2P-Kt: A Kotlin Multi-Platform ecosystem for Symbolic AI

Giovanni Ciatto[1] Lorenzo Rizzato[2]

[1] Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna
giovanni.ciatto@unibo.it

[2] lorenzo.rizzato@studio.unibo.it

May 2021

tuProlog vs 2P-Kt

tuProlog in the **past**

- Malleable Java-based Prolog interpreter
- Java library for Prolog-Java interoperability

tuProlog **nowadays**

- An open, multi-platform, multi-paradigm ecosystem for LP
- 2P-Kt as the main, Kotlin-based implementation
  - currently targetting both JVM and JS
  - involving both GUI and CLI executables
  - → providing a rich library for LP
- Where each LP aspect is made individually available for re-use

Useful Links

tuProlog project homepage

`http://tuprolog.unibo.it`

tuProlog GitHub repository **(please support us with your star =)**

`https://github.com/tuProlog/2p-kt`

Other useful links

- `https://github.com/tuProlog`
  - GitHub organization containing all tuProlog-related software projects
- `https://gitlab.com/pika-lab/tuprolog/2p-in-kotlin`
  - GitLab repository where the actual development occurs
- `https://pika-lab.gitlab.io/tuprolog/2p-in-kotlin`
  - documentation of 2P-Kt

Outline

Next in Line...

Focus on...

Implementations of Prolog – Chronological Perspective

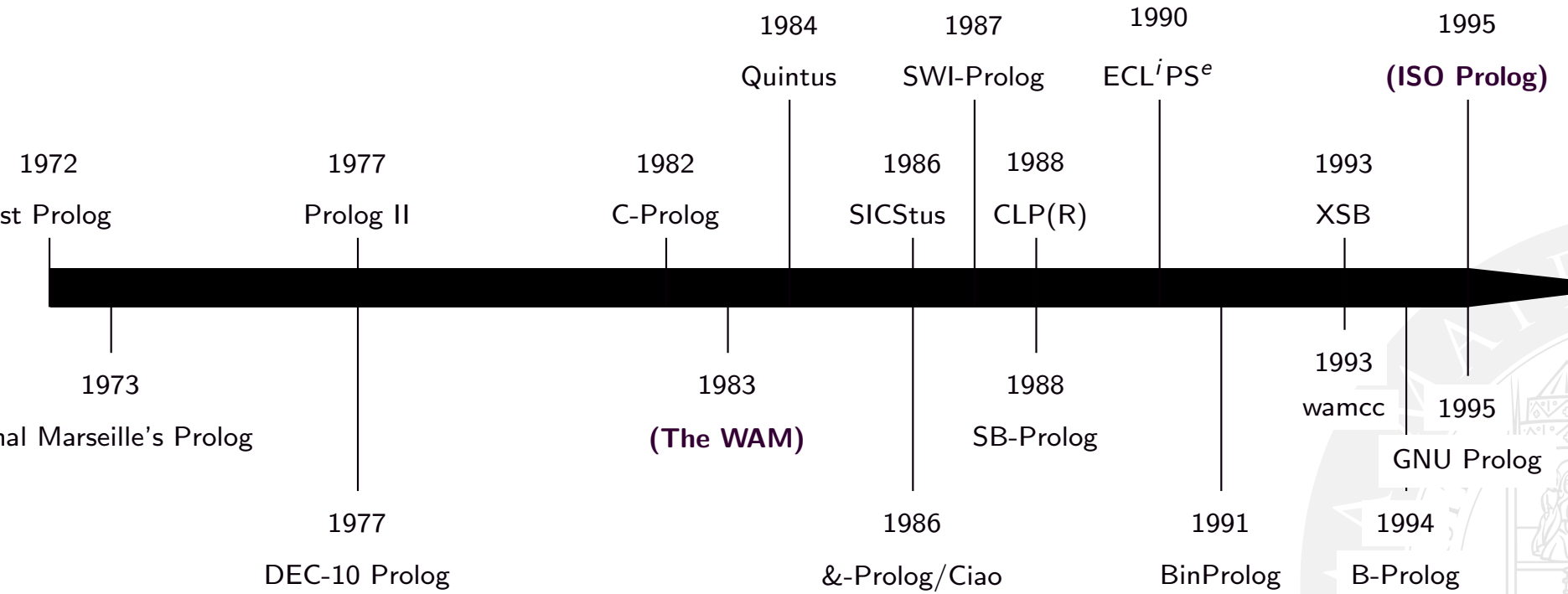


Figure: Time line of Prolog implementations, up to the definition of the ISO Standard [1] (source [9])

Implementations of Prolog – Family Tree

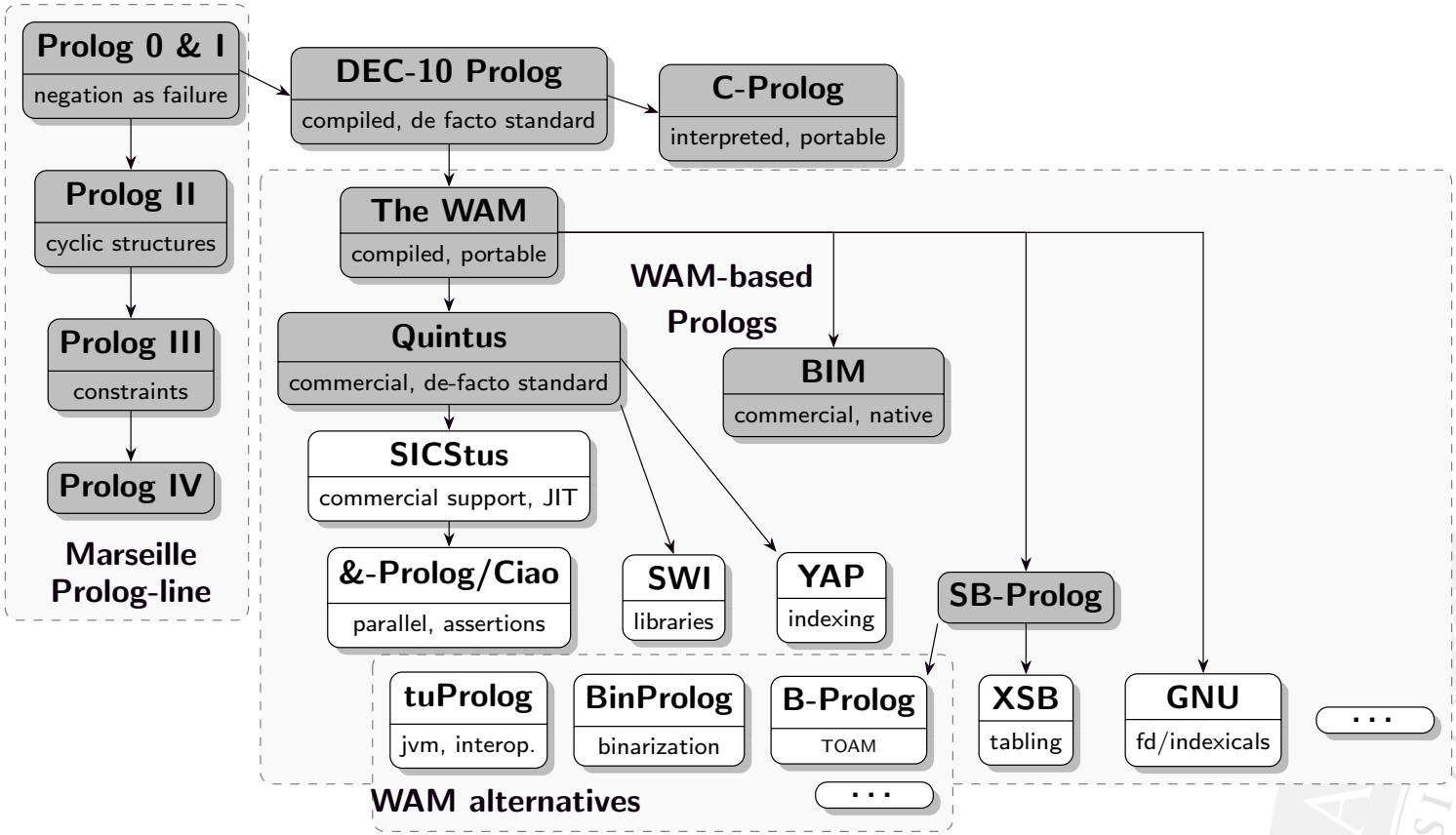
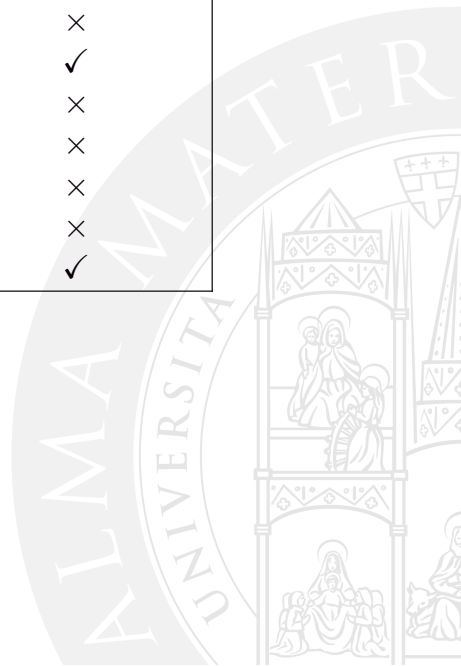


Figure: Prolog implementations in perspective. Notice the pivotal role of the WAM [19]

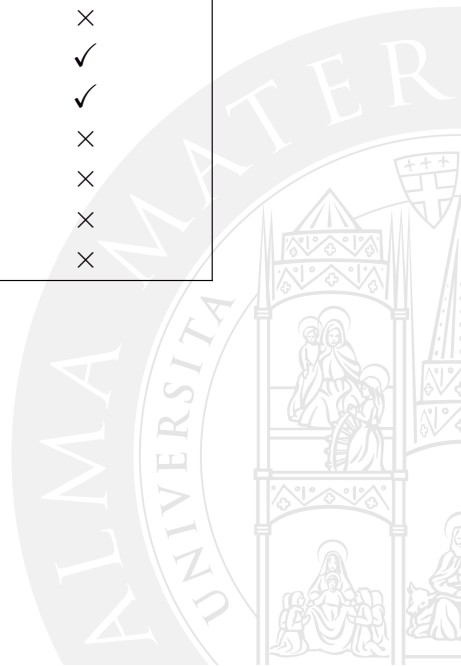
Implementations of Prolog – Features (from [9]) I

System	Open Source	Modules	Tabling	Parallelism	CLP	CHR	Global Variables
B-Prolog	×	×	✓	×	FD, B, Set	(✓)	✓
Ciao	✓	✓	✓	✓	Q, R, FD	✓	✓
ECLiPSe	✓	✓	×	✓	FD, Q, R, Set	✓	✓
GNU Prolog	✓	×	×	×	FD, B	×	✓
Jekejeke	×	✓	×	✓	FD, B	✓	×
JIProlog	✓	✓	×	×	×	×	×
SWI	✓	✓	✓	✓	FD, B, Q, R	✓	✓
SICStus	×	✓	×	×	FD, B, Q, R	✓	×
tauProlog	✓	✓	×	×	×	×	×
tuProlog	✓	×	×	✓	×	×	×
XSB	✓	✓	✓	✓	R	✓	×
YAP	✓	✓	✓	×	✓	✓	✓



Implementations of Prolog – Features (from [9]) II

System	Indexing	Type / Mode	Co-Routines	Testing	Debugger	Mutable Terms
B-Prolog	N-FA	×	(✓)	×	trace	×
Ciao	FA, MA	✓	✓	✓	trace / source	✓
ECLiPSe	most suitable	×	✓	✓	trace	×
GNU Prolog	FA	×	×	×	trace	✓
Jekejeke	FA, N-FA, MA	×	✓	×	spy	×
JIProlog	undocumented	×	×	×	trace	×
SWI	JIT, MA, deep	×	✓	✓	trace / graphical	✓
SICStus	FA	×	✓	✓	trace / source	✓
tauProlog	undocumented	×	×	×	×	×
tuProlog	FA	×	×	×	spy	×
XSB	all, trie	×	✓	×	trace	×
YAP	FA,MA,jit	×	×	×	trace	×



Implementations of Prolog – Features (from [9]) III

System	FLI	Non-Standard Data Types
B-Prolog	C, Java	arrays, sets, hashtables
Ciao	C, Java, Python, JS	×
ECLiPSe	C, Java, Python, PHP	arrays, strings,
Jekejeke	Java	arrays
JIProlog	Java	×
GNU Prolog	C, Java, PHP	arrays
tauProlog	JavaScript	×
tuProlog	Java, .NET, Android, iOS	arrays
SWI	C, C++, Java	dicts, strings
SICStus	C, Java, .NET, Tcl/Tk	×
XSB	C, Java, PERL	×
YAP	C, Python, R	×

Focus on...

tuProlog History I

The original idea

- light-weight Prolog system for distributed applications [7]
- intentionally designed around a minimal core
- *configurable* by loading/unloading libraries of predicates
- native support for multi-paradigm programming [8]
  - bi-directional integration among OOP and Prolog

tuProlog History II

- first release lightweight Prolog solver written in Java [7]
  - OOP-interoperable
  - state-machine-based [16]

- foundation of many research projects

  TuCSoN / ReSpecT coordination model for Internet applications based on network-aware and mobile agents [13] by means of reactions to communication events [12]

  MoK self-organising knowledge-oriented model based on biochemical tuple space [10]

  LPaaS micro-intelligence vision (lightweight logic solvers running on most devices) for IoT and Cloud/Edge computing [2]

  TuSoW bringing tuple-based coordination at the edge [5]

  Arg2P defeasible reasoning and argumentation tool [17]

tuProlog History III

Why a complete re-write

- Need to support LP in general, not only Prolog
  - re-design made 2P-Kt an ecosystem for LP
  - widening the scope w.r.t. Prolog
- Need to get rid of a lot of legacy-code
  - accumulated as the code base stepped through many persons
  - sub-optimal design choices due to limitations of early Java
  - lack of fine-grained test units made the codebase hard to maintain
  - lack of module & package segregation
  - lack of automated dependency management

Next in Line...

