", which is the type of integer references. This implies, in particular, that the type of `x` is valid for the rest of `x`'s scope. Conversely, in Mezzo, `x` may "have type" `ref int`, but it may also have, later on, type `ref string`. Thus, we need to come up with a new way to talk about "the" type of a variable.

We talk about types using *permissions*. A permission is written `x @ t`, where `x` is a program variable, and `t` is a type. One may think of a permission as an access token that grants its owner the right to use `x` as a variable of type `t`. Permissions are transient: this permission may disappear, to be replaced by `x @ u` at a later program point.

Obtaining a fresh permission is easy. Writing `let x = 1 in ...` makes `x @ int` available in the continuation `...`.

The permission mechanism *is* our type system. It is pervasive, in the sense that at any program point, a *current* permission is available. When the program starts, the current permission is `empty`. As execution progresses, the current

permission evolves to become a *conjunction* of permissions, which we write `p * q`. Permissions do not exist at run-time.

The current set of permissions tells the programmer what they can do with variables. For the expression `x + y` to be well-typed, the conjunction `x @ int * y @ int` has to be available at that program point.

Earlier, we type-checked the following program:

```
val x = 1

val _ =
  print x
```

The permission `val x @ int::int` that the type-checker outputs is the conjunction that remains after program execution.

2.2.2 Permissions for controlling effects

In the case of function, permissions describe the *assumptions* that a function makes on its arguments, and the *guarantees* that it provides on its return value. In other words, permissions describe the pre- and post-conditions of functions. Let us consider the following function signature.

```
val length: [a, b] (x: list a) -> int
```

The type of the `length` function looks like the one from ML. It expresses a precise invariant, though: the function will *consume* a permission `x @ list a` from its caller. That is, if one wishes to call `length x`, then one must give up `x @ list a`. Conceptually, the `length` function will *use* that permission to access `x` as a list of elements, then, once it is done with it, the permission will be *returned* to the caller, along with a result of type `int`.

As a syntactic convention, and unless otherwise specified, permissions for the arguments of functions are understood to be taken *and* returned to the caller.

What happens, in practice, is the following, where comments denote the set of available permissions at a given program point.

```
(* x @ list a *)
let l = length x in
(* l @ int * x @ list a *)
...
```

Let us now consider a different function signature. The function called `mswap` swaps *in-place* the two components of a mutable pair (`mpair`).

```
val mswap: [a, b] (consumes x: mpair a b) -> (| x @ mpair b a)
```

The presence of the `consumes` keyword indicates the function takes a permission `x @ mpair a b` from the caller, and *does not return it*. What is returned is the *unit type* `()`, along with a permission `x @ mpair b a`. We write the conjunction of a type `t` and a permission `p` as `(t | P)`, but the example above uses syntactic sugar for the case where `t = ()`.

This signature expresses the fact that the function *changes the type of its argument*. What happens, from the caller's perspective, is as follows.

```
(* x @ mpair a b *)
mswap x;
(* x @ mpair b a *)
...
```

The variable `x` went from `mpair a b` to `mpair b a`. This is a *type-changing update*, which the permission mechanism accurately describes.

2.2.3 Permissions for tracking ownership

For the example above to be sound, no one else must own a copy of `x @ mpair a b`, since this information would be invalidated by the call to `mswap`. The type system of Mezzo guarantees that permissions that denote mutable data, such as `x @ mpair a b` are uniquely-owned. We say that these permissions are exclusive. In practice, this means that it is impossible to obtain another copy of `x @ mpair a b`.

Conversely, the `val x @ int::int` permission that we saw earlier denotes immutable, permanent knowledge. It is therefore safe to share that information with "others" (other parts of the program, other threads), and we say that this permission is duplicable. In other words, one can obtain as many copies of it as desired.

As we will see later on, both Mezzo and the user can always tell whether a permission is exclusive or duplicable. This is done by looking up the definition of `t` in `x @ t`.

We just saw read-write, uniquely-owned permissions ("exclusive") and read-only, shared permissions ("duplicable"). Permissions which are neither exclusive or duplicable are said to be affine. Affine permissions are a strict superset of exclusive and duplicable permissions.

2.2.4 Permissions for tracking aliasing

Permissions can express two kinds of aliasing information: *must-alias* constraints, and *must-not-alias* constraints.

Must-alias Consider the following code snippet:

```
val _ =
  let x = newref 3 in
  let y = x in
  assert y @ ref int
```

The first line gives rise to `x @ ref int`, as we saw earlier. The second line is more interesting. Instead of reasoning about which one of `x` and `y` gets to own the reference, we generate a permission that embodies the fact that x and y are synonyms: the permission embodies a must-alias constraint. This permission is `y @ (=x)`.

The type `=x` is a *singleton type*, a type whose only inhabitant is `x` itself. Saying that `y` "has type" `=x` amounts to saying that `x` and `y` are actually the same thing; that is, that they're aliases. We use syntactic sugar for that special form of permissions, and write `y = x`.

The interesting point is that we dont't have to assign ownership of the reference to either `x` or `y`: the system just records the aliasing information and knows that it can substitute `x` for `y` wherever needed. In particular, when the user asserts that `y` is a reference to an integer at line 3, Mezzo automatically figures out that it can rewrite `x @ ref int * y @ (=x)` into `y @ ref int * y @ (=x)`, thus satisfying the assertion.

Singleton types are also used in structural types. Whenever one obtains a permission for, say, a tuple, the type-checker automatically introduces names for the components. For instance, instead of having `x @ (ref int, ref int)`, the type-checker will introduce internal names `l` and `r`, and transform this into `x @ (=l, =r) * l @ ref int * r @ ref int`. This is, again, useful to reason about ownership. Consider the example below.

```
val _ =
  let x = newref 5, newref 7 in
  let y, z = x in
  assert y @ ref int * z @ ref int;
  assert x @ (ref int, ref int);
  assert x @ (unknown, ref int) * y @ ref int;
```

Since we have an *expanded form* for `x`, type-checking the let-binding merely amounts to adding `y = l * z = r` into the current conjunction: we don't have to reason about whether the references are owned by `x` or by `y` and `z` individually.

Must-not-alias Permissions also implicitly contain must-not-alias information. This stems from the unique owner policy. Having, for instance, a conjunction such as `y @ ref int * z @ ref int` tells you that `y` and `z` *must be distinct*; otherwise, it would violate the unique owner guarantee of Mezzo. This is a *must-not-alias* constraint. Similarly, having `y @ ref int * z @ int` also guarantees that `y` and `z` are distinct.

Not all conjunctions are informative: having, for instance, `y @ int * z @ int` doesn't tell anything. The two variables may or may not be aliases. This is due to the fact that `int` is a duplicable type.

```
val _ =
  let x = 11 in
  let y = x in
  assert x @ int * y @ int;
```

The assertion above would fail with a reference to an integer.

