TyKo

Introduction

TyKo /taɪko/ is project aiming to achieve interoperability between Typst and JVM languages (primary Kotlin, but theoretically all of them). There are several parts, at different stages of development:

- ◧ Manipulating Typst values from JVM
  - [x] Basic types
  - [x] Content and other types
  - [x] Set, show rules
  - ◧ Functions
  - [ ] Plugins, modules
- ◧ Calling Typst compiler from JVM
  - ◧ Compilation
    - [x] PNG
    - [x] SVG
    - [ ] PDF
    - [x] HTML
  - [x] Query
  - ◧ World customization
    - ◧ Library customization
      - [x] Standartized (`inputs`, `features`)
      - [ ] Registering custom functions
      - [ ] Modules
    - [ ] Fonts
    - [ ] Styles by default
    - [x] File management
- [ ] Calling JVM from Typst document
  - [ ] Propagating JVM functions as Typst functions
  - [ ] Loading JARs as plugins
  - [ ] Freely throwing around JVM objects in Typst code.
- ◧ Misc
  - ⊟ Downloading package from central repo
  - [x] Obtaining source code info (parse tree)
  - [ ] Annotation processor for writing overloads

Contents

Library structure

Library consists of three packages:

- `model`

This is the package with all the classes representing Typst values. They mostly are named as `T` and then the type name in UpperCamel: `TInt`, `TFootnoteEntry` etc.

- `ffi`

This is the package with C-faced API. You need to explicitly mark your code `@OptIn(TyKoFFIEntity::class)` to use most of its content, except `TypstSharedLibrary`. They are also not at all guaranteed to remain the same way.

- `compiler`

This is, obviously, for accessing Typst compiler, and all the related things (World, Diagnostics etc). Some API in this package is expected to change, see warning in respective chapters

Model

Typst has dynamic typing, therefore reflecting it in statically typed JVM is quite tricky. Because of that, there are few simplifications. The library defines several types alongside ones defined in Typst docs:

0. All values are immutable. It is forbidden to put mutable `List`s and `Map`s into `TArray` and `TDictionary` and change it afterwards as well, as all the invariants are computed upon creation of `TValue` (common supertype for all Typst values).

1. There are functions, mostly elements, which accept only some sets of strings (`str`s). For example, both `top-edge` and `bottom-edge` parameters of the `highlight` element can be `str`s:

   ```
   highlight(
       fill: none | color | gradient | tiling ,
       stroke: none | length | color | gradient | stroke | tiling | dictionary ,
       top-edge: length | str ,
       bottom-edge: length | str ,
       extent: length ,
       radius: relative | dictionary ,
       content ,
   ) -> content
   ```

   … but not any str: `"ascender"`, `"cap-height"`, `"x-height"`, `"baseline"`, `"bounds"` for `top-edge`, and `"baseline"`, `"descender"`, `"bounds"` for `bottom-edge`. For this, we have enum-like subclasses of `TStr`, named as `T<element><param-name>` respectively, with some exceptions. Whenever you create a `TStr`, it has correct type. For example, `TStr("ascender")` would have type `TTextTopEdge`.

2. There are types, that are dictionaries, but with only specific keys. These are `location`, `point`, `sides` and `margin`. They all extend `TDictionary`, whenever you create a dictionary, the value has respective type. Also, `sides` extend `margin`. Empty dictionary has special separate type, which extends `sides`.

3. As seen above, there are a lot of union types. As there are somewhat around 30 different types in Typst, creating an interface for each possible union is impossible, so only those that are used in constructors are created. This approach somewhat limits typechecking, so, when Typst is stabilized, it'll be changed to more accurately reflect what is what (good example would be the introduction of `Numbering` type instead of `function|str`, or `Smart<T>` instead of `T|none`). For now, all unions have… not so much clever name, like `TFunctionOrStr`.

4. Some `str` values belong to more than one "enum", therefore we need "intersection types". There are much less of those, then of unions, so every inhabited intersection has its own interface. For example, `"smooth"` would be instance of `TCurveCloseModeAndImageScaling`.

5. Most of the interfaces have their `Companion` extend `TType` or `TElement`. Therefore, we can simply write `TInt`, and it is what `type(1)` would be. The exceptions are `TNone` and `TAuto`, which are values. `TNone.type()` and `TAuto.type()` are their respective types.