Why 2P-Kt

- Building an open an extensible ecosystem...
- ...supporting multiple logics (e.g. FOL, DL, TL, BDI, etc)
  - via as many inference rules as possible (e.g. deduction, abduction, induction)
    - implemented, in turn, via multiple resolution strategies (e.g. SLDNF, IFF, Probabilistic, etc.)
- Making each aspect of LP individually usable *per se*
- Reaching widest possible platform support
- Blending LP with OOP and FP
- Bridging symbolic and sub-symbolic AI

Which features of LP

1. Knowledge representation: terms, and clauses
2. Unification, based on [11] but possibly customisable
3. Clauses indexing and in-memory storage facilities for knowledge bases
4. Prolog-like syntax parsing & formatting
5. Serialization / deserialization of terms, clauses, and knowledge bases
6. Generic resolution API, possibly customisable via pluggable features
7. SLD NF (Prolog-like) resolution strategy [18, 6]
8. Support for other resolution strategies
9. Common UI facilities

Why Kotlin

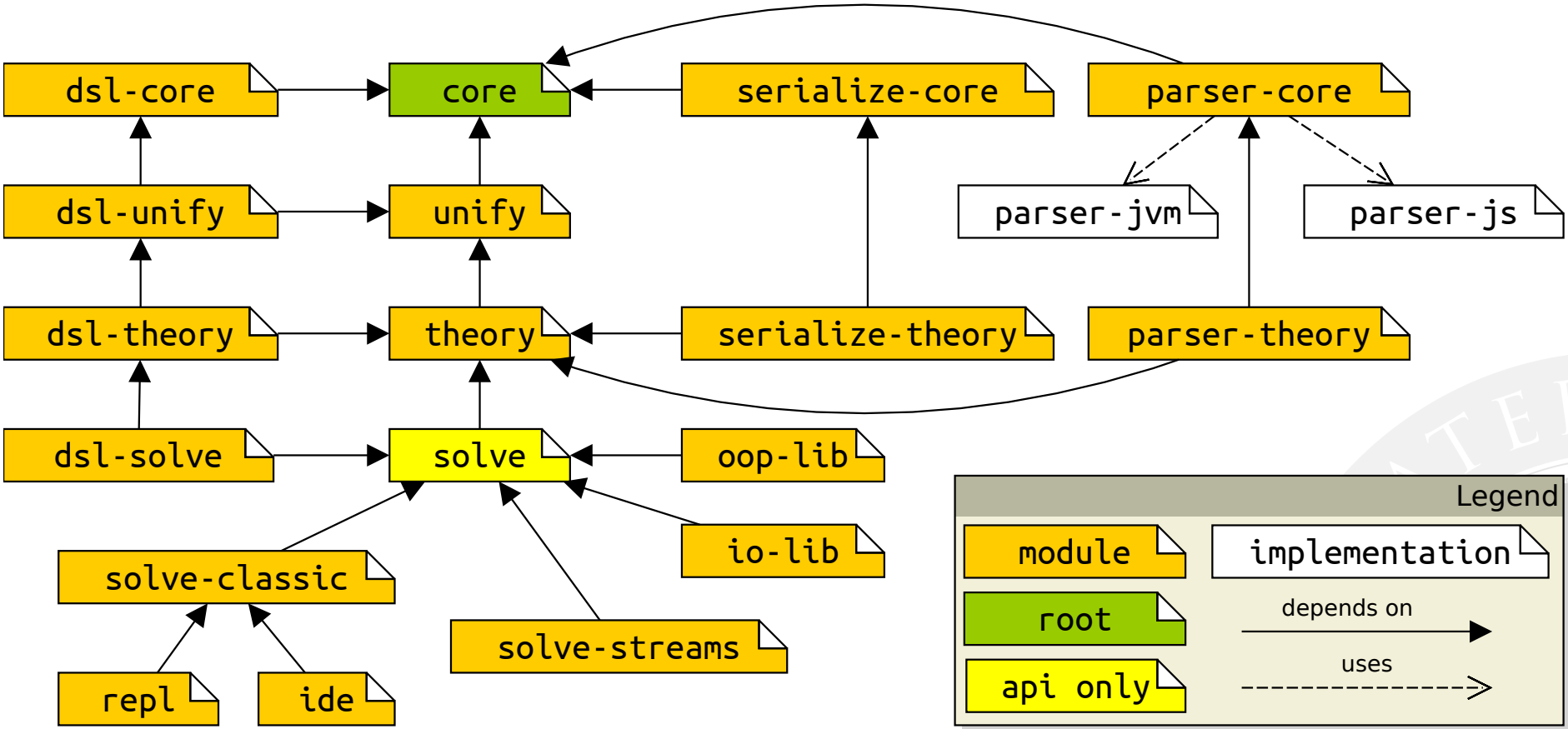
- Multi-platform support:
  - JVM (Win + Linux + Mac)
  - JS (Web, both browser and server side)
  - Android
  - Native?
  - iOS?
- Good platform-specific interoperability
  - Kotlin libraries can easily be exploited in bare Java projects
  - Kotlin libraries can be exploited in bare JavaScript projects
- Clean and practical framework and tool-kit available

Next in Line...

Project Map I

- 2P-Kt is an ecosystem of modules
  - denoted by Gradle's notation: *:moduleName*
- Modules are loosely-coupled, yet incrementally inter-dependent
  → onion-like architectural design
- Modules are compilation and deployment units
  - 1 module $\leftrightarrow$ 1 jar, on the JVM
- Using a module as a dependency $\implies$ importing all its dependencies

Project Map II



Project Map III

:core provides knowledge representations facilities & common features

eg terms, clauses, substitutions, operators, term formatting, basic exceptions, etc.

:unify provides support for logic unification

eg customisable notion of unificator based on [11]

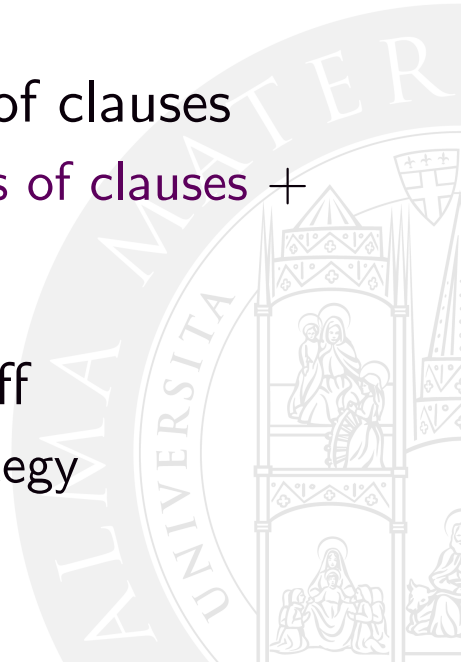
:theory provides support in-memory storage & indexing of clauses

eg mutable/immutable and ordered/unordered collections of clauses + logic theories

:solve provides generic support for resolution-related stuff

! agnostic w.r.t. inference procedures & resolution strategy

eg solvers, solutions, libraries, errors, flags, channels, etc.



Project Map IV

*:solve-** provide specific implementation for inference procedures & resolution strategy

- *:solve-classic* SLD NF (Prolog-like) resolution [18, 6] based on Piancastelli's state machine [15] (Prolog ISO [1] Compliant)
- *:solve-streams* SLD NF (Prolog-like) resolution [18, 6] based on Enrico Siboni's master thesis and on [3]

*:parser-** supports parsing of terms and clauses in Prolog syntax

- *:parser-core* supports parsing terms
- *:parser-theory* supports parsing knowledge bases and streams of clauses
- *:parser-jvm/js* platform-specific implementations, based on ANTLR [14]

(not to be used directly!)

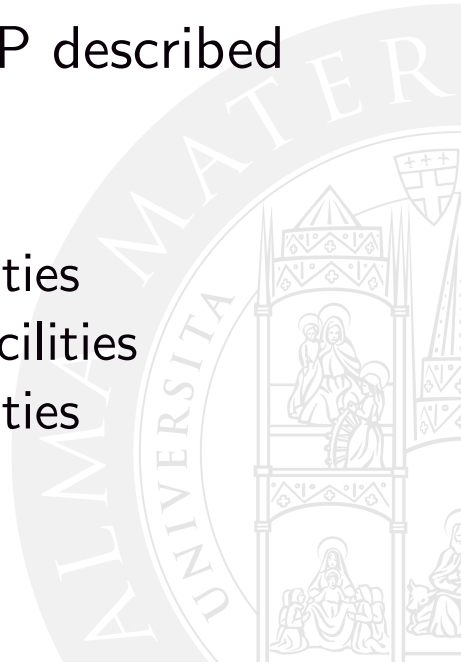
Project Map V

*:serialization-** support (de)serialization of terms, clauses, knowledge bases in YAML/JSON

- *:serialization-core* (de)serialization of terms and clauses
- *:serialization-theory* (de)serialization of knowledge bases and theories

*:dsl-** incrementally support the Kotlin-based DSL for LP described in [4], aimed at blending LP, FP, and OOP

- *:dsl-core* basic DSL for building terms/clauses in Kotlin
- *:dsl-unify* extension of the DSL including *:unify* facilities
- *:dsl-theory* extension of the DSL including *:theory* facilities
- *:dsl-solve* extension of the DSL including *:solve* facilities



Project Map VI

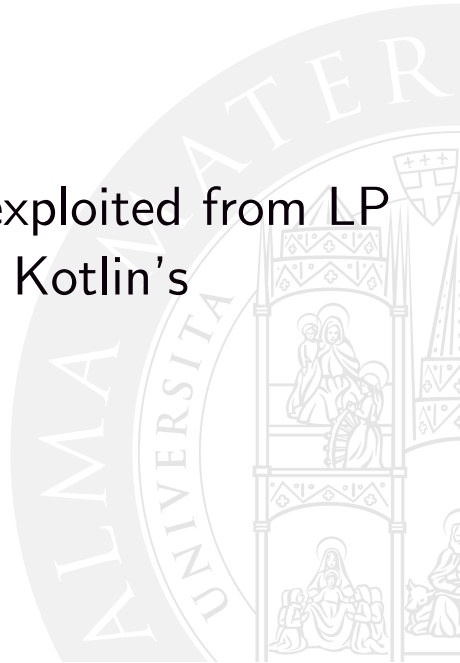
`:repl` command-line interface for Prolog

`:ide` JavaFX-based GUI for Prolog (customisable)

`:io-lib` Prolog ISO compliant Prolog library for I/O

`:oop-lib` Prolog library for OOP interoperability

- essentially, lets Kotlin's and Java's OOP facilities be exploited from LP
- only JVM is currently supported, due to limitations in Kotlin's reflection API



Next in Line...

