Mango

Release 0.2.1

Mark

Apr 26, 2021

CONTENTS:

Warning: Mango is in early development; many pages in this documentation are not ready yet.

Mango is a programming language to help you make large software projects reliable, for browsers and servers.

CHAPTER

ONE

QUICK INTRO

Great programmers should use great tools! Writing bug-free software is hard, but Mango is a language that has your back:

- Prevent mistakes with a strong, static type system
- Write composable code using immutability by default
- Prevent data races using ownership (affine types)
- Never forget to handle errors or clean up resources with linear types
- Describe state precisely through non-nullabiity and sum types
- Trust your invariants with easy enforcement
- Catch errors early by doing things compile-time when possible

(These are planned features and are subject to change.)

The guiding principle is that you should be able to reason about small parts of code, because Mango pushes your colleagues towards good design, and there are many invariants that you can rely on.

CHAPTER

TWO

LANGUAGE TOUR

Warning: This is an unconventional mechanism, so this should be considered experimental, and is extra likely to change before Mango 1.0.

2.1 Argument passing

To cut right to the point: Mango passes argument by reference when calling functions. Keep reading to learn more!

2.1.1 By value vs by reference

There are two common ways to 'move' variables, for example when calling a function. They are:

- **As values**, which are passed by copying their value. This is for example the case with primitives in Java, or structs on the stack in C.
- **By reference**, which are passed around by copying only the pointer to the actual data. This is for example the case with Fortran or Perl, and when requested in C++ (`&`).

But while most languages technically copy values, it can look a lot like they pass by reference! That's because a lot of variables are just references/pointers to data. The references get copied, but the data does not.

This is for example the case with everything in Python, objects in Java, and heap-allocated structs in C.

In some cases, this makes no semantic difference (even if it might affect performance). But sometimes it does:

```
fn update_list(x: List<Int>):
    x = [2 * x[0]]
fn update_number(x: List<Int>):
    x[0] = x[0] * 2

let li1 = [1]
update_list(li1)
print(li1)

# list with one element
let li2 = [1]
update_number(li2)
print(li2)
```

The above will show:

- As value: `[1]` and `[1]`, because in both cases, the list is copied by value (deeply), and that copy of the data is being changed.

- As reference: `[2]` and `[2]`, because in both cases, there is a reference to the data being updated, not a copy of it.
- In most languages, where lists are behind pointers: `[1]` and `[2]`. This is because in the first case, a new pointer is assigned, but to `x`, which is just a copy of `li1`, so it does not affect `li1`. In the second case, `x` and `li2` are copies, but they point to the same data. It is the data being pointed to is changed, so `li2` is affected.

Note that these differences show up for mutable data that has more than one reference to it.

2.1.2 Pros and cons

The vast majority of languages copy data when passing (by value). But to varying degrees, they put things behind pointers (everything in Python, non-primitives in Java, or explicitly heap-allocated things in C).

Advantages of passing data directly by copying:

- You're confident a function won't re-assign your variable.
- You do not need to worry how long the referenced data is going to be alive.
- It can prevent heap allocations and associated cleanup, which are expensive.
- It's very fast, for small data.

Advantages of passing a reference to data:

- Functions can change arguments passed in (more powerful).
- The (maximum) size does not need to be known at compile time.
- No need to copy large objects.
- Potentially smaller code size with generics.

2.1.3 Everything by reference in Mango

In Mango, **all variables behave as references**.

As noted above, most languages pass by value. And some advantages of that approach were listed in the previous section.

Why, then, does Mango pass by reference?

- All the advantages mentioned above.
- It is more consistent for `x = [1]` and `x[0] = 1` to affect the outer scope in the same way.
- The disadvantages are mitigated by Mango's other features:
  - Complexity due to mutable parameters is less due to mutability being an explicit, opt-in choice.
  - A substantial part of the speed for small objects can oftentimes still be achieved by the compiler.
  - Data being pointed at cannot disappear in Mango. Data has a single owner, and only (s)he can move it. It gets deleted automatically at the end of its life.

Note: Mango variables *behave* as reference. There is no guarantee that they will be implemented as references. The compiler may choose copy small variables if they are immutable or have a single owner, for example.

In short, by reference seems the more consistent choice for mutable parameters. The performance aspect is something for the compiler to worry about, and Mango's immutability and ownership help with that.

For many experienced programmers, passing by reference will be somewhat surprising. But many junior programmers will have been quite surprised in their early days by the mix of values and references in other languages.