TValue

`TValue` is the common interface for all Typst values. It has only two methods:

- `fun TValue.repr() : String`

Creates a code representation of the value. The resulting string is a valid code in Typst. It is also atomic from the point of the Typst compiler, so only prepending it by # is enough to make it valid Typst markup.

For now it is not human-readable in any way, but there are plans of improvement (that's why it's external func).

- `fun TValue.type() : TType`

Returns the `type` of the value.

Most of its inheritants are generated automatically, based on the information from the official documentation. There are, however, some types with specific behaviour.

TBool, TInt, TFloat, TStr, TArray, TDictionary, TBytes

These are the types, analogs of which exist in Kotlin. They can be instantiated with `<value>.t`, where `<value>` is respectively:

Typst type	TyKo type	Can be created by `.t` from
`bool`	`TBool`	`Boolean`
`int`	`TInt`	`Byte`, `Short`, `Int`, `Long`
`float`	`TFloat`	`Float`, `Double`
`str`	`TStr`	`String`
`array`	`TArray<out E>`	`List<E>`, where `E : TValue`
`dictionary`	`TDictionary<out V>`	`Map<String, V>`, where `V : TValue`
`bytes`	`TBytes`	`ByteArray`

… and converted back to Kotlin types with `.value`.

In addition to that, `TArray<E>` implements `List<E>` and `TDictionary<V>` implements `Map<String, V>`; and there are helper functions for construction:

```
fun <E : TValue> TArray(vararg elements: E) : TArray<E>
fun <V : TValue> TDictionary(vararg pairs: Pair<String, V>) : TDictionary<V>
```

```
val x = TArray(1.t, true.t)
val y = TDictionary("a" to "b".t, "c" to "d".t)

println(x.repr())
println(y.repr())
```

```
(1, true)
("a" : "b", "c" : "d")
```

TAlignment

This is a representation of `alignment`. There are, also, specific types for horizontal and vertical alignment (`THAlignment` and `TVAlignment`), which are inheritants of the `TAlignment`. Addition is defined:

```
val align = THAlignment.Center + TVAlignment.Horizon
println(align.repr())
```

Numeric types (TRatio, TFraction, TLength, TRelative, TAngle)

They can be created with the same postfixes, as in Typst, but simulated via extension values. All of them contain `float` inside, therefore, can be created from any Kotlin or Typst primitive number:

Typst type	TyKo type	Can be created by with
`ratio`	`TRatio`	`.pc`
`fraction`	`TFraction`	`.fr`
`length`	`TLength`	`.em`, `.pt`
`relative`	`TRelative`	
`angle`	`TAngle`	`.deg`, `.rad`

And for your convinience, there exists addition, but only for `length` and between `length` and `ratio`:

```
//!saturn:eval
println(1.rad.repr())
println((1.em + 1.pt + 1.pc).repr())
```

More operations will be added later.

Colors

Typst type	TyKo type
`cmyk`	`TCmyk`
`color.hsl`	`THsl`
`color.hsv`	`THsv`
`color.linear-rgb`	`TLinearRgb`
`luma`	`TLuma`
`oklab`	`TOklab`
`oklch`	`TOklch`
`rgb`	`TRgb`

All of them extend `TColor`. Also, `TRgb` can be created with `.t` from `java.awt.Color`:

```
import java.awt.Color

println(Color.WHITE.t.repr())
```

```
rgb("#ffffffff")
```

Functions

For specific tools developers' convinience, hierarchy of functions follows Typst's internals:

TyKo type	What is it	Example
`TNativeFunc`	Native function from stdlib	`calc.cos`
`TClosure`	User-defined function with Typst code, possibly with captured values	`it => it * 2` `let f(x) = x + 1; f`
`TWith`	Another function, with some of its arguments preset	`box.with(inset: 1em)`
`TElement`	Specific kind of native function, representing an element function. It also is a `selector`	`box`
`TNativeFunc` (2)	Can be used to create any function from Typst code	

However, when you write functions that only operate Typst values, not analyze them, you can simply ignore all that and only invoke them. That can be done with eiither `apply` method:

```
fun TFunction.apply(args: TArguments<*>): TValue
```

or with operators:

```
operator fun TFunction.get(vararg named: Pair<String, TValue>) : TWith

operator fun TFunction.invoke(vararg args: TValue): TDynamic
```

The idea here is just type checking. As we can't declare parameter type `vararg Pair<...>|TValue`, we separate them. If there are none named arguments, just use `(poistional, args)`. If there are none positional arguments, use empty `()` (empty `[]` are not allowed):

```
val lorem = TNativeFunc("lorem".t)
println(lorem(15.t).repr())
```

```
(lorem)(15)
```

As you can see, no actual computation happened. `operator ()` returns an instance of `TDelayedExecution`, execution of which can be `force`d with appropriate compiler. The reason is, there can be different libraries, and execution of functions can require context.