Main Abstractions from the *:core* Module I

Terms and Clauses

Immutable data structures for terms and clauses in-memory representation & manipulation

Substitutions

Immutable data structures representing variables assignments, applicable to terms/clauses

Operators and Operators Sets

Immutable data structures for representing logic operators and their esembles

Main Abstractions from the *:core* Module II

Formatters

Functional objects aimed at converting terms/clauses into strings of customisable format

Visitors

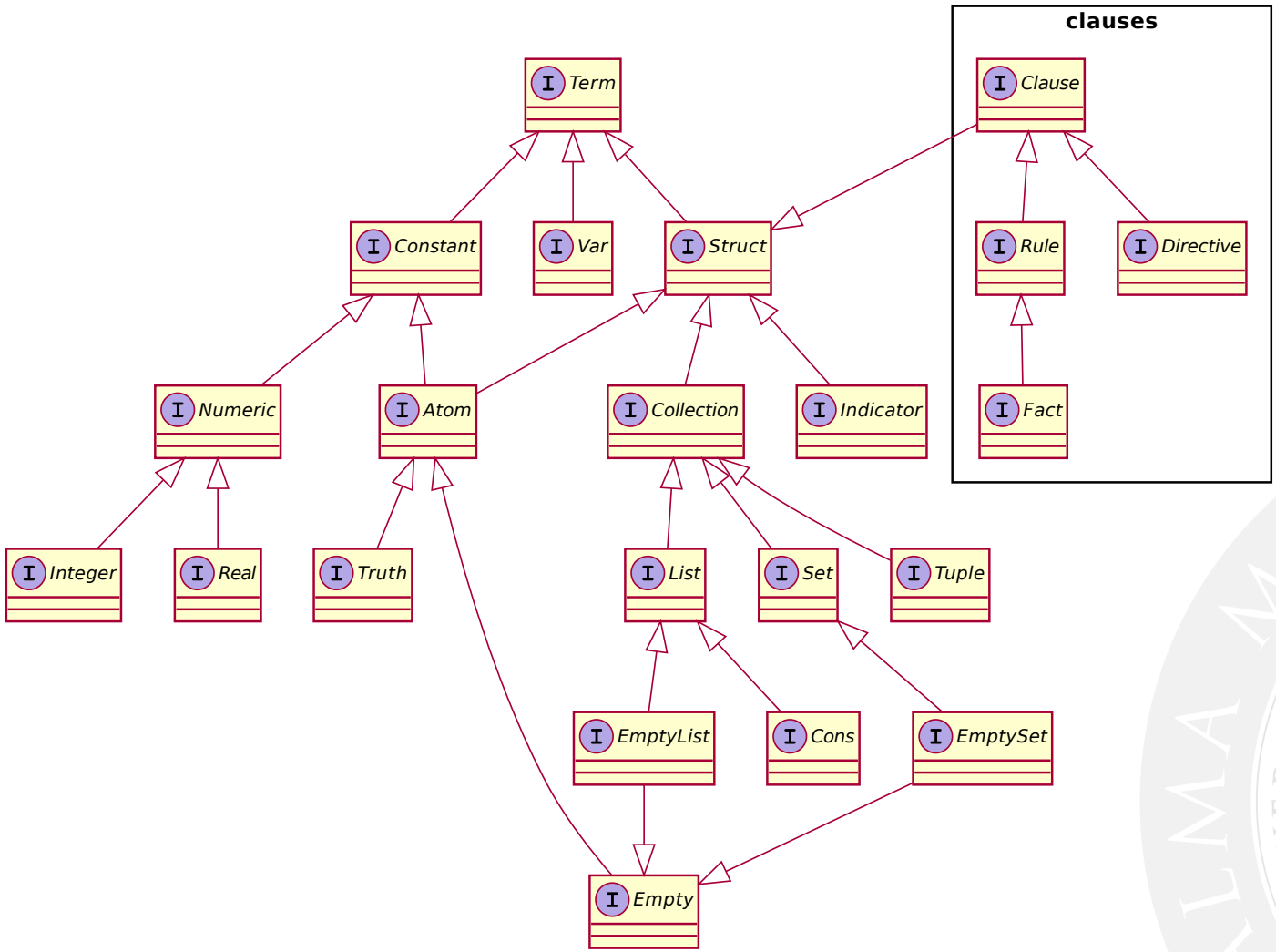
Functional objects aimed at easing type-dependent algorithms writing

Exceptions

Base exception types extensively exploited in all whole 2P-Kt *project*

Focus on...

Term Hierarchy I



Term Hierarchy II

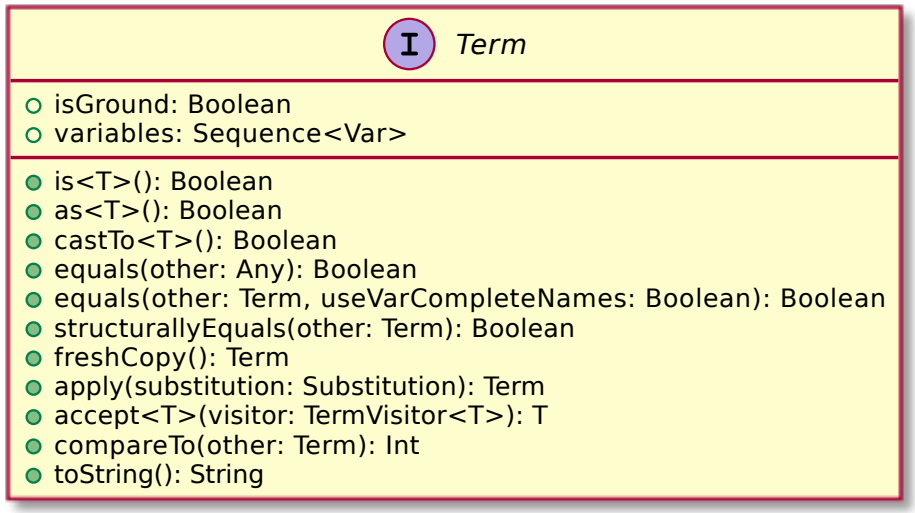
Common Conventions

- Only interfaces are publicly available, no classes
- Immutable design: all terms are immutable data structures
- The hierarchy is a DAG, not a tree
- Terms are totally ordered, as the are `Comparable` among each others
- We call "term" any object which (indirectly) implements `Term`
  - ! there including clauses, i.e. instances of `Clause`

The Term type I

The Term type

Base type for all terms



The Term type II

`is⟨SubType⟩: Boolean` checks whether the current term is an instance of ⟨*SubType*⟩ or not

`as⟨SubType⟩(): ⟨SubType⟩` casts the current term to ⟨*SubType*⟩ or returns null if not possible

`castTo⟨SubType⟩(): ⟨SubType⟩` casts the current term to ⟨*SubType*⟩ or throws an exception if not possible

`freshCopy(): Term` returns a deep copy of the current term where all `Variables` have been refreshed

`apply(Substitution): Term` applies the provided `Substitution` to the current `Term` or throws a `SubstitutionApplicationException` in case a failed `Substitution` has been provided

`equals(Term, Boolean): Boolean` checks whether the provided `Term` is deeply equal to the current one, letting the caller choose whether to compare variables by simple or by complete name

The Term type III

`equals(Any?): Boolean` checks whether the provides object is a `Term` is deeply equal to the current one, comparing variables by complete names

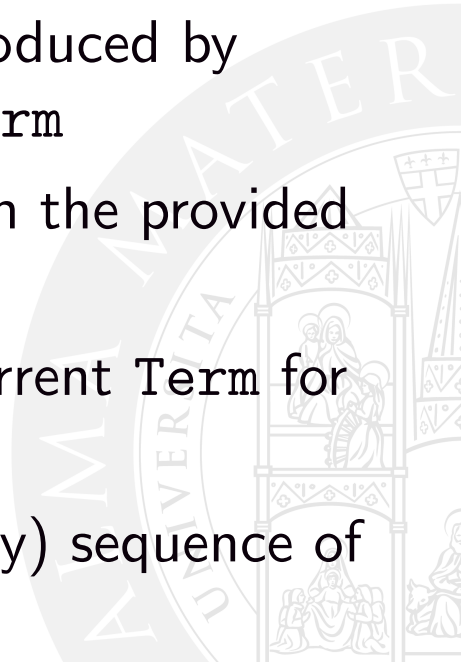
`structurallyEquals(Term): Boolean` checks whether the provided `Term` is deeply equal to the current one, in such a way that all variables are considered equal

`accept<T>(TermVisitor<T>): T` returns the object produced by letting the provided `TermVisitor` visit the current `Term`

`compareTo(Term): Int` compares the current `Term` with the provided one, according to the total ordering on `Terms`

`toString(): String` returns a representation of the current `Term` for debugging purposes

`variables: Sequence<Var>` returns the (possibly empty) sequence of variables contained in the current `Term`



The `Term` type IV

! duplicates are *not* removed

`isGround: Boolean` returns `true` if the current `Term` contains no `Variable`

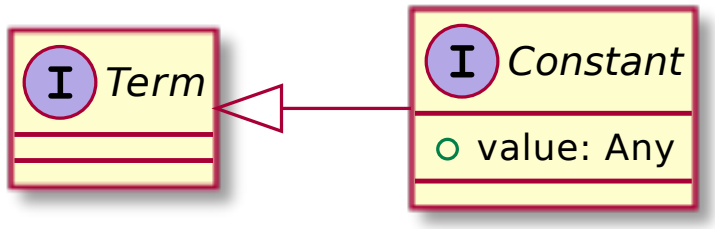
The Term type V

```
val a: Term = Atom.of("a") // a
val one: Term = Integer.of(1) // 1

val X: Term = Var.of("X") // X
val Y: Term = Var.of("Y") // Y

val f: Term = Struct.of("f", X, a, one, Y) // f(X, a, 1, Y)
val g: Term = Struct.of("g", a, one) // g(a, 1)

println(f.isGround) // false
println(g.isGround) // true

println(f.variables.toList()) // [X, Y]
println(g.variables.toList()) // []
```

Main Sorts of Terms I

The Constant type is a sub-type of Term

Base type for all constant terms (namely, strings and numbers)



`value: Any` returns the value of this `Constant`

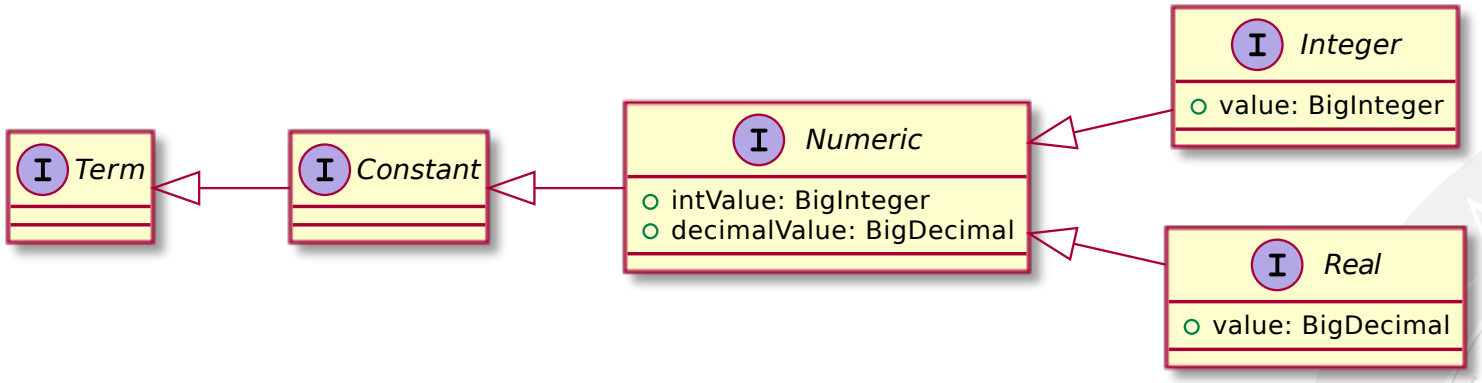
About Constants

- `Constants` are inherently ground
- Their `isGround` property always return `true`
- Their `variables` property always return an empty sequence

Main Sorts of Terms II

The Numeric type is a sub-type of Constant

Base type for all numeric terms (namely, reals and integers)



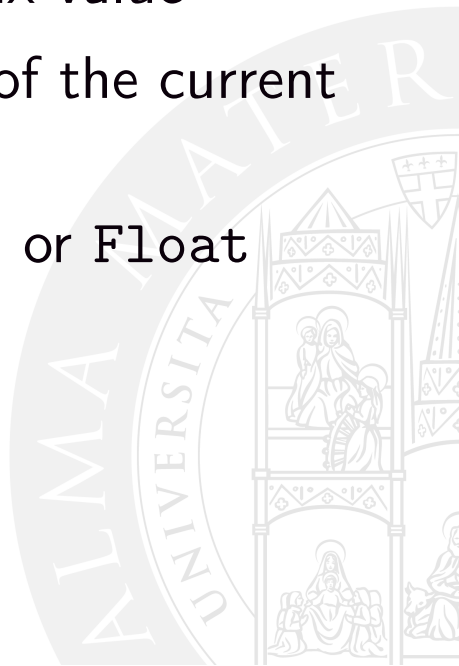
Main Sorts of Terms III

`intValue: BigInteger` returns the integer value of the current number

- `BigIntegers` can then be converted into `Int`, `Short`, etc.
- unlimited amount of digits → no overflow, no max value

`decimalValue: BigDecimal` returns the decimal value of the current number

- `BigDecimals` can then be converted into `Double` or `Float`
- i.e. a real number with arbitrary precision



Main Sorts of Terms IV

The Integer type is a sub-type of Numeric

Type for all terms representing an integer number

`value: BigInteger` returns `this.intValue`

- BigIntegers can then be converted into Int, Short, etc.
- unlimited amount of digits → no overflow, no max value

```
val i = Integer.of(1)

val iValue: BigInteger = i.value // BigInteger.of(1)
val iIntValue: BigInteger = i.intValue // BigInteger.of(1)
val iDecValue: BigDecimal = i.decimalValue // BigDecimal.of(1.0)

val iInt: Int = i.value.toInt() // 1
```