As a side note, some conjunctions embed a very strong meaning. A conjunction such as `x @ ref int * x @ ref int` violates the unique owner policy of Mezzo. This is impossible! Therefore, execution can never reach such a program point. The conjunction denotes *dead code*, that is, code that is never run.

3 A first example

Before giving a thorough presentation of the type system of Mezzo, we illustrate type-checking on two simple examples. The `map` function will be our running example: we first write a trivial implementation, and show how to perform type-checking in Mezzo. We then write a tail-recursive version of `map`, something which cannot be done in ML. We show how the type system of Mezzo allows us to type-check it.

This section anticipates a little bit on the syntax of both types an expressions. These will be described in the following sections.

The way one would write the `map` function in, say, OCaml, is as follows:

```
let rec map f l =
  match l with
  | hd :: tl ->
      f hd :: map f tl
  | [] ->
      []
```

3.1 Type-checking the classical map function

One can write the same simple function in Mezzo. Note the `consumes` annotation that leaves it up to the caller to duplicate the permission on the list, if possible.

```
open list

val rec map [a, b] (f: (consumes a -> b), consumes xs: list a): list b =
  match xs with
  | Cons { head = h; tail = t } ->
      let h' = f h in
      let t' = map (f, t) in
      Cons { head = h'; tail = t' }
  | Nil ->
      xs
  end
```

See the HTML demo

Let us see how this function is type-checked. In the snippet above, comments denote the currently available conjunction, as well as the reasoning steps that the type-checker performs. Comments omit the permission for `f` which remains available all throughout the body of `map`.

Initially, when execution enters the function body, `xs @ list a * f @ (consumes a) -> b` is available. Matching on `xs` *refines* the permission for `xs` into a *structural* one.

In the first branch, we learn that `xs` is a `Cons` cell: the permission for `xs` is updated in place to reflect this. Moreover, the type-checker introduces names for the fields of `xs`: it *expands* the structural permission. This is, as we saw, a must-alias constraint.

Calling `f` consumes the permission for `h` and gives a new one for `h'`. Similarly, calling `map` consumes the permission for `t` and produces a fresh permission for `t'`. Finally, we return: we call `ret` the return value of the function. The permission for `ret` is, again, a structural one, since we know it is a `Cons` block. The type-checker performs various rewritings and *foldings* to finally obtain `ret @ list b`, which satisfies the declared return type for `map`.

In the second branch, we just learn that `xs` is a `Nil` branch; returning `xs` actually creates a new equation, which the type-checker uses to rewrite the current conjunction, so as to make `ret @ Nil` appear. It turns out that `Nil` can also be seen as a value of type `list b`, and the type-checker is aware of that fact.

3.2 Type-checking the improved map function

The function above has to wait for the recursive call to `map` to complete before constructing the mapped list. Therefore, it uses linear stack space: it is not tail-recursive. There is, however, one way to write this function in a tail-recursive style, called destination-passing style.

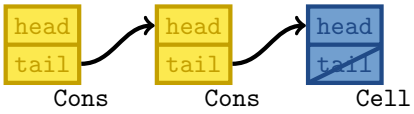

Figure 1: The last cell (blue) of the list is mutable

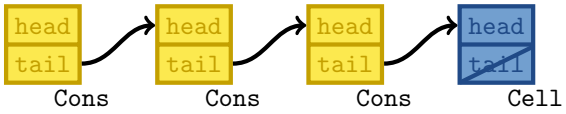

Figure 2: The next step consists in mapping the next element; the blue cell is frozen

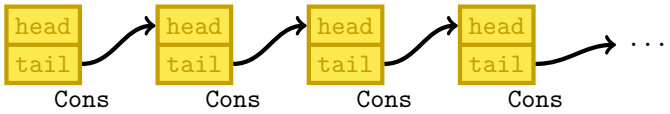

Figure 3: The procedure continues until it reaches the end of the list and the entire list has been mapped

The algorithm, as illustrated above, uses an *unfinished* list. The yellow cells are immutable, but the last cell is mutable (blue). Mapping the next element amounts to allocating a fresh blue cell, wiring the old blue cell to the new one, *freezing* the old, mutable blue cell, into an immutable yellow cell.

A traditional type system such as that of ML cannot express this situation: the in-progress, unfinished list is not representable in a traditional data type. Moreover, since the type of the blue cells moves from mutable, unfinished cells, to finished, well-formed cells, there must be reasoning about ownership, to justify why changing the type of a memory block is sound.

Here's how one would do it in Mezzo.

```
(* This file implements a destination-passing style, tail-recursive
 * version of [List.map] for immutable lists. It uses a temporary mutable cell
 * that is later on "frozen" when it's ready. *)
```

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

data mutable cell a = Cell { head: a; tail: () }

(* This is the recursion loop, which turns [c0], an unfinished, mutable list
cell, into a well-formed, immutable list. *)
val rec map1 [a, b] (
          f: (consumes a) -> b,
          consumes c0: cell b,
          consumes xs: list a
        ): (| c0 @ list b)
      =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let h' = f h in
      let c1 = Cell { head = h'; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end