Content

In addition, it has function `fun TContent.element() : TElement`.

Elements, that usually are available only inside `equations` or with `math` module, have `Math` prefix in them:

Typst type	**TyKo type**
`heading`	`THeading`
`emph`	`TEmph`
`cancel`	`TMathCancel`
`equation`	`TMathEquation`

Also, several type that don't have explicit constructor in Typst, have in TyKo:

TyKo type	**What is it**	**Example**
`TMathAlignPoint`	Align point in equations	`$ 1 & 2 \ 3 & 4 $`
`TContext`	Contextual expression	`#context text.fill`
`TSequence`	Markup block, more or less	`Just _formatted_ text`
`TStyled` (deprecated)	Represents `set` and `show` rules application	`#show emph: set text(fill:red)` `Just _formatted_ text`

```
TSequence(
    TArray(
        TText(text = "Just".t),
        TSpace(),
        TEmph(TText(text = "formatted".t)),
        TSpace(),
        TText(text = "text".t)
    )
)
```

```
Just *formatted* text
```

To avoid boilerplate, there are some improvements. First, `TText` can be created with `.text` from both `String` and `TStr`. Second, `TSequence` has vararg constructor.

```
TSequence(
    "Just".text,
    TSpace(),
    TEmph("formatted".text),
    TSpace(),
    "text".text
)
```

```
Just *formatted* text
```

Third, specifically `TSequence` and `TMathEquation` have DSL constructor, in context of which the text is added via unary +:

```
TSequence {
    +"Just"
    +TSpace()
    +TEmph("formatted".text)
    +TSpace()
    +"text"
}
```

```
Just formatted text
```

Styles (set and show)

For each `element` class, if it has any settable parameters, there exists corresponding `set element` class. The all extend `TSetRule`, and inherently `TStyle`:

```
println(TSetText(fill = java.awt.Color.RED.t).repr())
```

```
set text(fill: rgb("#ff0000ff"))
```

As well as `set`, there are `show` rules. As any `show` rule in Typst, it can either contain `selector` or not, and its `transform` can be either content, function, or an array of styles:

```
TStyled(
    TArray(
        TShowRule(TEmph.Elem, TSetText(fill = java.awt.Color.RED.t))
    ),
    TSequence {
        +"Just"
        +TSpace()
        +TEmph("formatted".text)
        +TSpace()
        +"text"
    },
)
```

```
Just formatted text
```

Besides, in TyKo every `style` is just a content, and can be inserted into the sequence. It then spans until the end of `TSequence`:

```
TSequence {
    +TSequence {
        +TShowRule(TEmph.Elem, TSetText(fill = java.awt.Color.RED.t))
        +"Just"
        +TSpace()
        +TEmph("formatted".text)
        +TSpace()
        +"text."
    }
    +TLinebreak()
    +TEmph("This is outside the scope.".text)
}
```

```
Just *formatted* text.
*This is outside the scope.*
```

TDynamic, delayed execution

`TDynamic` is the type that represents a value of yet unknown type. It is the subtype of ***all*** the TyKo types, except for `TPoint` and `TLocation` (those are excluded due to generic variance problems). Therefore, typechecking can be postponed (usually, until the value is `repr`ed and sent to the Typst compiler).

As was mentioned before, no actual execution happens when you call `invoke` operator on functions. Instead, the instance of `TDelayedExecution` is returned. It is a subtype of `TDynamic`. You can access fields and functions on it, that will result in more `TDelayedExecutions`, but any attempt to convert to Kotlin will fail with an exception:

```
val lorem = TNativeFunc("lorem".t)
println(lorem(15.t).type().repr())
println(lorem(15.t).strValue)
```

```
type((lorem)(15))
java.lang.UnsupportedOperationException
    at org.ldemetrios.tyko.model.TDelayedExecution.getStrValue(DelayedExecution.kt:24)
    at Script.<init>(script.kts:25)
```

You can force execution of a dynamic value, or, as other values can contain dynamic ones as their fields, you can force execution of any value, when you have Typst Compiler instance.