Main Sorts of Terms V

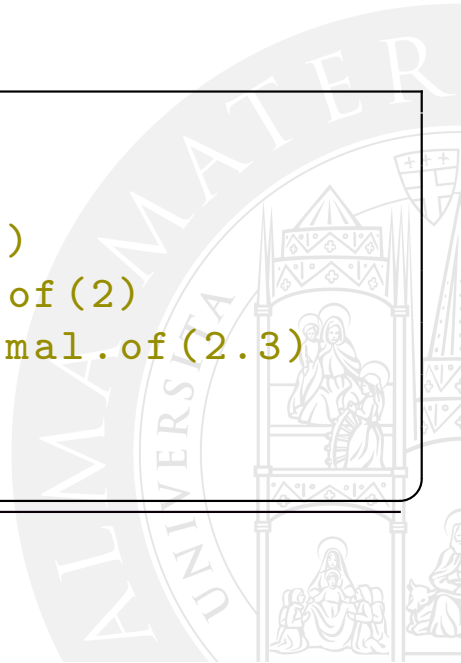
The Real type is a sub-type of Numeric

Type for all terms representing a real number

`value: BigDecimal` returns `this.decimalValue`

- BigDecimals can then be converted into Double or Float
- i.e. a real number with arbitrary precision

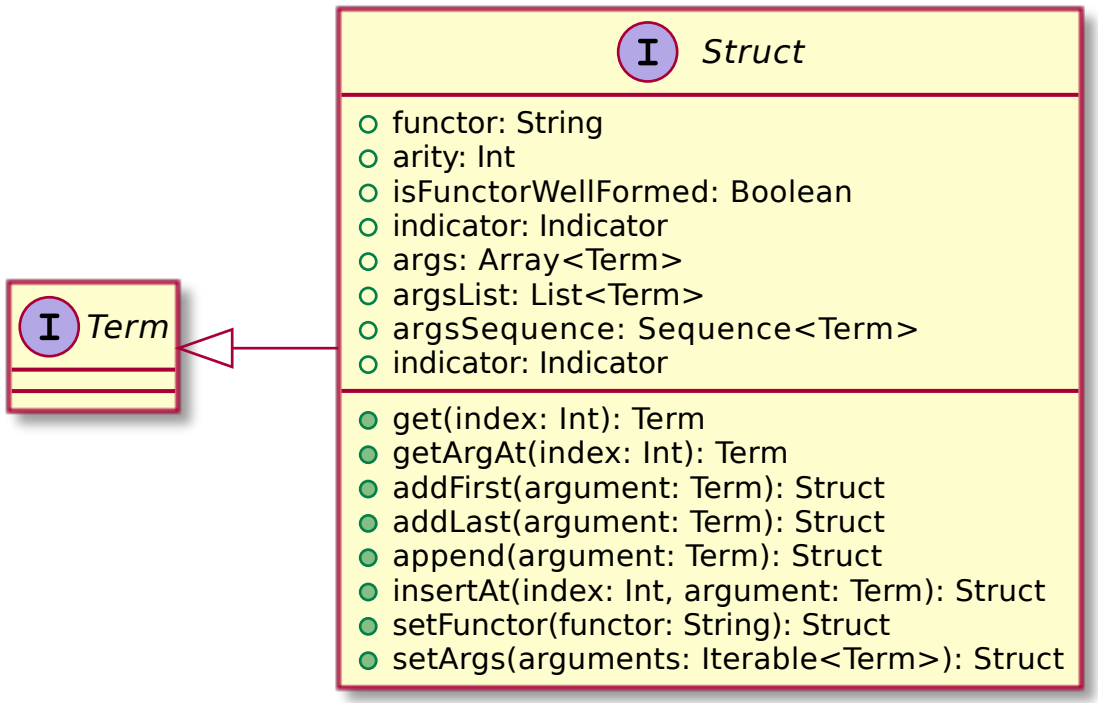
```
val r = Real.of(2.3)

val rValue: BigDecimal = i.value // BigDecimal.of(2.3)
val rIntValue: BigInteger = i.intValue // BigInteger.of(2)
val rDecValue: BigDecimal = i.decimalValue // BigDecimal.of(2.3)

val rDouble: Double = r.value.toDouble() // 2.3
```



Main Sorts of Terms VI

The Struct type is a sub-type of Term

Type for all terms of the form $f(t_1, \ldots, t_N)$, i.e. composed by a funtor string f and N terms $t_1, \ldots, t_N$ called arguments, where $N \geq 0$ is called arity



Main Sorts of Terms VII

`functor: String` returns f

`arity: Int` returns N

`args: Array<Term>` returns an array containing $t_1, \ldots, t_N$

`argsList: List<Term>` returns a list containing $t_1, \ldots, t_N$

`argsSequence: Sequence<Term>` returns a sequence containing $t_1, \ldots, t_N$

`isFunctorWellFormed: Boolean` returns true if the following condition hold for f

1. it begins with a lower case letter
2. it only contains letters (either lower or upper case), digits, or underscores

`toString(): String` returns f iff $N = 0$, otherwise a string of the form

$$f(t_1.\texttt{toString()}, \ldots, t_N.\texttt{toString()})$$

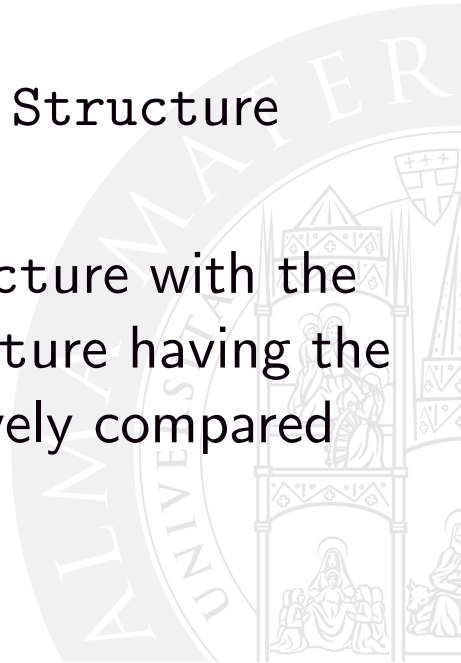
Main Sorts of Terms VIII

- in both cases, *f* is wrapped within single quotes iff `isFunctorWellFormed` is false

`equals(Any?): Boolean` compares this `Structure` with the provided object, returning true if the latter is a `Structure` having the same functor, arity, and arguments (which are recursively compared via `equals(Any?)`)

`hashCode(): Int` computes an hash code value for this `Structure` which is coherent w.r.t. `equals(Any?)`

`equals(Term,Boolean): Boolean` compares this `Structure` with the provided object, returning true if the latter is a `Structure` having the same functor, arity, and arguments (which are recursively compared via `equals(Term,Boolean)`)



Main Sorts of Terms IX

`structurallyEquals(Term): Boolean` compares this `Structure` with the provided object, returning true if the latter is a `Structure` having the same functor, arity, and arguments (which are recursively compared via `structurallyEquals(Term)`)

`indicator: Indicator` returns the indicator corresponding to the current `Struct`

- i.e. the `Structure` of the form `'/'(`f`,` N`)`

`get(Int): Term` returns an argument of the current `Struct` given its index

`getArgAt(Int): Term` an alias for `get(Int)`

`addFirst(Term): Term` returns a novel `Struct` of the form $f(t^*, t_1, \ldots, t_N)$, where t^* is the `Term` provided as argument

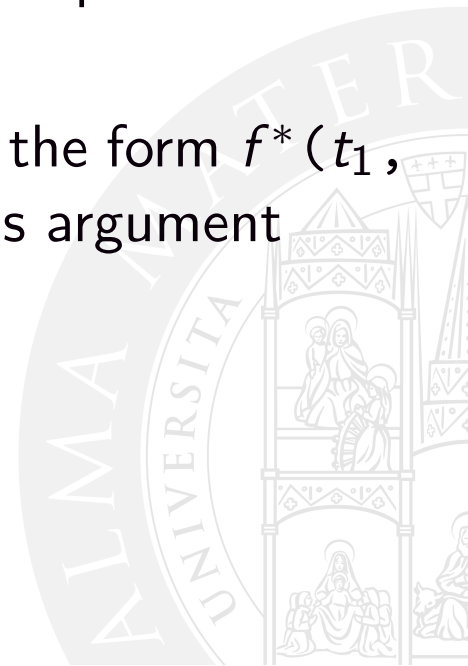
`addLast(Term): Term` returns a novel `Struct` of the form $f(t_1, \ldots, t_N, t^*)$, where t^* is the `Term` provided as argument

Main Sorts of Terms X

`append(Term): Term` an alias for `addLast(Term)`

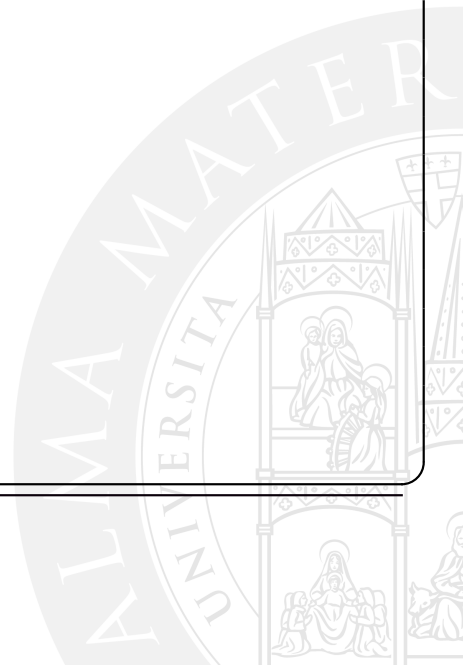
`insertAt(Int,Term): Term` returns a novel `Struct` of the form $f(t_1, \ldots, t_{i-1}, t^*, t_{i+1}, \ldots, t_N)$, where t^* is the `Term` provided as argument

`setFunctor(String): Term` returns a novel `Struct` of the form $f^*(t_1, \ldots, t_N)$, where f^* is the functor `String` provided as argument



Main Sorts of Terms XI

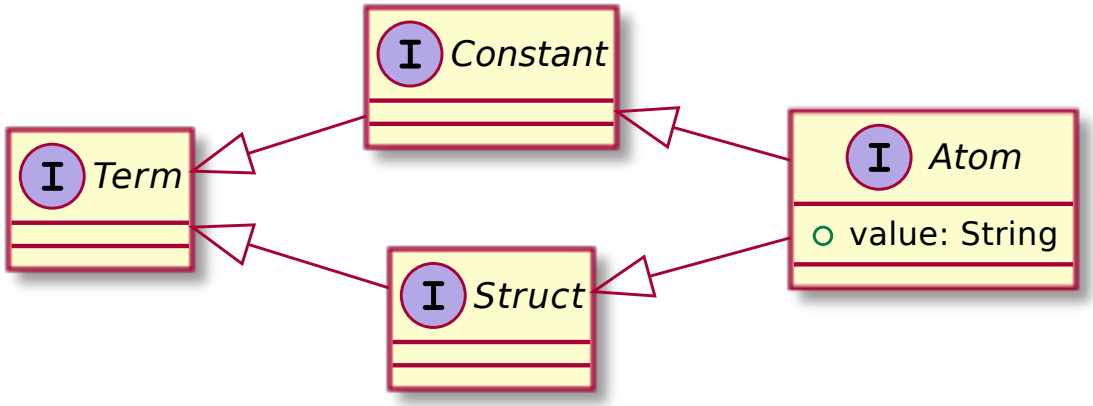
```
val a = Atom.of("a") // a
val one = Integer.of(1) // 1

val X = Var.of("X") // X
val Y = Var.of("Y") // Y

val ga1 = Struct.of("g", a, one) // g(a, 1)

println(ga1.functor) // g
println(ga1.arity) // 2
println(ga1.argsList) // [a, 1]
println(ga1.toString()) // g(a, 1)

val Fa1 = ga1.setFunctor("F") // 'F'(a, 1)
val FXa1 = fa1.addFirst(X) // 'F'(X, a, 1)
val FXa1Y = fa1.addLast(Y) // 'F'(X, a, 1, Y)

println(FXa1Y.functor) // F
println(FXa1Y.arity) // 4
println(FXa1Y.argsList) // [X_1, a, 1, Y_2]
println(FXa1Y.toString()) // 'F'(X_1, a, 1, Y_2)
```



Main Sorts of Terms XII

The Atom type is a sub-type of both Struct and Constant

Type for all constant 0-ary structures (i.e., with no arguments), representing strings