(* We need to unroll the recursion to take into account empty lists. *)
val map [a, b] (f: a -> b, consumes xs: list a): list b =
  match xs with
  | Nil ->
      xs
  | Cons { head; tail } ->
      let c = Cell { head = f head; tail = () } in
      map1 (f, c, tail);
      c
  end
```

See the HTML demo

The data type of lists is recalled. We define the `cell` type that represents an unfinished, mutable list cell.

The heart of the algorithm is `map1`, a recursive function that implements the main loop. This function transforms `c0`, an unfinished, mutable list cell, into the head of a well-formed, immutable list. Just like the standard definition of `map`, it consumes the original, to-be-mapped list.

The start of the algorithm is `map`, which unrolls the recursion. If the original list is non-empty, `map` creates a fresh cell to stand for the first element of the

new, mapped list, then calls `map1` on it. Per the semantics of `map1`, after the call to `map1` returns, `c` now points to a well-formed, immutable list of `b`.

Let us now focus on the heart of the algorithm, namely, the body of `map1`. In the case where the to-be-mapped list `xs` is empty, we can finish the mapping procedure by putting `xs` (which is `Nil`) into the last cell's `tail`. Once this is done, we turn it into an immutable list cell, by changing its tag from `Cell` to `Cons`. The list is now fully mapped.

In the case where the to-be-mapped list `xs` is not empty, we allocate a new, mutable cell `c1`, whose `head` has been mapped through `f`. The old, still-mutable blue cell `c0` is modified: its `tail` is made to point to `c1`, and it is frozen into an immutable `Cons` cell.

At this stage, `c0` is not yet a valid `list b`: its tail points to an unfinished `cell`, namely `c1`. We call `map1` recursively on `c1`. Per the semantics of `map1`, `c1` is now a `list b`, which allows us to deduce that `c0` also is a `list b`.

It is interesting to note that while the execution of the code is tail-recursive, the reasoning is not: an extra reasoning step takes place *after* the recursive call to `map1`, and is needed to justify why `c0` *has become* a well-formed list.

We provide an interactive feature where you can see the permissions evolving in real-time, as the execution progresses through the example. This should (hopefully) help illustrate the way permissions work on a concrete example.

4 Overview of types

We now present the whole zoology of types in Mezzo. This allows us to provide a reference before going into further discussion; this also allows us to talk about our syntactic conventions for function types in greater detail.

4.1 Kinds

Before going any further, we need to mention that types have *kinds*. Kinds classify types into three categories. A type is either:

- a program variable, such as `x`, at kind **term**
- a permission, such as `x @ t`, `p * q`, `empty`, at kind **perm**
- a regular type, such as `int -> int`, or `ref int`, at kind **type**

We sometimes mention the well-kindedness judgement informally in the syntax of types.

4.2 Syntax of types

Structural types Structural types talk about the *structure* of a memory location. Structural types are either **tuples types** of the form `(t1, ..., tn)`, or **constructor types** of the form `A { f1: t1 ...; fn: tn }`.

Structural types are always *expanded* by the type-checker, meaning that in practice, the type-checker will introduce names so that it manipulates types of the form `(=x1, ..., =xn)` or `A { f1 = x1, ...; fn = xn }`, as stated earlier.

Structural types always have kind **type**.

The **unit type** is the empty tuple, which we write `()`.

Type variables and type applications One may also encounter types variables such as `a` or `int` and type applications such as `list a`. Their kinds depend on how the variable was defined: `a` may be introduced with a quantification such as `[a: k] ...` (it has kind `k`), `int` may be defined as `abstract int: type` (it has kind **type**), and `list a` is a data type, meaning it has kind `type`.

Type / permission conjunction A type can be packed along with a permission, by using the type / permission conjunction, written `(t | p)`, where `t` has kind **type** and `p` has kind **perm**. This is of particular importance, especially in function types. In the case where `t` is the unit type `()`, we offer syntactic sugar, so that the user can write `(| p)`.

Quantified types Mezzo offers **universal** quantification, written as `[a: k] t`, and **existential** quantification, written as `{a: k} t`. The kind `k` is optional and defaults to **type**.

Singleton types Singleton types of the form `=x` requires that `x` be a program variable (hence, at kind **term**). A singleton type has kind **type**.

Permission types The anchored permission `x @ t`, where `x` is a program variable (thus, at kind **term**) and `t` has kind **type**, is a permission, hence at kind **perm**. The conjunction of permissions `p * q` as well as the empty permission `empty` are also naturally at kind **perm**.

Type variables and type applications may also be at kind perm.

Unknown type Is it sometimes convenient to use `unknown`, the top type at kind **type**.

4.3 Function types

Function types benefit from special syntactic conventions, which makes expressing their pre- and post-conditions easier. We thus devote an entire section to function types. Function types have kind **type**.

4.3.1 Consumes keyword

A basic function type is of the form `t -> u`. For instance, here is the signature of the `list::length` function, which we saw earlier.

```
val length: [a] (list a) -> int
```

We said that the permission for the argument was understood to be taken and returned to the caller, which embodies the fact that the ownership of the list is temporarily transferred to `length`, so that it computes the length, and then returned to the caller.

```
(* x @ list a *)
let l = length x in
(* l @ int * x @ list a *)
```

Permissions are not always returned to the caller. Let us take the `list::concat` function as an example. The function takes two lists and concatenates them. In order to be sound, that function must *consume* the ownership of the two original lists, so as to to build their concatenation. Failing to do that would mean that the caller could access an element both through the original list and the concatenated list, thus breaching soundness.

One can express this transfer of ownership using the special `consumes` keyword.

```
val concat: [a] (consumes list a, consumes list a) -> (list a)
```

The `consumes` keyword can appear only on the left-hand side of a function type. It can appear at arbitrary depth, several times; `consumes` may refer to a type with kind **type** or **perm**; `consumes` keywords cannot be nested. The example above would also work with:

```
val concat: [a] (consumes (list a, list a)) -> (list a)
```

Intuitively, all the parts of the argument that are not marked as consumes are preserved through the function call.

Formally, the `consumes` keyword is interpreted as follows. If the function has type `t -> u`, then we take `t1` to be `t` where `consumes v` is replaced by `v`. We also take `t2` to be `t` where `consumes v` is replaced by `unknown` or `empty`. We then understand the function type to mean `(root: t1) -> (u | root @ t2)`. That is, the function preserves the sub-parts of `t` that are *not* consumed.

4.3.2 Name introduction

Names can be introduced in function types using the *name introduction construct*. It is of the form `x: t`.

Any name introduced in the **domain** of a function type is available in the domain and the codomain. For instance, in the example below, both `t` and `u` may refer to `x`.

```
val f: (x: int, t) -> u
```

Any name introduced in the **codomain** of a function type is available in the codomain only. For instance, in the example below, only `u` may refer to `x`.

```
val g: t -> (x: int, u)
```

Name introductions may appear at arbitrary depth. Name introductions are hidden by function types and quantifiers. For instance, in the example below, `t` may *not* refer to `x` or `y`.

```
val h: (
  t,
  ((x: int) -> int),
  {u} (y: u)
) -> int
```

Internally, name introductions in the domain of a function type translate to *universal* quantifiers over the function type, while name introductions in the codomain of a function translate to *existential* quantifiers over the codomain. The name introduction constructs are replaced by a conjunction of a singleton type and a permission. Thus, the three functions above are understood as:

```
val f: [x: term] ((=x | x @ int), t) -> u
val g: t -> ({x: term} ((=x | x @ int), u))
val h: (
  t,
  ([x: term] (=x | x @ int) -> int),
  {u} (=y | y @ u)
) -> int
```

Name introduction constructs may also appear in data types and type annotations.

4.3.3 Currying and capture (advanced)

Please note that the function type above for `concat` use a *tuple* for its argument, instead of being a curried function. We strongly encourage users to write functions that take tuples. Indeed, function types are duplicable, meaning that they can't close over (capture) non-duplicable data. Thus, if one were to write the `concat` function above in a curried style, they would need the following signature:

```
val concat: [a] (xs: list a) -> (
  consumes (list a | xs @ list a) -> (list a)
)
```

We name the first list `xs`. The function type above expresses the fact that applying the function to its first argument will actually *not* modify or consume `xs`; rather, all the action will happen when applying the second argument. Thus, when calling the function on its second argument, the permission for the first argument is required for the function call to succeed.

Writing curried function types is difficult, and the resulting types generate extra work for the type-checker, so currying shouldn't be used unless absolutely necessary.

4.4 Controlling duplication

It is important, when reasoning about ownership, to understand whether a permission is duplicable or not.

4.4.1 To consume or not to consume?

Back to the concat example, from the point of view of the caller, permissions evolve as follows.