TSelector

These are regular Typst `selectors`. They can be used for querying Typst documents. For your convinience, `TRegex`, `TLabel` and `TElement` are subtypes of `TSelector`. And there are few functions, reflecting operations on selectors in Typst:

```
fun TSelector.before(selector: TSelector, inclusive: Boolean) : TSelector
fun TSelector.after(selector: TSelector, inclusive: Boolean) : TSelector

infix fun TSelector.before(selector: TSelector) : TSelector
infix fun TSelector.after(selector: TSelector) : TSelector

fun TSelector.and(vararg others: TSelector) : TSelector
fun TSelector.or(vararg others: TSelector) : TSelector
```

More on selectors can be found in the chapter about queries.

Compiler

This API is subject to change.
This sign will appear in several parts of this chapter, because current scheme doesn't allow enough flexibility, and is not thread-safe.

To avoid confusion, from now on:

- just "compiler" means an instance of `TypstCompiler` from `org.ldemetrios.tyko.compiler` package, and
- "native compiler" means original Typst compiler, written in Rust.

The only implementation of compiler TyKo now has — is `WorldBasedTypstCompiler`, which relies on entity similar to native compiler's `trait World`. Besides that, it needs an instance of `TypstSharedLibrary`, which gives way to call native compiler.

In a few words, `World` is what works as a bridge between native compiler and a filesystem. Such filesystem can be:

- backed up with actual filesystem, as in Typst CLI
- consist of only one file (`SingleFileWorld`)
- contain no files at all (`DetachedWorld`)
- refer to virtual fs (`ProjectCompilerService` in Kvasir ↗)

etc.

`TypstSharedLibrary` uses JNA to call functions in native compiler. Most of them are unsafe to use and are marked `@TyKoFFIEntity`, but there are several useful functions as well, which are described in respective chapter.

`TypstCompiler` composes that all together and provides functions for evaluation, compilation and querying.

World

This API is subject to change.
This is the way it is only because it reflects native way of handling things. It will be later separated into several entities.

The `World` interface requires to implement several things:

```
interface World {
    fun library(): StdlibProvider
    fun mainFile(): FileDescriptor
    fun file(file: FileDescriptor): RResult<ByteArray, FileError>
    fun now(): WorldTime?
    val autoManageCentral: Boolean
}
```

library

This API is subject to change.
Probably, there will be more flexibility in setting up what is stdlib

This method should return `StdlibProvider`, which describes what should be considered a standard library. For now, it's only an instance of `StdlibProvider.Standard`, which should define two values:

```
val features: List<Feature>
val inputs: TDictionary<TValue>
```

First is the same list of features you pass via `--feature` flag in CLI. The only possible feature available now is `Feature.Html`.

Second is what will be accessible through `sys.inputs`. Unlike with CLI, values can be not only `strs`. For example, Kvasir passes background and foreground colors of current theme.

The result of this method will be obtained only one time, when first compilation is performed.

mainFile

This is not thread-safe.
Because of this you can't compile several documents in parallel with single `World`.

This denotes an entry point to a compilation. This is called at least one time per compilation, except for `evalDetached` calls (in which case — 0 times).

The result of this function better not change during the compilation.

FILE

This function provides access to filesystem. The structure of file descriptors and errors is quite self-explanatory, and closely follows that of native compiler. `RResult` is a analog of Rust `Result`, with two options `Ok` and `Err`.

Results of these functions are cached, you need to explicitly call `.reset()` on compiler.

NOW

This functions establishes "current" time for document. The result of this function is obtained once, before the first compilation. It can be one of three:

- `WorldTime.System` . Before each compilation, compiler will ask system for current time, and sets it as document time.
- `WorldTime.Fixed(Instant)` . Sets time, provided by given Instant, as time for each document.
- `null` . Calls for time from document will result in error.

autoManageCentral

This value determines how native compiler should handle packages from Typst Universe

- When set to `true`, the native compiler will download packages with `preview` namespace and cache in user's folder as it does in CLI.
- When set to `false`, it will ask `World` to provide access to files.

Files with non-null `PackageSpec`, but not `preview` namespace are unaffected by this flag, attempts to access those will be redirected to the `World` regardless.