Main Sorts of Terms XIII

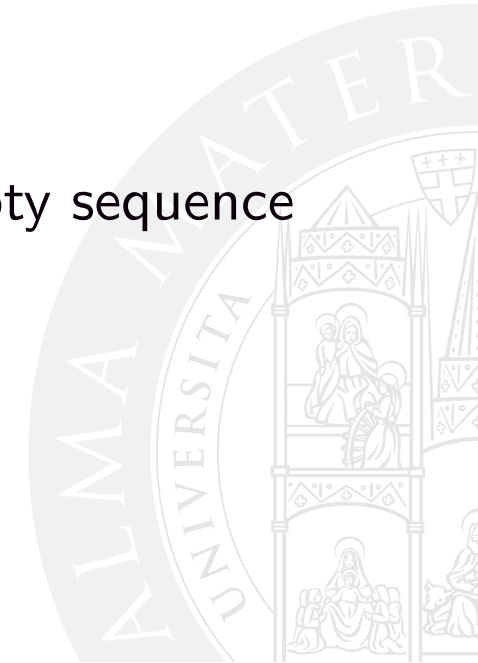
`value: String` returns `this.functor`

`arity: Int` must return 0

`args: Array<Term>` must return the empty array

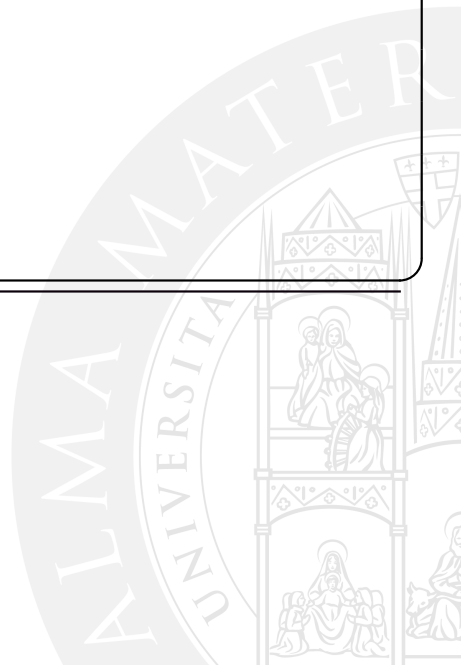
`argsList: List<Term>` must return the empty list

`argsSequence: Sequence<Term>` must return the empty sequence



Main Sorts of Terms XIV

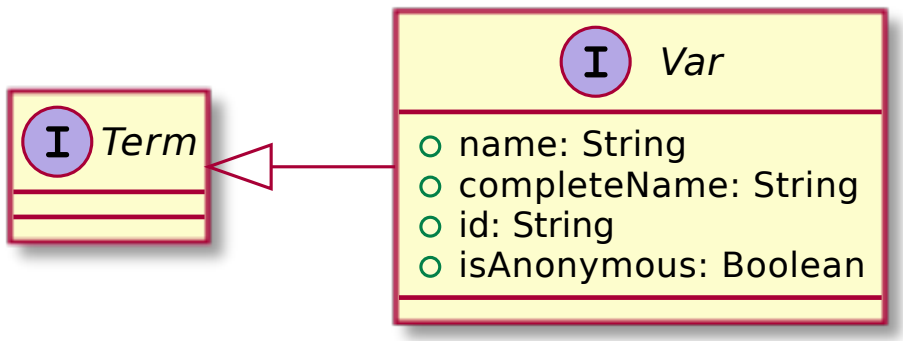
```
val a = Atom.of("a") // a

println(a.value) // a
println(a.toString()) // a

val anAtom = Atom.of("an atom") // 'an atom'

println(a.value) // an atom
println(a.toString()) // 'an atom'
```



Main Sorts of Terms XV

The Var type is a sub-type of Term

Type for all variable terms, representing named placeholders for other terms.



Main Sorts of Terms XVI

About Variable names

- Variables are identified by complete name
- Variables' complete names are Strings of the form

  Name_Identifier

  where

  Name is the variable's simple name

  Identifier is the variable's identifier, i.e. an implementation-specific string aimed at distinguishing Variables having the same simple name
- Variables whose simple name is '_' are called anonymous

`name:` String returns *Name*

`completeName:` String returns *Name_Identifier*

Main Sorts of Terms XVII

`id: String` returns *Identifier*

`isAnonymous: Boolean` returns true iff *Name* = '_'

`toString(): Boolean` returns this `this.completeName`

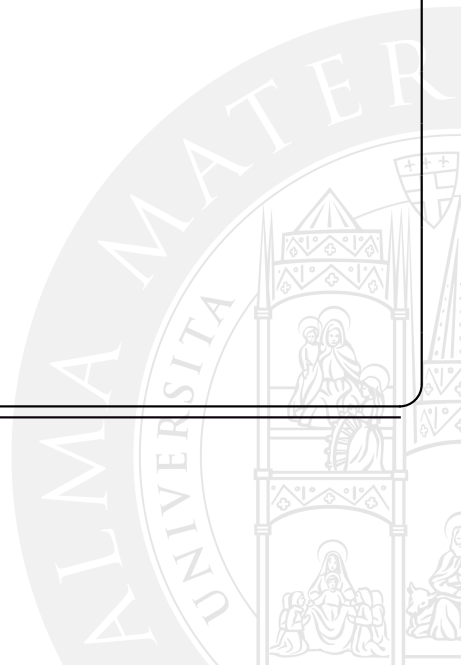
`equals(Any?): Boolean` compares this `Variable` with the provided object, returning true if the latter is a `Variable` having the same complete name

`equals(Term,Boolean): Boolean` compares this `Variable` with the provided object, returning true if the latter is a `Variable` having the same complete (resp. simple) name, assuming that the provided `Boolean` is true (resp. false)

`structurallyEquals(Term): Boolean` compares this `Variable` with the provided object, returning true if the latter is a `Variable`

Main Sorts of Terms XVIII

```
val X = Var.of("X")

println(X.isAnonymous) // false
println(X.name) // X
println(X.completeName) // X_1
println(X.id) // 1

val whatever = Var.anonymous()

println(whatever.isAnonymous) // true
println(whatever.name) // _
println(whatever.completeName) // __2
println(whatever.id) // 2
```



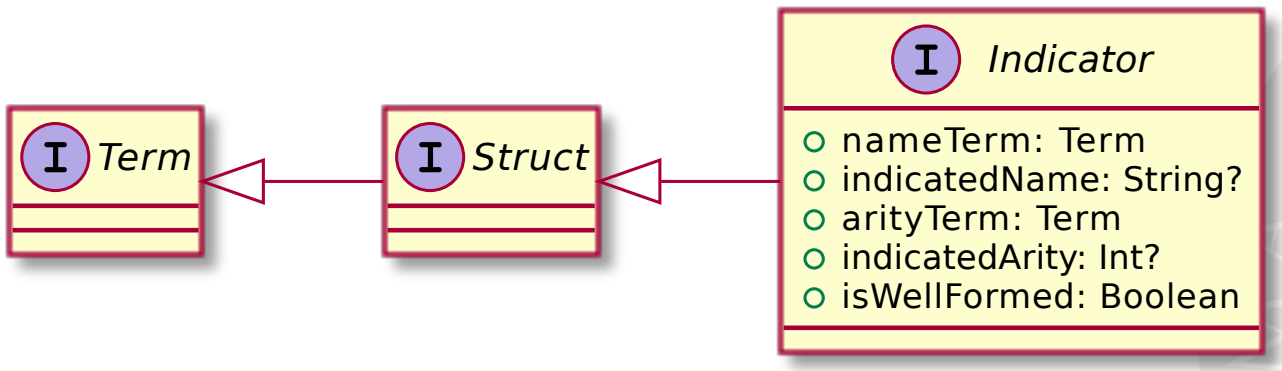
Main Sorts of Terms XIX

The Indicator type is a sub-type of Struct

Type for all binary structures of the form

$$'/'(f,\ n)$$

where f, n are arbitrary terms. If f is an atom and n is a non-negative integer, then the indicator is considered *well formed*



Main Sorts of Terms XX

`nameTerm: Term` returns *f*, shortcut for `get(0)`

`indicatedName: String?` if *f* is an atom, returns *f*`.value`, otherwise `null`

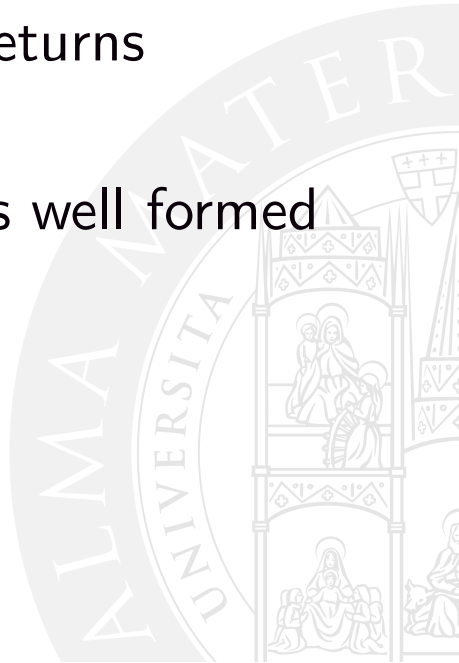
`arityTerm: Term` returns *n*, shortcut for `get(1)`

`indicatedArity: Int?` if *n* is a non-negative integer, returns *n*`.intValue.toInt()`, otherwise `null`

`isWellFormed: Boolean` returns `true` if the indicator is well formed

`functor: String` must return `"/"`

`arity: String` must return 2



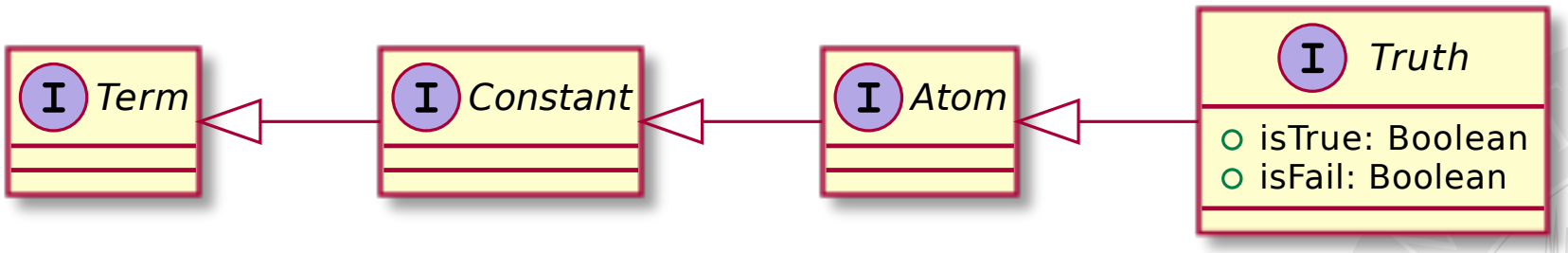
Main Sorts of Terms XXI

```
val fa1 = Struct.of("f", Atom.of("a"), Integer.ONE) // f(a, 1)

val f2: Indicator = fa1.indicator // f/2

val f2Name: Term = f2.nameTerm: // Atom.of("f")
val f2Arity: Term = f2.arityTerm // Integer.of(2)
val f2N: String? = f2.indicatedName // "f"
val f2A: Int? = f2.indicatedArity: // 2

val fN: Indicator = Indicator.of(Atom.of("f"), Var.of("N")) // f/N

val fNName: Term = fN.nameTerm: // Atom.of("f")
val fNArity: Term = fN.arityTerm // Var.of("N")
val fNN: String? = fN.indicatedName // "f"
val fNA: Int? = fN.indicatedArity: // null
```

Main Sorts of Terms XXII

The Truth type is a sub-type of Atom

Type for special atoms representing either tautology or contradiction. It has only three admissible values: `true` (tautology), `false`, and `fail` (contradictions).

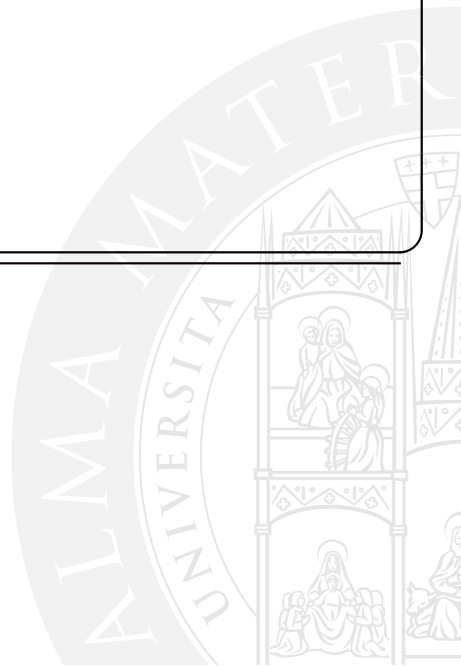


`isTrue: Boolean` returns `true` if the atom is a tautology

`isFail: Boolean` returns `true` if the atom is a contradiction

Main Sorts of Terms XXIII

```
val t = Truth.TRUE
val f1 = Truth.FALSE
val f2 = Truth.FAIL

println(t.value) // true
println(f1.value) // false
println(f2.value) // fail

println(t.isTrue + " " + t.isFail) // true false
println(f1.isTrue + " " + f1.isFail) // false true
println(f2.isTrue + " " + f2.isFail) // false true
```