```
(* xs @ list a * ys @ list a *)
let zs = concat (xs, ys)
(* zs @ list a *)
```

Here, we set ourselves in the case where `a` is still a type variable, which is non-duplicable (`a` could be anything). Incidentally, this means that a `list` of `a` is also non-duplicable, as duplicating the list would duplicate the ownership of the elements.

What happens if we pick concrete values for `a`? If `a = ref int`, the story remains the same, since `list ref int` is not duplicable either.

```
(* xs @ list (ref int) * ys @ list (ref int) *)
let zs = concat (xs, ys)
(* zs @ list (ref int) *)
```

If, however, we pick `a = int`, the story becomes different. The `list` type denotes immutable lists; `int` also denotes immutable data. Therefore, `list int` denotes an immutable fragment of the heap; it is therefore a duplicable permission. Hence, the type-checker will *automatically save a copy of* `xs @ list int * ys @ list int` before it performs the call to `length`, meaning that the conjunction is *preserved* through a call to `length`.

```
(* xs @ list int * ys @ list int *)
let zs = concat (xs, ys)
(* zs @ list int *)
```

This is call-site polymorphism over the duplicability of the argument. Thus, when writing a function type in Mezzo, it is generally advisable to consume your arguments; this puts no extra burden on the caller, and allows them to save a copy of the permission if they have the ability to do so.

4.4.2 Determining duplicability

Here is an informal set of rules to help you determine whether a type is duplicable or not.

- `=x` is duplicable, as it is just a pointer that contains not ownership information.
- `unknown` and `empty` are both duplicable.
- `(t1, ..., tn)` is duplicable if all of its components are.
- `A { f1: t1; ...; fn: tn }` is duplicable if `A` belongs to an immutable data type, and all of its fields are duplicable. `B { f1: t1; ...; fn: tn }` is exclusive if `B` belongs to an exclusive data type.
- `a`, a type variable introduced by a quantifier, is initially considered affine, that is, neither duplicable or exclusive (as mentioned earlier).
- Mutable data type applications such as `ref int` are exclusive.
- Type aliases are silently expanded.
- Abstract types are affine unless a fact is specified.
- A type / permission conjunction `(t | P)` is duplicable if both `t` and `P` are.

- `[a: k] t` and `{a: k} t` are duplicable if, assuming the variable `a` is affine, the type `t` is still duplicable.
- A permission conjunction `p * q` is duplicable if both `p` and `q` are.
- An anchored permission `x @ t` is duplicable if `t` is.
- Function types `t -> u` are always duplicable.

Immutable data types are not mentioned in the list above. They get a special treatment that requires a fixpoint computation. For instance, Mezzo knows that `list a` is duplicable as long as `a` itself is duplicable. We omit the details.

5 Writing type definitions

Since Mezzo is such a strongly typed language, we devote an entire section on how types are defined in Mezzo.

5.1 Data types

Data type definition may appear at any top-level position in a `.mz` file.

5.1.1 Definition

Let us start with lists, the classical example of a data type.

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

Data type definitions are introduced by the `data` keyword; *branches* are separated using a vertical pipe `|`. Each branch has a *tag*, in our case, `Nil` or `Cons`, along with a number of fields, listed between braces `{ ... }`. In the case where there are no fields, the `{ ... }` may be omitted.

Defining the `list` data type allows one to write types such as `x @ list int`; it also allows one to write two new *concrete* types. Concrete types describe the in-heap structure of a block; after defining `list`, one can write `x @ Cons { head: ...; tail: ... }` to express the fact that `x` is a memory block of size 3, with a `Cons` tag, and two fields `head` and `tail`. Similarly, after defining the `list` type, writing `x @ Nil` also becomes possible.

5.1.2 Mutability

Unless specified, a data type is considered to be immutable. One can define mutable data types using the `mutable` keyword. Here is the definition of *mutable* lists.

```
data mutable mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
```

Field names can be shared between data types, but constructors shadow each other. We thus use `MNil` and `MCons` to make sure one can still talk about `Nil` and `Cons`, which belong to `list`.

5.1.3 Recursion

Data types are recursive by default. There is currently no option to make a data-type non-recursive. Data types in Mezzo are inductive.

Mutually recursive data types can be defined using the `and` keyword. Here is a (slightly artificial) example.

```
data tree a = Tree { size: int; node: node a }
and  node a = Empty | Node { left: tree a; contents: a; right: tree a }
```

5.1.4 Parameterizing data type definitions, packing permissions

The `list` example admits a parameter. Parameters take a kind annotation. When omitted, the kind defaults to `type`.

```
data mutable treeMap k (c : term) a =
  TreeMap { tree: tree k a; cmp: =c | c @ (k, k) -> int }
```

The example above, taken from Mezzo's standard library, showcases a fairly sophisticated data type definition. This is the type of mutable tree maps; it is parameterized of the type of keys `k` and the type of values `a`; it also happens to be parameterized by a certain identifier `c`, which, being a program variable, has kind `term`.

The program variable `c` is the comparison function for keys; by parameterizing tree maps over their comparison function, we make sure that the "right" comparison function is always wired to the data type itself, rather than relying on the user providing the correct comparison function at each call of a function operating on tree maps.

A tree map also needs to embed information about the comparison function, namely, that it is a function that compares keys, with type `(k, k) -> int`. This is done by *packing* a permission inside the branch, using a vertical bar between braces `{ ... | ... }`. Anything left of the bar is fields; at the right of the bar is a permission.

A canonical example of a type using packed permissions is *rich booleans*. Rich booleans embed permissions in their `False` and `True` branches. Their definition, taken from the Mezzo standard library, is shown below. Note the kind annotations on the type's parameters.

```
data rich_bool (p : perm) (q: perm) =
  | False { | p }
  | True { | q }
```

5.2 Type abbreviations

Since we've defined the `list` data type, we can now write types of the form `Cons { head: a; tail: list a }`. This is tedious, though. Fortunately, we can define a type abbreviation.

```
alias cons a = Cons { head: a; tail: list a }
```

Type abbreviations are introduced by the `alias` keyword. They are **not** recursive. They accept type parameters the same way data type definitions do.

While data types always have kind `type`, type abbreviations can have kind `perm` or `type`. Thus, they take an optional kind annotation.