2.1.4 A word of warning

While the idea is to help make mutability of arguments more consistent, please do not mistake this for an encouragement of updating arguments everywhere.

In many cases, the elegant solution is for parameters to be immutable input, and for the return value to be the only output. With opt-in mutability, Mango gives you the tools to work this way.

2.2 Records

Similar to / also known as: class, struct, product type.

2.2.1 Records

Like most languages, Mango lets you combine several individual fields into a single type:

```
record Person:
        name: Name,
        birthday: PastDate
```

You can then create a `Person` instance using

```
let her = Person(
        name = Name::new("Eve").unwrap(),
        birthday = PastDate::new(1988, 1, 31),
)
```

Record types can have behaviour associated with them, mostly using traits, but also directly:

```
impl Person:
        fn greet(&self):
                print("hello, 0:s", self.name)
```

In the future, Mango might support an 'anonymous' version of records, where no record name or member names need to be specified. (This is like tuples in some languages).

2.2.2 No inheritance

Records in Mango cannot extend other records. There is no inheritance of fields or logic.

There is polymorphism, however. A record can implement a trait, and different implementations of the trait can then be used polymorphically. That is, you can write code for `Fruit`, and it can be used for `Apple` and `Orange`. Virtual calls are also supported.

How does Mango solve the problems that Java solves with inheritance?

- Behaviour can be shared by delegating.
- Runtime polymorphism is possible using traits with #.

Why not just inheritance or behaviour and data? This is too big a topic to do justice, but briefly:

- It creates tight coupling between the super and subclass. Especially the inability to change the superclass because anyone might have relied on any of it is harmful (fragile base class).
- It is easy while debugging to be surprised by an overload being called instead of method on the reference type.
- Code-sharing and polymorphic abstraction are separate concepts. If they are conflated, it is easy to end up with subclasses that can't really replace their parent, they just shared some logic.

It is also not possible to embed the fields of a record inside another record, as in Go. You can of course have a record field that is another record.

There is currently no special syntax for delegation, to mimic inheritance. Delegation is encouraged though, so it is not impossible that such syntax will one day be added.

2.3 Unions

Similar to / also known as: sum type, enum, enumeration, oneof.

2.3.1 One of

It is possible to specify that *one of* several types should be used.

```
record Network:
         socket: Socket
         secure: Boolean
record Disk:
        fh: FileReadHandle

union DataSource = Network | Disk
```

In this case, the `DataSource` is either a network or a disk instance.

This could also be achieved by doing:

```
union DataSource:
        record Network:
                socket: Socket
                secure: Boolean
        record Disk:
                fh: FileReadHandle
```

The only difference is that `Network` and `Disk` are inside the namespace `DataSource`.

Unless explicitly implemented, unions automatically implement those trait that are implemented by every variant, by delegating to the current variant.

2.3.2 Tagged vs untagged

There are some differences between these 'one of' types in different languages. Many languages have nothing, or only have enums that cannot hold instance data.

Of languages that have more expressive sum union types, there are two variants:

- Tagged (sum types): each variant has a name and so can be distinguished, even if the types or values are the same. E.g. the type would not be `String | int | int`, but rather `A(String) | B(int) | C(int)`.
- Untagged (union types): possible values are all values of member types, with duplicates removes. So `String | int | int` is the same as `String | int`, as the two integer variants are indistinguishable.

In Mango, it is tracked which variant is present. But the tag is not part of the union, the type is used instead.

This means that Mango unions behave a bit like untagged unions, in that `String | int | int` is not a valid union type. But unlike the (untagged) unions in C, the variant is stored and can be found for any union instance.

You can create your own tags easily, and you are usually encouraged to do so:

```
union Tagged:
        record A(String)
        record B(int)
        record C(int)
```

The advantages of using the type as 'tag' are less verbosity and the ability for a type to be part of multiple unions.

2.3.3 Unions vs trait impl

This has some parallels to inheritance, or to implementing traits.

The variants `A`, `B` and `C` of `union U = A | B | C` are assignable to variables of type `U`, and nothing else is.

The same would be true if `U` were a trait and `A`, `B`, and `C` the only records implementing it.

Indeed, languages like Kotlin implement sum types as a special form of inheritance (using `sealed`). In Mango the parallel is less explicit.

The biggest difference with between union variants and trait implementations, is that union variants form a _closed_ group: all members are known and fixed.