Collections I

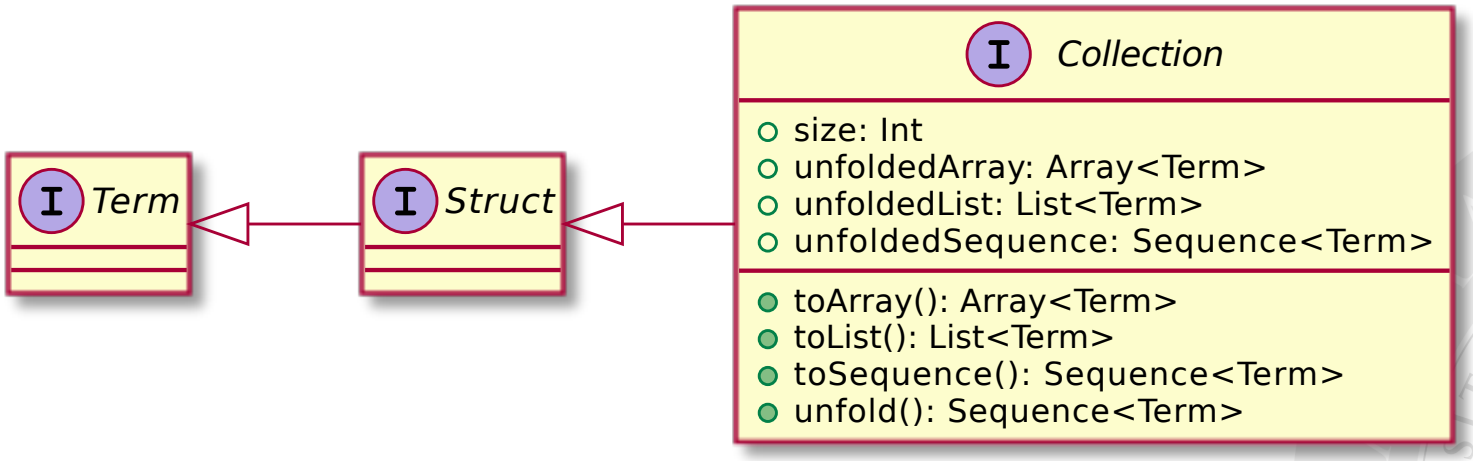
How to fold items with structures

- Suppose you want to collect terms $t_1, \ldots, t_N \ldots$
- ... using binary structures whose functor is f
- Two folding strategies: with or without a termination term T
  - explicit termination: $f(t_1,\ f(t_2,\ \ldots f(t_N,\ T)\ \ldots))$
  - implicit termination: $f(t_1,\ f(t_2,\ \ldots f(t_{N-1},\ t_N)\ \ldots))$
- A particular collection is therefore determined by:
  - the particular choice of f
  - the particular folding strategy adopted
  - the particular termination term T, if any

Collections II

The Collection type is a sub-type of Struct

Base type for all right-recursive structures aimed at containing an unlimited amount of terms.



Collections III

`unfoldedSequence: Sequence<Term>` in general, returns a sequence containing the terms $t_1, \ldots, t_N$, plus T in case of explicit folding strategy

`unfoldedList: List<Term>` in general, returns a list containing the terms $t_1, \ldots, t_N$, plus T in case of explicit folding strategy

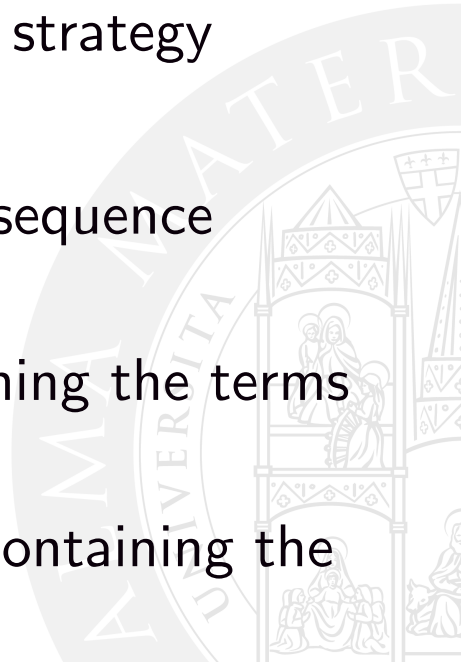
`unfoldedArray: Array<Term>` in general, returns an array containing the terms $t_1, \ldots, t_N$, plus T in case of explicit folding strategy

`size: Int` in general, returns N

`toSequence(): Sequence<Term>` in general, returns a sequence containing the terms $t_1, \ldots, t_N$

`toList(): List<Term>` in general, returns a list containing the terms $t_1, \ldots, t_N$

`toArray(): Array<Term>` in general, returns an array containing the terms $t_1, \ldots, t_N$



Collections IV

`unfold(): Sequence<Term>` in general, returns a sequence containing the terms $f(t1, f(t_2, \ldots))$, $f(t_2, \ldots)$, $\ldots$, plus T in case of explicit folding strategy, or $f(t_{N-1}, t_N)$ otherwise

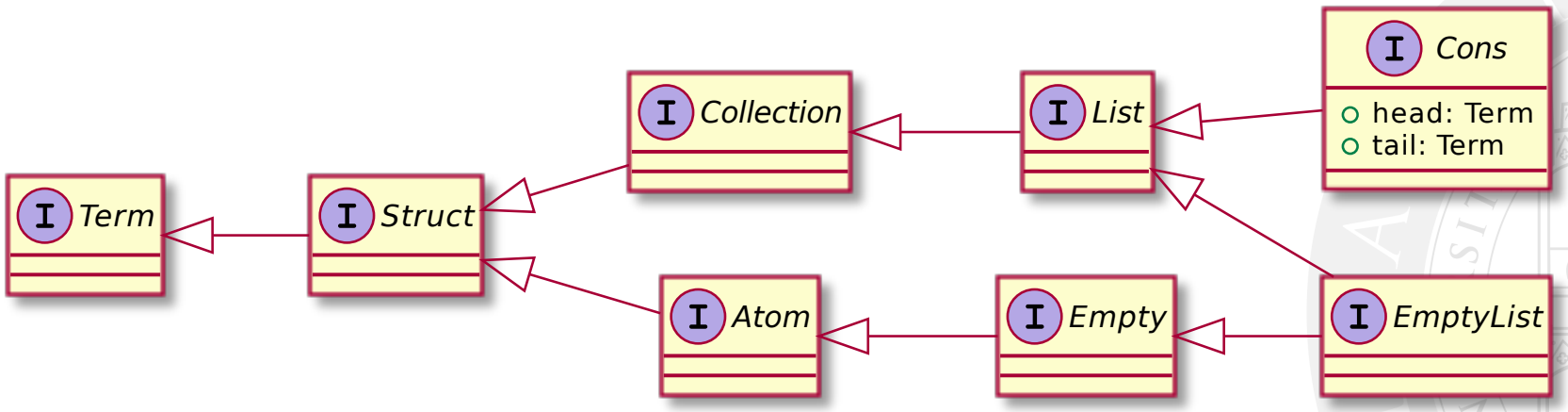
Collections V

The List type is a sub-type of Collection

Base type for logic lists, i.e. collections using '.' as functor and '[]' as the termination term. So all lists are terms of the form

$$'.'(t_1, '.'(t_2, \ldots '.'(t_N, T)\ldots)).$$

A list is considered well formed iff $T \equiv []$.



Collections VI

About logic lists

- There are two sub-types of `List`:
  - `Cons`, capturing all terms of the form `'.'`$(h,\ t)$ where both h and t are terms of any sort
  - `EmptyList`, which has only one value, namely the atom `'[]'`
- Logic lists are represented as strings using the following notation:

$$[t_1,\ t_2,\ \ldots t_n\ |\ T]$$

  - where '$|\ T$' may be omitted in case $T \equiv []$

`isWellFormed: Boolean` returns `true` if the list is well formed

`last: Term` returns T

`isCons: Boolean` returns `true` if the list is not empty

`isEmptyList: Boolean` returns `true` if the list is empty

Collections VII

`unfoldedSequence: Sequence<Term>` returns a sequence containing the terms $t_1, \ldots, t_N$, []

`unfoldedList: List<Term>` returns a list containing the terms $t_1, \ldots, t_N$, []

`unfoldedArray: Array<Term>` returns an array containing the terms $t_1, \ldots, t_N$, []

`size: Int` returns N if the list is well formed, or $N + 1$ otherwise

`toSequence(): Sequence<Term>` returns a sequence containing the terms $t_1, \ldots, t_N$, plus T iff $T \not\equiv []$

`toList(): List<Term>` returns a list containing the terms $t_1, \ldots, t_N$, plus T iff $T \not\equiv []$

`toArray(): Array<Term>` returns an array containing the terms $t_1, \ldots, t_N$, plus T iff $T \not\equiv []$

Collections VIII

`unfold(): Sequence<Term>` returns a sequence containing the terms
$'.'(t1, '.'(t_2, \ldots)), '.'(t_2, \ldots), \ldots, T$

```
val l1 = Cons.of(
    Atom.of("a"),
    Cons.of(
        Atom.of("b"),
        Cons.of(Atom.of("c"), Empty.list())
    )
) // .(a, .(b, .(c, [])))

println(l1.head) // a
println(l1.tail) // [b, c]

val l2 = List.of(Atom.of("a"), Atom.of("b"), Atom.of("c")) // [a, b, c]

println(l1 == l2) // true
println(l1.size) // 3
l2.unfoldedSequence.forEach { println(it) } // a, b, c, []
l2.toSequence().forEach { println(it) } // a, b, c

val l3 = List.from(Atom.of("a"), Atom.of("b"), last=Var.of("T")) // [a, b | T]
l3.unfoldedSequence.forEach { println(it) } // a, b, T
l3.toSequence().forEach { println(it) } // a, b, T
```

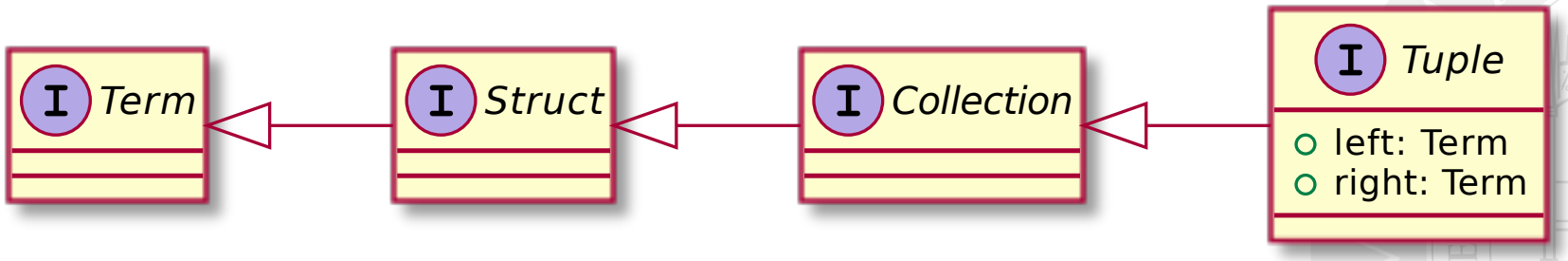
Collections IX

The Tuple type is a sub-type of Collection

Base type for logic conjunctions, i.e. collections using ',' as functor and an implicit termination strategy. So all tuples are terms of the form

$$',' (t_1, ',' (t_2, \ldots ',' (t_{N-1}, t_N) \ldots)).$$

Notice that tuples always contain at least 2 terms.



Collections X

About logic tuples

- Logic tuples are represented as strings using the following notation:

$$(t_1,\ t_2,\ \ldots t_n)$$

`unfoldedSequence: Sequence<Term>` returns a sequence containing the terms $t_1, \ldots, t_N$

`unfoldedList: List<Term>` returns a list containing the terms $t_1, \ldots, t_N$

`unfoldedArray: Array<Term>` returns an array containing the terms $t_1, \ldots, t_N$

`size: Int` returns N

`toSequence(): Sequence<Term>` returns `this.unfoldedSequence`

`toList(): List<Term>` returns `this.unfoldedList`

Collections XI

`toArray(): Array<Term>` returns `this.unfoldedArray`

`unfold(): Sequence<Term>` returns a sequence containing the terms

$$\texttt{','}(t1,\ \texttt{','}(t_2,\ \ldots)),\ \texttt{','}(t_2,\ \ldots),\ \ldots,\ \texttt{','}(t_{N-1},\ t_N)$$

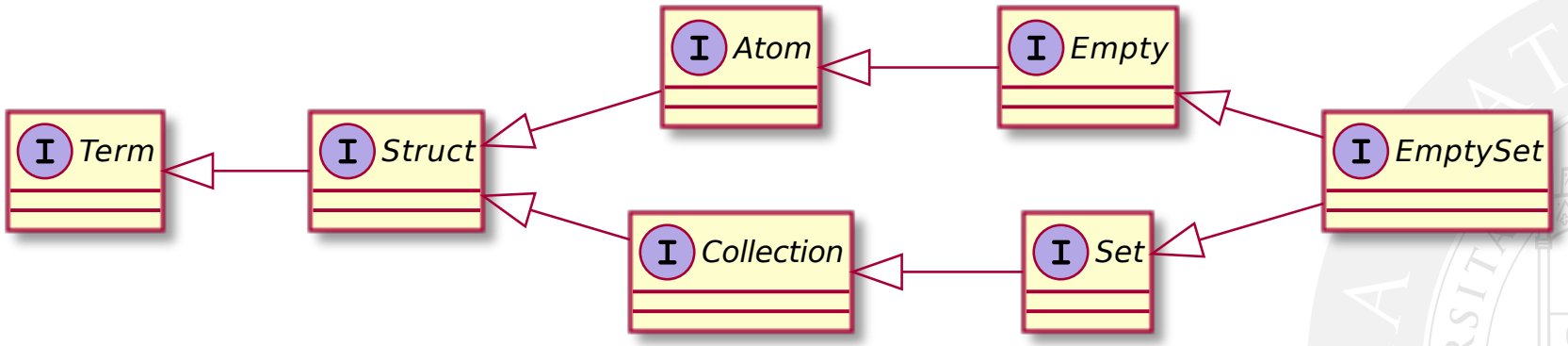
Collections XII

```
val l1 = Struct.of(",",
    Atom.of("a"),
    Struct.of(",",
        Atom.of("b"),
        Atom.of("c")
    )
) // ','(a, ','(b, c))

println(l1.left) // a
println(l1.right) // (b, c)

val l2 = Tuple.of(Atom.of("a"), Atom.of("b"), Atom.of("c")) // (a,
    b, c)

println(l1 == l2) // true
println(l2.size) // 3
l2.toSequence().forEach { println(it) } // a, b, c
```

Collections XIII

The Set type is a sub-type of Collection

Base type for logic sets, i.e. a particular sort of collection using {} as functor, which may either be empty or contain a single term—which may possibly be a tuple.