```
alias stashed (p: perm) : perm = p
```

5.3 Advanced topics

The sections above contain enough information to work define types in Mezzo. We present a few extra features of type definitions; these contain quite a few forward references, so a casual reader may want to skip these upon a first reading of the tutorial. Most of the code snippets from this section are taken from Mezzo's standard library.

5.3.1 Binding rules

Both data type branches and type abbreviations can be prefixed with a succession of existential bindings.

```
alias channel a =
  {q: term} (=q, l: lock (q @ fifo a), condition l)
```

The definition above also uses a name introduction construct which is syntactic sugar for introducing a type variable. Writing `l: ...` introduces a new binding for `l`. The binding is interpreted as being an existential one, and is located as high as possible, while still not crossing any other binders. In the example above, `l` is bound below `q`. Thus, the example above is equivalent to:

```
alias channel a =
  {q: term, l: term} (=q, =l, condition l | l @ lock (q @ fifo a))
```

5.3.2 Variance

Both data types and type abbreviations accept variance annotations for their type parameters. Consistently with OCaml, we use `+` for covariance, `-` for contravariance. Mezzo automatically computes variance using the definition; annotations in implementations serve as an assertion that the type has the expected variance. Variances annotations mostly appear when defining interfaces.

```
data mutable fifo +(a: type) =
  Empty    { length: int; tail: ()      }
| NonEmpty { length: int; tail: dynamic }
adopts cell a
```

5.3.3 Axiomatized types

Mezzo makes it possible to define abstract types. This is useful to axiomatize predicates.

```
abstract nests (x : term) (p : perm) : perm
fact duplicable (nests x p)
```

Just like type abbreviations, abstract types take a return kind annotation. They also take an optional fact.

5.3.4 Local type definitions

Local alias definitions Types can be locally defined via a `let`-binding. Note the `let alias` in the snippet below.

```
val new_generic_iterator [a] (consumes l: list a):
  iterator::iterator a (l @ list a) =

  let alias post: perm = l @ list a in
  iterator::wrap [a, (iterator a post), post]
    (new_iterator l, next [a, post], stop [a, post])
```

This is useful for advanced code that requires a lot of type annotations; defining a local alias avoids repeating the same type too many times.

Local data type definitions It is actually possible to define local data types.

```
alias adopter_ t = ref (list t)
alias dynamic_ = unknown

val take2_ [t] exclusive t => (
  parent: adopter_ t,
  child: dynamic_
): rich_bool empty (child @ t) =

  let data outcome =
    | Found { contents: list t | child @ t }
    | Not_found { contents: list t }
  in

  ...
```

The function above needs to perform a search in `parent`, which is a reference to a list of `t`. The search is not a classical one, so using `list::find` is not possible here. We could use the `either::either` type from the Mezzo standard library, by instantiating it into `either::either (list t | child @ t) (list t)`. For clarity, however, we define a local data type with more meaningful constructor names, namely `Found` and `Not_found`.

This feature is to be used with care. Remember that local data types cannot escape their scope.

```
val x =
  let data t = A | B in
  A
```

The example above, when executed, will give the following output:

```
val x @ A
```

This is misleading: the constructor `A` is no longer in scope, so it can't be referenced. The Mezzo printer is currently unable to account for that.

Also, it is unclear what the type of `x` should be in the example below, since in each branch, the type definitions are distinct.

```
val x =
  if true then begin
    let data t = A in
    A
  end else begin
    let data t = A in
    A
  end
```

In that case, the type-checker will just keep `x @ unknown`.

5.3.5 Per-branch mutability

Instead of putting the `mutable` keyword in front of the data type, one can be more precise.

```
data tree k a =
  | Empty
  | mutable Node { left: tree k a; key: k; value: a; right: tree k a; height: int }
```

This refines the "is duplicable" check for the constructor types of `tree`, and also limits assignments to the `Node` case.

6 Writing expressions (programs)

6.1 Top-level definitions

A `.mz` file is made up of an succession of top-level definition. Top-level definitions can be either one of:

- type definitions, see previously,
- value definitions,
- function definitions,
- an open directive, see modules.

6.1.1 Value definitions

Value definitions are introduced by the keyword `val`, followed by a pattern.

```
val x, y = 1, 2
```

Value definitions do not require type annotations.

6.1.2 Function definitions

A function definition is introduced by the `val` keyword, followed by the function name. Function types are always annotated, meaning that if the function is polymorphic, universal quantifiers should be explicitly specified. The type of the argument is mandatory, as well as the return type of the function. The general form of a function definition is thus:

```
val f quantifiers? arg_type: return_type =
  function_body
```

For instance, here are the first few lines of the `list::concat` function.

```
val concat [a] (consumes xs: list a, consumes ys: list a): list a =
  ...
```

The type of the argument deserves some explanation. The name introductions, such as `xs` and `ys` in the example above, follow the same binding conventions that we explained before. However, they are also interpreted as a pattern that binds new names available in the function body. In other words, `xs` and `ys` are names that one can use in a `function_body` because the example above is understood to be:

```
val concat [a, r: term] (=r | r @ (list a, list a)): list a =
  let xs, ys = r in
  ...
```

6.2 Patterns

Pattern are either one of:

- a tuple pattern `(p1, ..., pn)`
- a constructor pattern `A { f1 = p1; ...; fn = pn }`

- an as-pattern `p as x`
- an annotated pattern `p: t`
- a wildcard `_`

Mezzo features punning, meaning that a constructor pattern `A { f1; ...; fn }` will be understood as `A { f1 = f1; ...; fn = fn }`.

6.3 Manipulating concrete types: assigning, reading, matching

6.3.1 Matching

Matches in Mezzo are terminated by an `end` keyword. We do not (yet) provide special facilities for matching on lists, meaning that the user has to write:

```
(* x @ list a *)
match x with
| Cons { head; tail } ->
    (* x @ Cons { head; tail } * head @ a * tail @ list a *)
    ...
| Nil ->
    (* x @ Nil *)
    ...
end
```

The example above uses *punning*. Matching refines permissions in-place, meaning that in the branch of a `match` expression, the permission for the expression being match is refined to a more precise one that uses a constructor type. Once a constructor permission is available for `x`, one may attempt to read and write fields from `x`.

6.3.2 Reading

One can read the field `f` of variable `x` by writing `f.x`. This is well-typed if a permission `x @ A { ...; f: t; ... }` is available in the environment. The type-checker always expands structural permissions, meaning that reading always consists in adding a new equation into the environment.