This has the result that downcasting only makes sense for unions, not for traits (there are technical reasons as well). Downcasting a trait to a specific record would mean that that record is treated differently from any other implementation of the trait. This is an odd choice to make if you can't know which implementations there are (because anyone could have created one).

Another difference in some languages, is that a record can implement any number of traits, but a variant can only belong to one union. This is _not_ the case in Mango, a record can belong to multiple unions:

```
record A
record B
record C
union P = A | B
union Q = B | C
```

2.3.4 Notes

- While there are expressive ways to require combinations of types, there is no general way to require the _absense_ of a type (i.e. no way to make sure that a trait is not implemented for a record or union).
- With product types (records) and sum types (unions), Mango has algebraic types.

2.4 Traits

Similar to / also known as: interfaces, protocols

2.4.1 Traits

2.4.2 Intersection types

It is also possible to specify a combination of types, if the types are traits:

```
fn pythagoras(x: T, y: T) -> T where T = Add<T, Out=T> and Mul<T, Out=T>:
        x * x + y * y
```

Here `T` is a type that implements both `Add` + `Mul`.

Intersection types are not distinct concrete types (like records or unions), just bound on existing concrete types. You cannot create an instance of `Add + Mul`, but it is nonetheless very useful for type parameters.

Note that it does not make sense to create an intersection containing one or more known, concrete types.

- Two concrete types does not make sense because an object cannot have two concrete types.
- One concrete type with a trait does not make sense because a concrete type already implies a known set of traits per scope, so it is either always or never satisfied.

2.5 Type parameters

Also known as: type arguments, generics

CHAPTER

THREE

SETUP GUIDE

3.1 Directories and paths

Mango uses these directories:

3.1.1 Configuration directory

This directory contains Mango configuration files.

The default locations are:

- Linux: *$HOME/.config/mango* (actually *$XDG_CONFIG_HOME/mango*), falling back to *$HOME/.mango/config*.
- Windows: *mango* inside *FOLDERID_RoamingAppData*, falling back to *.mango/cache* inside *FOLDERID_Profile*.
- MacOS: *$HOME/Library/Application Support/mango*, falling back to *$HOME/.mango/cache*.

The default location can be overwritten by setting *MANGO_USER_CONFIG_PATH*.

This directory should not take up much space. You should not delete it directory unless you want to reset Mango (or have uninstall it).

3.1.2 User cache directory

A cache directory for cache shared between all projects a user has.

This contains things like:

- Artifacts from external dependencies (but not for project code).
- Mango compiler and daemon state.
- Lock files.

The default locations are:

- Linux: *$HOME/.cache/mango* (actually *$XDG_CACHE_HOME/mango*), falling back to *$HOME/.mango/cache*, or a temporary directory.
- Windows: *mango* inside *FOLDERID_LocalAppData*, falling back to *.mango/cache* inside *FOLDERID_Profile*, or a temporary directory.
- MacOS: *$HOME/Library/Caches/mango*, falling back to *$HOME/.mango/cache*, or a temporary directory.

The default location can be overwritten by setting *MANGO_USER_CACHE_PATH*.

It should be safe to delete this directory, but only if mango is not running. Doing so will make the next build(s) slower.

3.1.3 Project target directory

This contains:

- Compiled binaries and libraries.
- Intermediary build output for project code (but perhaps not for external dependencies).
- Other build artifacts, like static resources.
- Compiler state information.

The default location is a directory *target* at the root of your project.

This default location can be overwritten by setting *MANGO_TARGET_DIR*. It should not be shared with other projects.

It should be safe to delete this directory, but only if mango is not running. Doing so will make the next build(s) slower.

Mango works in a browser, so you can try it without setup!

However, it is not ready enough yet to work with. This guide will follow later!

CHAPTER

FOUR

REFERENCE

4.1 Grammar

Formal grammar description is not yet available.

Warning: The syntax is not complete yet, this will likely be expanded before Mango 1.0.

4.2 Keywords

4.2.1 Keywords

- `import`: At the top of a source file, import another file or package.
- `record`: See *Records*.
- `union`: See *Unions*.
- `trait`: See *Traits*.
- `let`: Declares a variable.
- `mut`: Makes a reference mutable.
- `if`: A condition.
- `else`: In combination with `if`, the alternative case.
- `for`: A for-loop that iterates over an iterator.
- `while`: A while-loop that iterations until a condition no longer holds.
- `in`: In combination with `if` or `for`.
- `fun`: Declares a function.
- `return`: (Early) return from a function (hint: may be omitted for final return).
- `and`: Boolean 'and'.
- `or`: Boolean 'or'.
- `not`: Boolean 'not'.