TypstSharedLibrary

This is an interface that provides access to the native compiler. In theory you could provide your own implementation, but it's extremely unsafe. The way you create an instance of this interface is through `TypstSharedLibrary.instance(Path)`. The path should point to a correct shared library file for current OS and architecture. It is forbidden to create more than one from `TypstSharedLibrary` from a single `.so`, `.dll` or `.dylib` file. While `instance` method tries to prevent it from happening by caching instances, you will most likely get an `IllegalStateException`. You, however, in theory are allowed to create Library from physically different files, it was not tested.

Most of the functions require World to function, but there are several you can use without one.

FORMAT

```
fun TypstSharedLibrary.format(string: String, column: Int, tab: Int): String
```

Formats the given source using Typstyle.

PARSESOURCE

```
fun TypstSharedLibrary.parseSource(string: String, mode: SyntaxMode):
FlattenedSyntaxTree
```

Parses source in given `SyntaxMode` (either `Markup`, `Code`, or `Math`), and returns flattened syntax tree. Flattened syntax tree is a list of indexed marks, each mark is either:
- Start of syntax node with given SyntaxKind
- Start of erroneous syntax node with given error message.
- End of syntax node.

ANYINSTANCE

Returns any of the created instances (expectedly the first created), or throws NoSuchElementException, if there are none.

EVICT_CACHE

As native compiler uses memoization, there could appear memory "leaks". For example, evaluating

```
fun main() {
    val world = WorldBasedTypstCompiler(sharedLib, DetachedWorld())
    for (i in 0 until 20000000) {
        world.evalDetached("1 + 2")
    }
}
```

causes 4.2Gb of cache to be allocated. That means, that when writing heavy applications with compilation of differrent documents, you should call `evict_cache` from time to time. This function redirects call straight to `comemo::evict`.

Typst Compiler

There are several functions available.

Compilile "raw"

This API is subject to change.
It will later be separated into compilation to `PagedDocument`/`HtmlDocument`, and actual rendering, to allow partial (re-)compilation.

```
fun TypstCompiler.compileHtmlRaw(): Warned<RResult<String,
List<SourceDiagnostic>>>

fun TypstCompiler.compileSvgRaw(fromPage: Int, toPage: Int):
                Warned<RResult<List<String>, List<SourceDiagnostic>>>

fun TypstCompiler.compilePngRaw(fromPage: Int, toPage: Int, ppi: Float):
                Warned<RResult<List<ByteArray>, List<SourceDiagnostic>>>
```

`SourceDiagnostic` represents an error or a warning arose during compilation. It as well closely follows native structs, with one difference being `Span`. Unlike native Span, it already contains index, line and column for both text range start and end. Indices are byte indices, not character indices! Those are the same only when you work with ASCII files.

PDF compilation is not yet available.

Compile

```
fun TypstCompiler.compileHtml(): String

fun TypstCompiler.compileSvg(fromPage: Int = 0, toPage: Int = Int.MAX_VALUE):
                List<String>

fun TypstCompiler.compilePng(
    fromPage: Int = 0, toPage: Int = Int.MAX_VALUE, ppi: Float = 144.0f
) : List<ByteArray>
```

This ignores all warnings, and unwraps RResult into "natural JVM way" of handling things: if it compiled, return result, if not, throw an exception. Trace of the first error is then added to the stacktrace, others are added as suppressed.

```kotlin
val source = """
    #let f(x) = 1 / 0
    #let g(x) = f(x) * 2
    #let h(x) = g(x)
    #show emph: it => h(it)
    This is _emphasized_
"""

val compiler = WorldBasedTypstCompiler(
    sharedLib, SingleFileWorld(source, Feature.ALL)
)

compiler.compileSvg()
```

```
org.ldemetrios.tyko.compiler.TypstCompilerException: cannot divide by zero
    at main.typ.f(main.typ:2)
    at main.typ.g(main.typ:3)
    at main.typ.h(main.typ:4)
    at main.typ.show emph(main.typ:5)
    at main.typ.<source>(main.typ:6)
    at org.ldemetrios.tyko.compiler.TypstCompilerKt.compileSvg(TypstCompiler.kt:58)
    at org.ldemetrios.tyko.compiler.TypstCompilerKt.compileSvg$default(TypstCompiler.kt:53)
    at Script.<init>(script.kts:34)
```

```kotlin
val source = """
    #set heading(numbering: "1.")

    = Fibonacci sequence
    The Fibonacci sequence is defined through the
    recurrence relation.
    It can also be expressed in _closed form._

    #let count = 8
    #let nums = range(1, count + 1)
    #let fib(n) = (
        if n <= 2 { 1 }
        else { fib(n - 1) + fib(n - 2) }
    )

    The first #count numbers of the sequence are:
    #(nums.map(fib).map(str).join(", "))
""".trimIndent()

val compiler = WorldBasedTypstCompiler(
    sharedLib, SingleFileWorld(source, Feature.ALL)
)

TRaw(compiler.compileHtml().t, lang = "html".t)
```