```
let x = Cons { head = 1; tail = Nil } in
(* x @ Cons { head = h; tail = t } * h @ int * t @ Nil *)
let y = x.head in
(* x @ Cons { head = h; tail = t } * h @ int * t @ Nil * y = h *)
...
```

6.3.3 Assigning

A constructor `B` is said to be mutable if the data type is mutable or if the branch it belongs to is declared as mutable.

One can assign an expression `e` to the field `f` of variable `x` by writing `f.x <- e`. This is well-typed if a permission `x @ B { ...; f: t; ... }` is available and if constructor `B` is mutable. In a similar fashion to reading, performing an assignment changes the singleton type for the field of a constructor permission, meaning we preserve the invariant that all types are in expanded form.

```
let x = MCons { head = 1; tail = Nil } in
(* x @ MCons { head = h; tail = t } * h @ int * t @ Nil *)
x.head <- 2;
(* x @ MCons { head = h'; tail = t } * h @ int * t @ Nil * h' @ int *)
...
```

One can also **change the tag** of a constructor. That is, transform `x @ A { ... }` into `x @ B { ... }`. This is well-typed as long as `A` is mutable and both `A` and `B` have the *exact same number of fields*. If `B` is immutable, the operation turns a mutable piece of memory into an immutable one. This is useful for patterns such as progressive initialization, which we saw in the tail-recursive map example.

```
(* c0 @ Cell { head = c0_h; tail = c0_t } * c0_h @ b * c0_t @ () * xs @ Nil *)
c0.tail <- xs;
(* c0 @ Cell { head = c0_h; tail = xs } * c0_h @ b * c0_t @ () * xs @ Nil *)
tag of c0 <- Cons
(* c0 @ Cons { head = c0_h; tail = xs } * c0_h @ b * c0_t @ () * xs @ Nil *)
```

In the example above, `c0` is a `Cell` that is initialized in a gradual fashion: initially, the `tail` field is not ready. It contains a dummy value: the unit `()` type. Then, we selectively initialize the `tail` field of `c0`, meaning that we are then free to turn it into a `Cons` cell.

6.3.4 Intermediate states

In the examples above, we use constructor types that represent the precise layout of a memory block. These constructor types do not necessarily correspond to a projection of a nominal type. That is, one can write:

```
let x = Cons { head = (); tail = () } in
(* x @ Cons { head: (); tail: () } *)
...
```

The permission for `x` is a perfectly legit one: it represents a memory block of size three. However, it can't be turned into any sort of valid list. One can think of these intermediary states as temporarily broken invariants that are restored when matching a type annotation of the form `list a`. The type annotation could stem, as an example, from the return type of a function.

6.4 Control-flow

6.4.1 Conditionals

Mezzo provides `if-then-else` expressions. Unlike OCaml, we do not allow the following form:

```
if e then
  let y = ... in
  (* sequence of instructions that are parsed into the then-branch *)
else
  ...
```

In that situation, we require you to use the `begin` and `end` keywords for the contents of the `then`-branch.

6.4.2 Loops

Mezzo provides `for` and `while`-based loops. The syntax is as follows:

```
  preserving p while e do e;
  preserving p for x = e to/downto/below/above e do e;
```

Both loop constructs are desugared into recursive functions. Since functions cannot close over non-duplicable data, one may want to indicate a permission that will be used and preserved by the loop. This is the purpose of the optional `preserving` keyword.

Unlike OCaml, we do not use a special `do` / `done` construct. If the loop bodies are sequences, they will have to be enclosed in `begin` / `end` constructs.

Here are examples of loops.

```
val square_naive (x: int, p: int): int =
  let res = newref 1 in
  preserving res @ ref int for i = 1 to p do begin
    res := !res * x;
  end;
```

```
  !res

val square_binary (x: int, p: int): int =
  let res = newref 1 in
  let i = newref 1 in
  preserving i @ ref int while !i < p do i := !i * 2;
  preserving res @ ref int * i @ ref int while !i > 0 do begin
    res := !res * !res;
    if (!i & p) <> 0 then
      res := !res * x;
    i := !i / 2;
  end;
  !res
```

In general, loops are less powerful than recursive functions. The usual style in Mezzo consists in using recursive functions wherever possible.

6.5 Closures

A closure may be introduced either using an anonymous function;

```
let f = fun (x: int): int = x * x in
...
```

or using the more traditional form:

```
let f (x: int): int = x * x in
...
```

We strongly encourage the user to write functions in an uncurried style, for it makes reasoning about permissions easier.

Functions can only close over immutable data; if a function wants to mutate a piece of data, it should require an extra permission.

```
val _ =
  let x = newref 0 in
  let bump_x (| x @ ref int): () =
    x := !x + 1;
  in
  bump_x ();
  bump_x ()
```

The `bump_x` function above would, in OCaml, be a function from unit `()` to unit. Here, we need to specify that the function requires an extra permission in order to run; we take advantage of the special syntactic sugar for `(() | P)` in order to write a concise function signature. Please note that since the `consumes` keyword is not used, the permission is understood to be preserved by the function, meaning that we can call it several times.

6.6 Type annotations

Any sub-expression can be annotated using the `:` colon operator. We do not support nested type annotations, that is, if there already is a type provided by the context, we won't allow you to re-annotate under it.

6.7 Assertions and failures

Any program can fail at run-time by using the special `fail` instruction. The programmer can assert the existence of a given permission `p` by writing `assert p`.

6.8 Polymorphic function calls

In the presence of a polymorphic function call, the Mezzo type-checker will try to guess (infer) the value of the polymorphic variable.

```
open list

val _ =
  let l = cons ((1, 2), nil) in
  print (length l)
```

The program above, when run with `-warn-error +5` will give:

```
File "/tmp/snippetd5da80.mz", line 5, characters 15-28:
We instantiated a as (inst→((inst→int::int), (inst→int::int)))
File "/tmp/snippetd5da80.mz", line 6, characters 16-17:
We instantiated a as (inst→((inst→int::int), (inst→int::int)))
File "/tmp/snippetd5da80.mz", line 6, characters 8-18:
We instantiated a as (inst→int::int)
```

These functions are all polymorphic:

- `cons @ [a] (a, list a) -> a`

- `print @ [a] a -> ()`
- `length @ [a] list a -> int`

The type-checker (accurately) guessed that `cons` was used with type `(int, int)`, just like `length`. It also guessed that the value we wanted to print was an `int`.

This mechanism sometimes doesn't work quite well. Because showing a real-world example here would probably scare the reader off, let us consider the example below:

```
data option a =
  | None
  | Some { contents: a }

val some [a] (consumes contents: a): option a =
  Some { contents }

val x =
  let r = newref () in
  some [(int | r @ ref ())] 1
```

Here, the type-checker has no way of figuring out that we wish to obtain `x @ option (int | r @ ref ())`. The natural choice is having `x @ option int`. Thus, we perform a *type application* by providing the expected value for `option::some`.

The syntax of type applications is either `[t1, t2, ...]` to instantiate type parameters in the order they were declared. An alternative syntax is available, for convenience: if one wishes to instantiate only the type variable named `a`, then one can write `[a: t]`.

6.9 Playing with existentials (advanced)

Existential types are automatically unpacked when added into the context. This means there is no way to refer to "the" existential, as it doesn't have a name.