Collections XIV

About logic sets

- There are two sub-types of `Set`:
  - `Set`, capturing all terms of the form `'{}'(`x`)` where x is a term of any sort
  - `EmptySet`, which has only one value, namely the atom `'{}'`
- Logic sets are represented as strings using the following notations:
  - $\{\}$ in case of empty set
  - $\{t_1, \ldots t_N\}$ in case $x \equiv (t_1, \ldots, t_N)$, i.e. if x is a tuple
  - $\{x\}$ otherwise

`unfoldedSequence: Sequence<Term>` returns a sequence containing the terms $t_1, \ldots, t_N$ if x is a tuple, just x otherwise, or nothing if the set is empty

Collections XV

`unfoldedList: List<Term>` returns a list containing the terms $t_1, \ldots, t_N$ if x is a tuple, just x otherwise, or nothing if the set is empty

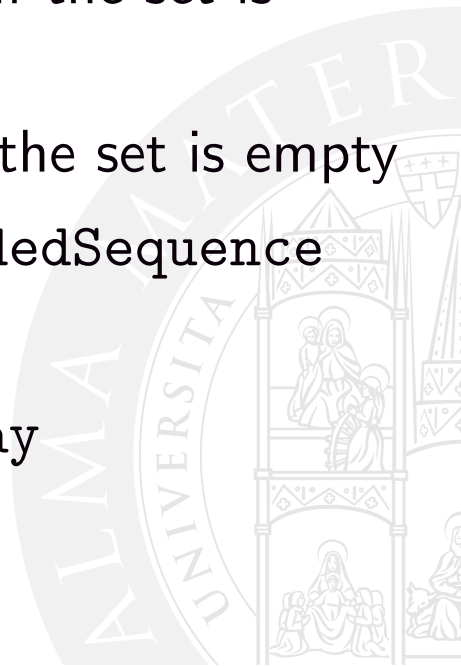
`unfoldedArray: Array<Term>` returns an array containing the terms $t_1, \ldots, t_N$ if x is a tuple, just x otherwise, or nothing if the set is empty

`size: Int` returns N if x is a tuple, 1 otherwise, or 0 if the set is empty

`toSequence(): Sequence<Term>` returns `this.unfoldedSequence`

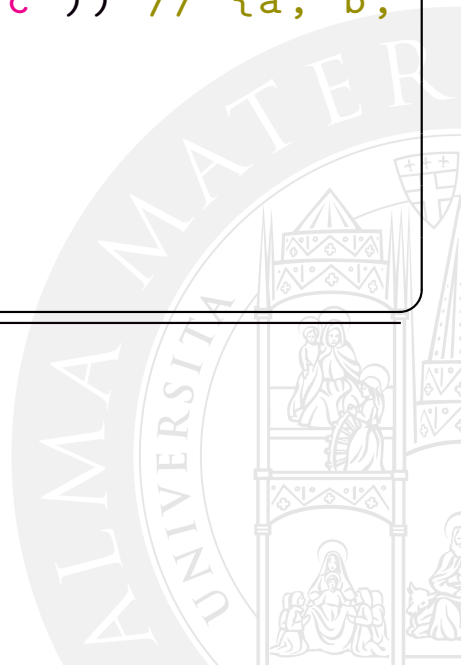
`toList(): List<Term>` returns `this.unfoldedList`

`toArray(): Array<Term>` returns `this.unfoldedArray`



Collections XVI

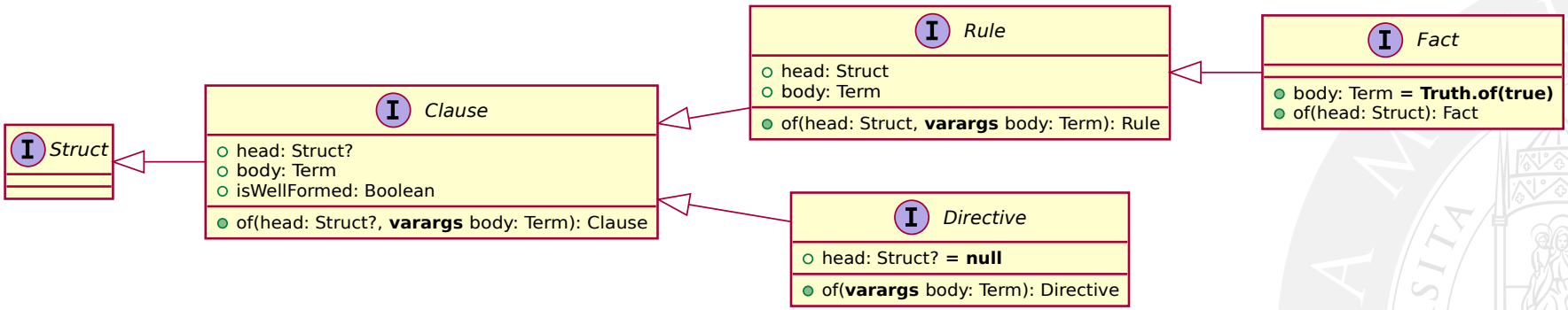
```
val l1 = Struct.of("{}",
    Tuple.of(Atom.of("a"), Atom.of("b"), Atom.of("c"))
) // '{}'( ','(a, ','(b, c)) )

val l2 = Set.of(Atom.of("a"), Atom.of("b"), Atom.of("c")) // {a, b,
     c}

println(l1 == l2) // true
println(l2.size) // 3
l2.toSequence().forEach { println(it) } // a, b, c
```



Clauses I

The `Clause` type is a sub-type of `Struct`

Base type for all structures aimed at representing Horn clauses. They all use `':-'` as functor and carry either 1 or 2 arguments, named *head* and *body*.



Clauses II

About clauses

- There are three sub-types of `Clause`:
  - `Rule`, capturing all structures of the form `':-'`$(h,\ b)$ where h is a `Structure` and b is a term of any sort
    - `Fact`, capturing all rules of the form `':-'`$(h,$ `true`$)$, where $b \equiv$ `true`
  - `Directives`, capturing all structures of the form `':-'`(b) where b is a term of any sort
- Logic clauses are represented as strings using the following notations:
  - h `:-` $b_1,\ \ldots,\ b_N.$ in case of rules where $b \equiv (b_1,\ \ldots,\ b_N)$
  - h `:-` $b.$ in case of rules
  - $h.$ in case of facts
  - `:-` $b_1,\ \ldots,\ b_N.$ in case of directives where $b \equiv (b_1,\ \ldots,\ b_N)$
  - `:-` $b.$ in case of directives

Clauses III

`head: Struct?` returns *h* for rules, or `null` for directives

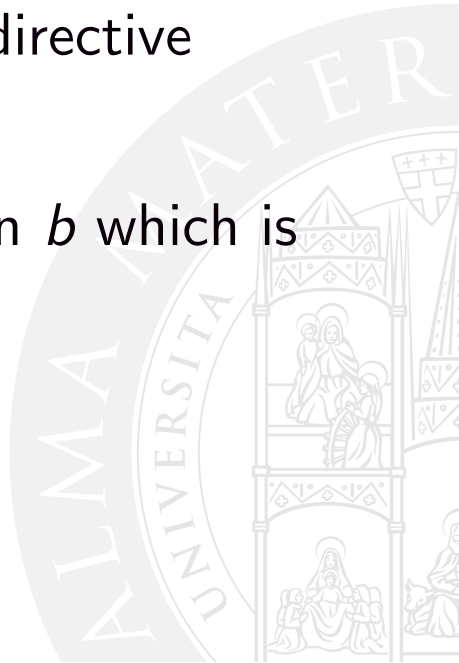
`body: Term` returns *b*

`isRule: Boolean` returns `true` if the clause is not a directive

`isDirective: Boolean` returns `true` if the clause is a directive

`isFact: Boolean` returns `true` if the clause is a fact

`isWellFormed: Boolean` returns `true` iff no sub-term in *b* which is interpretable as a logic goal is a `Variable`



Clauses IV

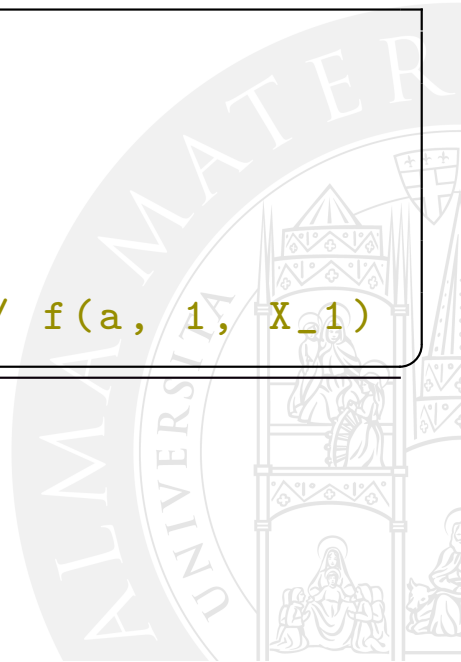
```
val rule = Rule.of(
    Struct.of("anchestor", Atom.of("abraham"), Atom.of("jacob")),
    Struct.of("parent", Atom.of("abraham"), Atom.of("isaac")),
    Struct.of("parent", Atom.of("isaac"), Atom.of("jacob"))
) // anchestor(abraham, jacob) :- parent(abraham, isaac), parent(isaac, jacob).

println(rule.head) // anchestor(abraham, jacob)
println(rule.body) // (parent(abraham, isaac), parent(isaac, jacob))

val fact = Fact.of(
    Struct.of("parent", Atom.of("abraham"), Atom.of("isaac"))
) // parent(abraham, isaac).

println(fact.head) // parent(abraham, isaac)
println(fact.body) // true

val directive = Directive.of(
    Struct.of("anchestor", Var.of("X"), Var.of("Y"))
) // :- anchestor(X, Y).

println(directive.head) // null
println(directive.body) // anchestor(X, Y)
```

Creating Terms via Static Factories I

Conventions on terms/clause creation in 2P-Kt:

- Terms and clauses can be created via static factory methods on interfaces

```
val atom1 = Atom.of("a") // a
val atom2 = Atom.of("a b") // 'a b'
val integer = Integer.of(1) // 1
val real = Real.of(2.3) // 2.3
val X = Var.of("X") // X_1
val struct = Struct.of("f", atom1, integer, X) // f(a, 1, X_1)
```



Creating Terms via Static Factories II

- Factory methods always instantiate the most specific type possible:

```
val arg1 = Numeric.of(1) // this actually creates an Integer
val arg2 = Numeric.of(2.3) // this actually creates a Real
val arg3 = Atom.of("[]") // this actually creates an EmptyList
val struct = Struct.of(".", arg1, Struct.of(".", arg2, arg3))
val list = struct as Cons // struct is actually a Cons
println(list) // [1, 2.3]
println(list.isWellFormed) // true, as the last item is
    EmptyList
```

Creating Terms via Static Factories III

Creating Integers

`Integer.of(`⟨*Kotlin Integer Type*⟩`)` creates an `Integer` out of a Kotlin integer

`Integer.of(BigInteger)` creates an `Integer` out of a `BigInteger`

`Integer.of(String,Int)` parses the string as an `Integer` with the provided base

`Integer.ZERO` constant for the `Integer` equal to 0

`Integer.ONE` constant for the `Integer` equal to 1

`Integer.MINUS_ONE` constant for the `Integer` equal to -1

Creating Terms via Static Factories IV

Creating Reals

`Real.of(`⟨*Kotlin Floating Type*⟩`)` creates a `Real` out of a Kotlin float

`Real.of(BigInteger)` creates a `Real` out of a `BigDecimal`

`Real.of(String)` parses the string as a `Real`

`Real.ZERO` constant for the `Real` equal to 0.0

`Real.ONE` constant for the `Real` equal to 1.0

`Real.MINUS_ONE` constant for the `Real` equal to -1.0

`Real.ONE_HALF` constant for the `Real` equal to 0.5

`Real.ONE_TENTH` constant for the `Real` equal to 0.1

Creating Terms via Static Factories V

Creating Atoms

`Atom.of(String)` creates an `Atom` out of a string

- returns a `Truth` if `"true"`, `"false"`, or `"fail"` is passed
- returns an `EmptyList` if `"[]"` is passed
- returns an `EmptySet` if `"{}"` is passed