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <h2>1. Fibonacci sequence</h2>
    <p>The Fibonacci sequence is defined through the recurrence relation. It can also be expressed
in <em>closed form.</em></p>
    <p>The first 8 numbers of the sequence are: 1, 1, 2, 3, 5, 8, 13, 21</p>
  </body>
</html>
```

Query

```kotlin
fun TypstCompiler.queryRaw(selector: String, format: SerialFormat):
                Warned<RResult<String, List<SourceDiagnostic>>>

fun TypstCompiler.query(selector: TSelector): TArray<TValue>
```

As clearly seen, `queryRaw` takes selector as a String, and its `format` can be any serialization format Typst supports: PrettyJSON, JSON, YAML. `query` returns just the queried values (it uses JSON under the hood, but that's a story for another time). Unlike official Typst compiler, TyKo can properly serialize and deserialize any value, with, probably, slight hiccups on functions (those are still WIP).

~~~kotlin
val source = """
    #show raw.where(lang: "typ") : it => [
        #metadata(it) <x>
        #it
    ]

    Parentheses are smartly resolved, so you can enter your expression as
    you would into a calculator and Typst will replace parenthesized
    sub-expressions with the appropriate notation.

    ```typ
    $ 7.32 beta + sum_(i=0)^nabla (Q_i (a_i - epsilon)) / 2 $
    ~~~
    Preview
    Not all math constructs have special syntax.
    Instead, we use functions, just like the image function we have seen before.
    For example, to insert a column vector, we can use the vec function.
    Within math mode, function calls don't need to start with the \# character.
    ```typ
    $ v := vec(x_1, x_2, x_3) $
    ```
"""

val compiler = WorldBasedTypstCompiler(
    sharedLib, SingleFileWorld(source, Feature.ALL)
)

val code = compiler.query(TLabel("x".t))
    .map { it as TMetadata<*> }
    .map { it.value as TRaw }
    .map { compiler.evalDetached("[\n" + it.text.strValue + "\n]") }

TSequence(TArray(
    code.flatMap { listOf(it as TContent, TParbreak(), TParbreak()) }
))
```

$$7.32\beta + \sum_{i=0}^{\nabla} \frac{Q_i(a_i - \varepsilon)}{2}$$

$$v := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Eval detached

This API is subject to change.

It will later require only stdlib, not the World, to eval.
```

```
fun TypstCompiler.evalDetachedRaw(source: String):
                RResult<String, List<SourceDiagnostic>>

fun TypstCompiler.evalDetached(source: String): TValue
```

`evalDetached` just evaluates an expression. Note that it treats an expression as in code mode, not markup mode. Warnings are not issued here.

Reset

This API is subject to change.
It will later be possible to partially clear cache, and share file cache between several compilations.

```
fun TypstCompiler.reset()
```

This function clears the file cache inside the compiler.

Force

```
fun TypstCompiler.force(value: TValue): TValue {
    return evalDetached(value.repr())
}
```

This function forces execution of any `TDelayedExecution` in the value. This is the same as call `repr` and then `evalDetached`… which is exactly what `force` does.

```
val compiler = WorldBasedTypstCompiler(sharedLib, DetachedWorld())

val lorem = TNativeFunc("lorem".t)
println(lorem(15.t).repr())
println()
println(compiler.force(lorem(15.t)).repr())
```

```
(lorem)(15)

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore."
```

It, however, ***does not*** expand (apply) `set` and `show` rules. That's what *realization* is for, and it is not supported here yet.

Examples & applications

Did you mean "recursion"?

This entire documentation is created with the help of TyKo.