```
val f (): {t, u} (t, u) =
  1, 2
```

The function above returns a pair whose two elements have existentially-quantified, distinct types. If one calls `f`…

```
val _ =
  let z = f () in
  (* z @ (t, u) *)
```

… then types `t` and `u` are automatically existentially-quantified and the user has no way to refer to them. This can be fixed by using `let flex`.

6.9.1 Binding a flexible variable

The `let flex` construct introduces a new unification variable that will be instantiated as needed.

```
val f (): {t, u} (t, u) =
  1, 2

val _ =
  let x, y = f () in
  (* x @ t * y @ u *)
  let flex t' in
  assert x @ t';
  let flex u' in
  assert y @ u'
```

In the example above, one binds fresh type variables `t'` and `u'`; they could serve for anything, but we require them to satisfy `x @ t'` and `y @ u'` respectively. This means that they have no choice[2] but to instantiate into the existentially-quantified, anonymous variables `t` and `u` introduced by the call to `f`. This henceforth gives us a name that allows us to refer to these variables.

The instantiation choices can be printed out using the `-warn-error +5` command-line argument:

```
File "/tmp/snippet27f964.mz", line 11, characters 2-15:
We instantiated u' as (inst→u)
File "/tmp/snippet27f964.mz", line 9, char 2 to line 11, char 15:
We instantiated t' as (inst→t)
val f @ (consumes /root33: ()) -> ({t} {u} (t, u) | /root33 @ ())
```

6.9.2 Packing an existential

If one wants to perform the converse operation, that is, turning a regular type into an existentially-quantified one, one can use the `pack` instruction.

[2] Actually, there are several choices for instantiating `t'`: `unknown` would be a valid choice, just like `=x`. The type-checker arbitrarily discards these two.

```
alias extpair = {t} (t, t)

val f (): extpair =
  let z = 1, 2 in
  pack z @ extpair witness int;
  z
```

7 Error messages

7.1 Controlling error messages

Error messages can be tamed using the `-warn-error` command-line flag. It follows the OCaml convention: it is a sequence of warning specifiers.

- `+num` enables warning number `num`
- `-num` disables warning number `num`
- `@num` enables and marks as fatal warning number `num`
- `+num` enables warnings in the given range
- `-num` disables warnings in the given range
- `@num` enables and marks as fatal warnings in the given range

The current set of warnings is:

- `1` (disabled by default) *uncertain merge operation*, see the paper "The implementation of Mezzo".
- `2` (enabled by default) *resource allocation conflict*, see the paper "The implementation of Mezzo".
- `3` (enabled by default) *no multiple arguments*, triggered when performing a curried function application
- `4` (enabled by default) *local type*, triggered when a permission is dropped because a type would escape its scope,
- `5` (disabled by default) *instantiation*, prints out the value of an flexible variable when instantiated; the location is the scope of the variable,
- `6` (enabled by default) *multiple function types*, triggered when several permissions are available for the same function *and* a function call would work with several of them: the type-checker makes an arbitrary decision then.

7.2 Making sense out of error messages

Error messages are possibly the main weakness of Mezzo. First of all, the Mezzo type-checker is not complete. That is, an error could mean two things: either that your code is ill-typed, or that the inference of the Mezzo type-checker failed. Fortunately, the latter case is rare, and by the time you write such examples, you'll probably be in touch with us. In this section, we focus on error messages that happen when the programmer makes a mistake.

Let's see what happens if we call an uncurried function in a curried style.

```
File "/tmp/snippetc5cea4.mz", line 6, characters 2-7:
Functions take only one (tuple) argument
File "/tmp/snippetc5cea4.mz", line 6, characters 4-5:
Could not extract from this subexpression (named /x_50) the following type:
(=/root33* | /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int)))
some explanations follow:
could not from /x_50 subtract (
  =/root33*
| /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int))) because none of the following w
  rule Must-Be-Singleton,
    could not subtract: =/x_50 - (
      =/root33*
    | /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int))) because none of the followi
      rule Wrap-Bar-L,
        could not subtract: (=/x_50 | empty) - (
          =/root33*
        | /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int))) because none of the fol
          rule Bar-vs-Bar,
            subtract: =/x_50 - =/root33* using rule Singleton,
              subtract: /x_50 - /root33* using rule Flex-R,
                prove equality: /root33* = /x_50 using rule Instantiate,
            could not subtract: empty - (inst→/x_50) @ (=x*, =y*) *
            y* @ int::int *
            x* @ int::int because none of the following worked:
              rule Add-Sub,
                debug info: t1 = empty t2 = (inst→/x_50) @ (=x*, =y*) *
                y* @ int::int *
                x* @ int::int using rule Remaining-Add-Sub,
                perform: P = P * empty using rule Add-Sub-Add,
                could not subtract a set of permissions (inst→/x_50) @
                  (=x*, =y*) *
                y* @ int::int *
                x* @ int::int because none of the following worked:
                  rule Perms,
                    could not subtract permission (inst→/x_50) @ (=x*, =y*) because none of
```

```
rule Sub-Anchored,
  could not from /x_50 subtract (=x*, =y*) because none of the followi
    rule Try-Perms,
      could not subtract: unknown - (=x*, =y*) because no rule was fou
    rule Try-Perms,
      could not subtract: =/x_50 - (=x*, =y*) because no rule was four
    rule Try-Perms,
      could not subtract: int::int - (=x*, =y*) because no rule was fo
```

The error message above is triggered by trying to type-check the following program:

```
val f (x: int, y: int): int =
 x + 2 * y

val _ =
  f 4 5
```

There are two parts to this error message. The first one is warning 3, which gives us a hint as to what's wrong exactly. We're being lucky.

The second part is the default style of error messages. Indeed, when encountering a type error, the type-checker prints out a *typing derivation*, that is, it prints a sequence of all the typing rules that it tried to apply in order to justify why a particular operation, in this case a function call, is well-typed.

This error message is difficult to read.

- Internal names are all over the place: indeed, the `(x: int, y: int) -> int` type has been translated into the internal syntax that doesn't have the *name introduction construct* or the consumes keyword. The internal version of the type of `f` is the following:

  ```
  [root33: term, x: term, y: term]
    (=root33 | root33 @ ((=x | x @ int), (=y | y @ int)) ->
    (int | root33 @ ((=x | x @ int), (=y | @ int)))
  ```

  A fragment of this type (namely, the domain of the arrow) appears in the first few lines of the error message.

- Subexpressions are also named with auto-generated names. The constant expression `4` is given an internal name (`/x_50`), and the internal name appears.

- The type-checking algorithm uses flexible variables, so every time a name is suffixed with `*`, this is a type variable whose value we're trying to guess.

- The type-checker backtracks: the last six lines show various attempts to prove `/x_50 @ (=x*, =y*)`.

So, what is happening here? Seeing that `f` is a function, the type-checker isolates the domain of the function type, that is, the expected type of the argument: `(=root33 | root33 @ ((=x | x @ int), (=y | y @ int))`. Because `root33`, `x` and `y` are universally quantified, the type-checker turns them into *flexible variables*. Flexible variables are an implementation tool that makes a type variable "waiting to be instantiated". In practice, we're trying to guess what `root33`, `x` and `y` should be.

The argument is the constant `4` expression, named `/x_50`. If that function call is going to work, then `/x_50` *must* have type `(=root33* | root33* @ ((=x* | x* @ int), (=y* | y* @ int))`. In other words, the argument of the function must have the type expected by the function. And that's the operation that failed.

```
Could not extract from this subexpression (named /x_50) the following type:
(=/root33* | /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int)))
some explanations follow:
```

If we rephrase that error message, the type-checker is merely saying: I cannot obtain the permission `/x_50 @ (=/root33* | /root33* @ ((=x* | x* @ int::int), (=y* | y* @ int::int)))`.

The type-checker is not stuck immediately, though: it figures out that if `/x_50 @ (=root33* | ...)` must be shown, then we have to *instantiate* the flexible variable `root33` into `/x_50`:

```
rule Must-Be-Singleton,
  ... (more intermediate steps) ...
    subtract: =/x_50 - =/root33* using rule Singleton,
                  subtract: /x_50 - /root33* using rule Flex-R,
                    prove equality: /root33* = /x_50 using rule Instantiate,
```

The error happens later on:

```
          could not subtract: empty - (inst→/x_50) @ (=x*, =y*) *
```

Namely, the type-checker can't show `/x_50 @ (=x*, =y*)`, that is, that the argument passed to `f` is a tuple. This is normal: that's where we made a mistake. The type-checker thus goes through more intermediate steps, then chockes on the final step:

```
could not from /x_50 subtract (=x*, =y*) because none of the following worked:
  rule Try-Perms,
    could not subtract: unknown - (=x*, =y*) because no rule was found
  rule Try-Perms,
    could not subtract: =/x_50 - (=x*, =y*) because no rule was found
  rule Try-Perms,
    could not subtract: int::int - (=x*, =y*) because no rule was found
```

Several permissions indeed were available for `/x_50`: `/x_50 @ unknown`, `/x_50 @ =x_50`, as well as `/x_50 @ int`. None of them allows the type-checker to deduce that `/x_50` is a tuple.

8 More examples

TODO

- The `map` example
- The `magic-map` example
- The `tail-rec-map` example
- Trees

9 Adoption and abandon

TODO

Re-use some example, say, the FIFO.

10 The Mezzo module system

Mezzo is equipped with a primitive module system. A module is comprised of an implementation (a `.mz` file) and a mandatory interface (a `.mzi` file).

10.1 Exporting items

A signature is made of a succession of items, namely:

- a duplicable value definition, of the form `val x: t`, or

- a type definition,
- an open directive.

Functions are duplicable values, meaning they can be exported without restrictions. Be aware that the colon in an implementation is replaced by an arrow in an interface:

```
val f (x: int, y: int): int =
  ...
```

becomes, in the interface:

```
val f: (x: int, y: int) -> int
```

Type definitions include abstract type definitions, which we saw earlier as a means to axiomatize predicates. They have a more natural use in an interface: using the `abstract` keyword in an interface makes the corresponding type in the implementation, well, abstract.

```
(* mutableTreeMap.mz *)
data tree k a =
  | Empty
  | mutable Node { left: tree k a; key: k; value: a; right: tree k a; height: int }

(* mutableTreeMap.mzi *)
abstract treeMap k (cmp: term) +a
```

10.2 Facts, variance

As mentioned earlier, we may want to reveal *facts* about data types. For instance, we may want to reveal that `treeMap` is exclusive, or that `list a` is duplicable as long as `a` is duplicable. This can be done using the *fact* mechanism.

```
fact exclusive (treeMap k cmp a)
```

Variance annotations can be specified just like we did earlier.

10.3 Referring to things defined in other modules

One can refer to a type, value, or constructor defined in another module using `m::t`, `m::x` or `m::A`. One can import all the items defined by module `m` using `open m`, a top-level directive.

Unlike OCaml, module names exactly follow the case of the corresponding file.

10.4 Restrictions (digression)

Interfaces are mandatory. Because of the expressiveness of Mezzo, using strong updates, one may end up with an implementation whose interface cannot be represented using the interface language (e.g. mutual recursion between a value and a type). We feel like extending the interface language to account for that would be a poor design choice; moreover, writing code that prints the inferred signature while detecting non-representable signatures is non-trivial. We thus went for the easy route and require an interface to be present.

Exporting non-duplicable values is tricky. For instance, if the module dependency graph is as follows:

Let us now imagine the following contents:

```
(* m.mzi *)
val r: ref int

(* n.mz *)
open m
val _ =
  print (!r + 1);
  r := true

(* o.mz *)
open m
val _ =
  print (!r + 1);
  r := !r + 1

(* p.mz *)
open n
open o
```

Depending on the linking order, this program may or may not run into a segmentation fault! Thus, we must require that if module `n` imports module `m`, then `n` must not *alter the signature* of `m`.

We used to have this check in place, but it turns out that exporting non-duplicable values is never used in practice. We thus removed it.

11 Interaction between Mezzo and OCaml

TODO

Interacting through the MezzoLib. Runtime representation of Mezzo (time to implement slim / fat)?

12 Writing idiomatic Mezzo code

12.1 To consume or not to consume?

12.2 Encouraging memory re-use

TODO

In-place zipper example

13 Advanced topics

TODO

existentials (using them, packing them, the auto-unpack feature), let flex, point to articles, etc. Also, mode constraints. Variance, how a -> unit is not covariant (because of the surface syntax hacks). Interaction between OCaml and Mezzo (the restrictions). Declaring environments as inconsistent. Annotation propagation. When to annotate? Merges.

14 The Mezzo standard library

TODO

Nesting (use the union-find example). Channels. Locks. Weak references. Etc. etc.