This list is sure to grow.

4.2.2 Symbols

`%` (percentage) Comment.

`%%` (double percentage) Documentation.

`:` (colon) After a value, this is followed by a type tag.

In combination with various keywords, this starts a block, either multiline by indentation, or inline.

`+`, `-`, `*`, `/` (plus, minus, asterisk, slash) As binary operators on values, these are the usual mathematical operations.

As unary operators, + and − are positive and negative values.

In types, `X*` is a shorthand for a collection of `X`.

As a binary operator for types, + is used to indicate a combination of trait bounds.

`//` (double slash) As binary operator, integer division.

`==`, `>`, `<`, `<=`, `>=` Ordering operators.

`=` (single equals) Assignment, parameter default, named arguments, map literals.

`+=` (operator and equals) Updating assignment (note `a += b` can be a different operation from `a = a + b`).

`"string"` (double quotes) String literal.

Note that single quotes cannot be used for strings.

`x"string"` (letter followed by string) Modified string literal, e.g. raw or templated.

`[1, 2]` (square brackets) As a value, this is an array literal.

As a postfix for a value, this is indexing.

As a postfix for a type, this is a generic parameter.

`{a=2}` (curly braces) Map literal.

`()` (parentheses) Used for grouping, function invocation, used in function signatures.

newline End of statement unless following ellipsis.

`...` (ellipsis) At the end of a line, this means the next line is a continuation of the same statement.

`,` (comma) Separator in array or map literals and in function signatures of calls.

A comma is optional if it would be at the end of a line.

`.` (period) Access a method or field

`?` (question mark) As a postfix for a value, this unwraps a result type, returning early in case of failure.

In types, `X?` is a shorthand for an optional of `X`.

`&` (ampersand) As a prefix for values or types, this indicates borrowing.

> **Warning:** Ampersand might change to postfix.

`#` (hash) As a type prefix, `#X` means a dynamic, mixed types implementing `X` (whereas `X` is a a static, pure type implementing `X`). See *Type parameters*

> **Warning:** Hash might change to postfix.

| (pipe) For values, apply an operation to every element in a collection.

> **Warning:** Pipe is not confirmed yet.

For types, `|` is used to separate union variants.

@ (at) As a value postfix, this awaits the result.

\ (backslash) Used for escaping strings.

Note that these symbols, common in several languages, do not have their C-like meaning in Mango:

- `x++` and `x--`: use `x += 1` or `x -= 1`
- `&&` and `||`: use `and` or `or`

TODO: Several symbols in Mango have a double meaning: one in a type and one in a value.

4.2.3 Reserved

Adding a keyword is a breaking change, so a large number of keywords are currently marked as reserved. Some of them will probably be released before Mango 1.0.

- `abstract`
- `alias`
- `all`
- `annotation`
- `any`
- `as`
- `assert`
- `async`
- `await`
- `become`
- `bool`
- `box`
- `break`
- `by`
- `byte`
- `catch`
- `class`
- `closed`
- `companion`
- `const`
- `constructor`
- `continue`

- data
- debug
- def
- default
- defer
- del
- delegate
- delegates
- delete
- derive
- deriving
- do
- double
- dynamic
- elementwise
- elif
- end
- enum
- eval
- except
- extends
- extern
- false
- family
- field
- final
- finally
- float
- fn
- get
- global
- goto
- impl
- implements
- in
- init

- int
- interface
- internal
- intersect
- intersection
- is
- it
- lambda
- lateinit
- lazy
- local
- loop
- macro
- match
- module
- move
- NaN
- native
- new
- nill
- none
- null
- object
- open
- operator
- out
- override
- package
- param
- pass
- private
- public
- pure
- raise
- real
- rec

- reified
- sealed
- select
- self
- set
- sizeof
- static
- struct
- super
- switch
- sync
- synchronized
- tailrec
- this
- throw
- throws
- to
- transient
- true
- try
- type
- unite
- unsafe
- until
- use
- val
- var
- vararg
- virtual
- volatile
- when
- where
- with
- xor
- yield

Warning: The language is not ready yet, more types will probably be added both before and after Mango 1.0.

4.3 Special types

CHAPTER

FIVE

CONVENTIONS

Mango is in early development, conventions have not been established.

CHAPTER

SIX

GUIDES

Mango is in early development, there are no guides yet.