This documents contains metadata inserts:

```
#let ignore-results = sys.inputs.at("saturn-run", default: false)

#let cnt = state("saturn", 0)

#let invoke(func, ..args, handler) = context {
    [#metadata((func, args))#label("saturn-import-" + str(cnt.get()))]
    if (ignore-results) [] else {
        handler(read("saturn-output/" + str(cnt.get()) + ".typc"))
    }
    cnt.update(it => it + 1)
}

...

#context [#metadata(cnt.get())<saturn-import-num>]
```

Runner then queries that metadata by labels, evaluates function calls, and writes results into files nearby. For example, let's take a look…

```
#context query(<saturn-import-0>)
```

```
(metadata(value: ("add", arguments(1, 2))),)
```

This just ensures that runner is working correctly:

```
#read("saturn-output/0.typc")
```

```
3
```

And a simple `show` rule transforms `kt` raw inserts into compiler calls…

I could insert the code of the runner here, and make it evaluate itself, but I'm just to lazy to think through layers of recursion.

Website

Typst can compile to HMTL now, right? Let’s use it!

Here’s a simple server using Ktor. First, we’ll need a `World`

```
val PARAMS_PATH = "${File.separator}__query-parameters.typc"

class RealWorld() : World {
    lateinit var main: Path
    lateinit var params: Map<String, List<String>>

    override fun file(file: FileDescriptor): RResult<ByteArray, FileError> =
        when (file.pack?.namespace) {
            null -> {
                val f = RealWorld::class.java.classLoader.getResource(
                    file.path.drop(1).replace(File.separator, "/")
                )

                when {
                    file.path == PARAMS_PATH -> RResult.Ok(
                        TDictionary(
                            params.mapValues { TArray(it.value.map(String::t)) }
                        ).repr().toByteArray()
                    )

                    f == null -> RResult.Err(FileError.NotFound(file.path))

                    else -> try {
                        RResult.Ok(f.readBytes())
                    } catch (e: IOException) {
                        RResult.Err(FileError.Other(e.message))
                    }
                }
            }

        else -> RResult.Err(FileError.Package(PackageError.Other(
            "Custom namespace package are not allowed here"
        )))
    }

    override fun library(): StdlibProvider = object : StdlibProvider.Standard {
        override val inputs: TDictionary<TValue> = TDictionary("ktor" to true.t)

        override val features: List<Feature> = listOf(Feature.Html)
    }

    override fun mainFile(): FileDescriptor =
        FileDescriptor(null, main.toString())

    override fun now(): WorldTime? = WorldTime.System

    override val autoManageCentral: Boolean = true
}
```

Several things are going on here. First of all, for now `inputs` should be established upon `World` creation. That's why we will pass parameters through a fictional file. Second, `main` file is not pinned here, but should remain the same during compilation. For later use, I will simplify that, but in real cases a `ThreadLocal` should be created there. Hopefully, it will be fixed in little time.

Now, we start a server:

```kotlin
fun main() {
    val sharedLib = TypstSharedLibrary.instance(Path(SHARED_LIBRARY_PATH))
    val world = RealWorld(Path("/home/ldemetrios/Workspace/KtorPlay"))
    val compiler = WorldBasedTypstCompiler(sharedLib, world)

    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        routing {
            get("/{path...}") {
                try {
                    val path = call.parameters.getAll("path")
                        ?.joinToString(File.separator) ?: ""

                    compiler.reset()
                    world.main = Path(path)
                    world.params = call.request.queryParameters.toMap()
                    val result = compiler.compileHtml()
                    call.respond(
                        TextContent(
                            result,
                            ContentType.Text.Html.withCharset(Charsets.UTF_8),
                            HttpStatusCode.OK
                        )
                    )
                } catch (e: Throwable) {
                    call.respondText(e.stackTraceToString())
                }
            }
        }
    }.start(wait = true)
}
```

Nothing special. Just simple Ktor server.

Now only Typst files remain.

```typst
#let params = if sys.inputs.at("ktor", default: false) {
    eval(read("/__query-parameters.typc"))
} else { (:) }

Hello, #params.at("name", default:("World",)).at(0)!
```

Here, we see if it's an application running us (`ktor` is set to `true`), or just a preview in editor. If it's the former, we read query parameters. Each parameter is an `array` of `str`, usually of size `1`.

Now we query our newly created “website”:

http://127.0.0.1:8080/main.typ

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
        <p>Hello, World!</p>
    </body>
</html>
```

http://127.0.0.1:8080/main.typ?name=Dmitry

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
        <p>Hello, Dmitry!</p>
    </body>
</html>
```