Creating Terms via Static Factories VI

Creating Variables

`Var.of(String)` creates an `Variable` out of a string

- the string is considered as the `Variable`'s simple name
- ! there is no way to create a `Variable` with a particular complete name

`Var.anonymous()` creates an anonymous `Variable`

```
val X1 = Var.of("X")
val X2 = Var.of("X")

println(X1 == X2) // false!!!
```

Creating Terms via Static Factories VII

Creating Structures

`Struct.of(String,⟨Container⟩)` creates an `Structure` with the given functor and arguments

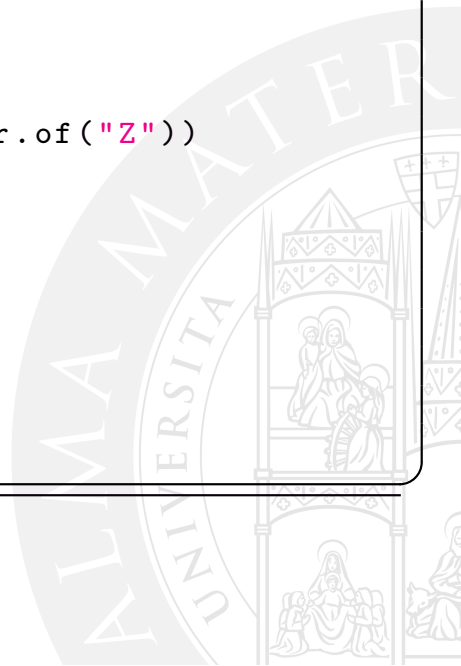
- where ⟨*Container*⟩ may be an iterable, a sequence, or a varargs of `Terms`

`Struct.template(String,Int)` creates an `Structure` with the given functor and arity, whose arguments are all anonymous variables

`Struct.fold(String,⟨Container⟩,Term?)` folds the terms in ⟨*Container*⟩ using the provided functor recursively, adopting either an explicit or implicit folding strategy depending on whether the last argument is null or not

Creating Terms via Static Factories VIII

```
Struct.template("f", 2)
// is a shortcut for
Struct.of("f", Var.anonymous(), Var.anonymous())

Struct.fold("f", arrayOf(Atom.of("a"), Atom.of("b"), Atom.of("c")), null)
// is a shortcut for
Struct.of("f",
    Atom.of("a"),
    Struct.of("f",
        Atom.of("b"),
        Atom.of("c")))

Struct.fold("f", arrayOf(Atom.of("a"), Atom.of("b"), Atom.of("c")), Var.of("Z"))
// is a shortcut for
Struct.of("f",
    Atom.of("a"),
    Struct.of("f",
        Atom.of("b"),
        Struct.of("f",
            Atom.of("c"),
            Var.of("Z"))))
```



Creating Collections via Static Factories I

Creating Lists

`List.empty()` creates an `EmptyList`

`Empty.list()` creates an `EmptyList`

`List.of(`⟨*Container*⟩`)` creates a `[]`-terminated `List` containing all the terms in ⟨*Container*⟩

- where ⟨*Container*⟩ may be an iterable, a sequence, or a varargs of `Terms`

`List.from(`⟨*Container*⟩`,Term?)` creates a `List` containing all the terms in ⟨*Container*⟩ and terminated by either the last argument, if it is non-null, or be the last term in ⟨*Container*⟩, otherwise

`Cons.singleton(Term)` creates `List` containing just one item

`Cons.of(Term,Term)` creates a term of the form `'.'`$(t_1,\ t_2)$ where t_1, t_2 are the first and second arguments, respectively

Creating Collections via Static Factories II

```
List.from(t1, t2, t3, last = List.empty()) // [t1, t2, t3]
// is a shortcut for
List.of(t1, t2, t3)
// which is a shortcut for
Cons.of(
    t1,
    Cons.of(
        t2,
        Cons.of(
            t3,
            Empty.list())))

List.from(t1, t2, t3, last = Var.of("T")) // [t1, t2, t3 | T]
// is a shortcut for
Cons.of(
    t1,
    Cons.of(
        t2,
        Cons.of(
            t3,
            Var.of("T"))))

Cons.singleton(t) // [t]
// is a shortcut for
Cons.of(t, List.empty())
```

Creating `Collections` via Static Factories III

Creating `Tuples`

`Tuple.of(Term,Term,`⟨*Container*⟩`)` creates a `Tuple` containing at least 2 terms, other than the ones in ⟨*Container*⟩

- where ⟨*Container*⟩ may be a (possibly empty) iterable, a sequence, or a varargs of `Terms`

Creating Collections via Static Factories IV

Creating Sets

`Set.empty()` creates an `EmptySet`

`Empty.set()` creates an `EmptySet`

`Set.of(`⟨*Container*⟩`)` creates a `Set` containing all the terms in ⟨*Container*⟩

- where ⟨*Container*⟩ may be an iterable, a sequence, or a varargs of `Terms`

Creating Clauses via Static Factories I

Creating Clauses

`Clause.of(Struct?,`⟨*Container*⟩`)` creates either a `Directive` or a `Rule` depending on whether the provide `Struct` is `null` or not. In both cases items in ⟨*Container*⟩ compose the body of the clause, whereas the `Struct` is an optional head

- where ⟨*Container*⟩ may be an iterable, a sequence, or a varargs of `Terms`
- ! in any case, if ⟨*Container*⟩ contains at least 2 terms, a `Tuple` is created behind the scenes

`Rule.of(Struct,`⟨*Container*⟩`)` creates a `Rule` using provided `Struct` as head and the *conjunction* of terms in ⟨*Container*⟩ as body

- ! returns a `Fact` is ⟨*Container*⟩ is empty

`Fact.of(Struct)` creates a `Fact` given the `Struct` acting as head

Identity vs. Equalities I

Terms Identity

Two terms are identical iff:

- both are `Integers` having the same value
- both are `Reals` having the same value
- both are `Atoms` having the same value
- both are `Variables` having the same complete name
- both are `Structures` having the same
  - functor, and arity
  - their arguments are pair-wise identical

- Terms' identity is tested via `Term.equals(Any?)`
- or via `Term.equals(Term, true)`
- or via t_1 == t_2 (because of Kotlin's operator overloading)
  - in Kotlin, objects' identity is tested via t_1 === t_2)

Identity vs. Equalities II

Terms Equality

Two terms are equal iff:

- both are `Integers` having the same value
- both are `Reals` having the same value
- both are `Atoms` having the same value
- both are `Variables` having the same simple name
- both are `Structures` having the same
  - functor, and arity
  - their arguments are pair-wise equal

- Terms' equality is tested via `Term.equals(Term, false)`

Identity vs. Equalities III

Terms Structural Equality

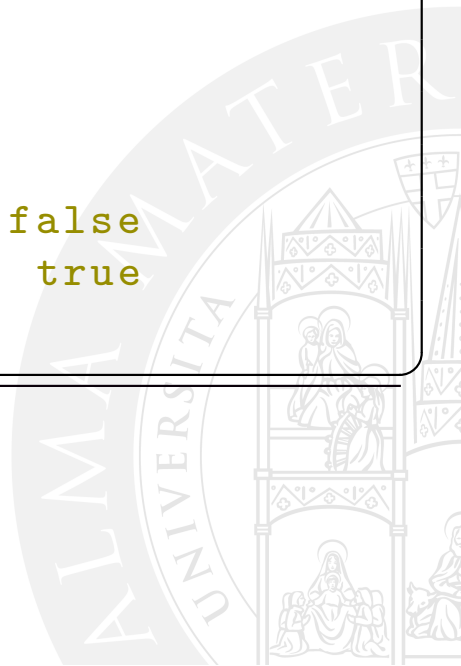
Two terms are structurally equal iff:

- both are `Numberss` having the same value
- both are `Atoms` having the same value
- both are `Variables`
- both are `Structures` having the same
  - functor, and arity
  - their arguments are pair-wise structurally equal

- Terms' structural equality is tested via `Term.structurallyEquals(Term)`

Identity vs. Equalities IV

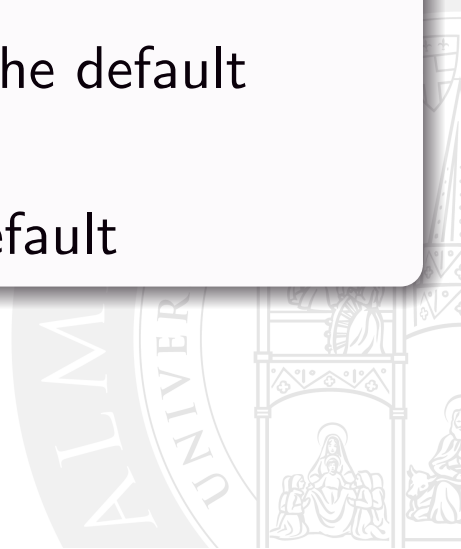
```
val t1 = Struct.of(Integer.ONE, Atom.of("a"), Var.of("X"))
      // f(1, a, X_1)
val t2 = Struct.of(Integer.ONE, Atom.of("a"), Var.of("X"))
      // f(1, a, X_2)
val t3 = Struct.of(Real.ONE, Atom.of("a"), Var.of("Y"))
      // f(1.0, a, Y_3)

println(t1.equals(t2)) // false
println(t1 == t1) // alias for the above
println(t1.equals(t2, useVarCompleteName = true)) // false
println(t1.equals(t2, useVarCompleteName = false)) // true
println(t1.structurallyEquals(t2)) // true
```



Total Ordering of Terms I

Overview

- Terms are totally ordered
- Terms are comparable according to the default ordering
- Interface `TermComparator` aims to compare terms according to some custom ordering
- `TermComparator.DefaultComparator` implements the default ordering of terms
- `Term.compareTo` exploits `DefaultComparator` by default



Total Ordering of Terms II

Default ordering of terms

$$\texttt{Var} < \texttt{Real} < \texttt{Integer} < \texttt{Atom} < \texttt{Struct}$$

- variables are lexicographically ordered by complete name
- numbers are ascendantly ordered by value
- atoms are lexicographically ordered by value
- structures are ordered by
  1. by arity, ascendantly
  2. then by functor, lexicographically
  3. then by 1^{st} argument
  4. then by 2^{nd} argument
  - ⋮

Total Ordering of Terms III

```
val terms = listOf(
        Atom.of("z"),                                   // z
        Struct.of("g", Atom.of("Z")),                   // g(z)
        Integer.ONE,                                    // 1
        Real.ONE,                                       // 1.0
        Var.of("X"),                                    // X
        Integer.of(2),                                  // 2
        Real.of(-3.4),                                  // -3.4
        Var.of("F"),                                    // F
        Struct.of("f", Atom.of("A"), Integer.ZERO),     // f(A, 0)
        Atom.of("a")                                    // a
)

terms.sorted().forEach { println(it) }
// F, X, -3,4, 1.0, 1, 2, a, z, g(z), f(A, 0)
```

Terms' Immutability I

Takeaway

All terms are immutable

Terms' Immutability II

Consequences

- Terms cannot be modified
  - → terms are side-effect free
  - → caching & instance re-use can be exploited pervasively
  - → no concurrency issue
- They must always be re-created from scratch, upon edit
- Editing a term implies creating a copy of that term
- Thus, many operations imply term copying:
  - assigning a value to a variable
  - applying a substitution to a term
  - altering some structure's arguments/functor
  - refreshing some term's variables

  ...

! Cloning a deep term may be time consuming

Terms' Immutability III

About Cloning Lists

- Lists are very likely to become very deep

→ List cloning is made lazy to save computational resources

Focus on...

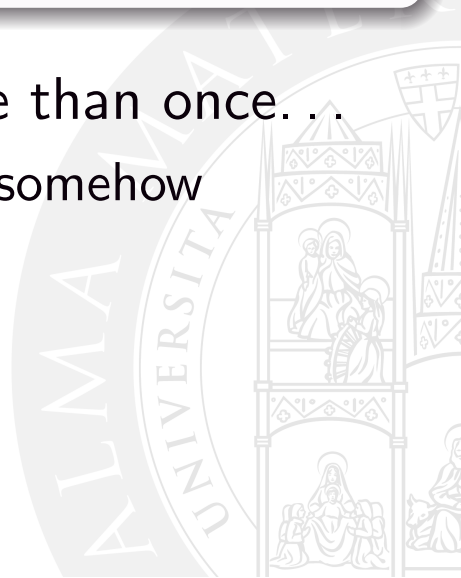
Conventions for Variables I

About Variables Creation

- Variables are always created different
- There is no way to create two equals variables
  - ! this is deliberate

→ To create a term where the same variable occurs more than once...

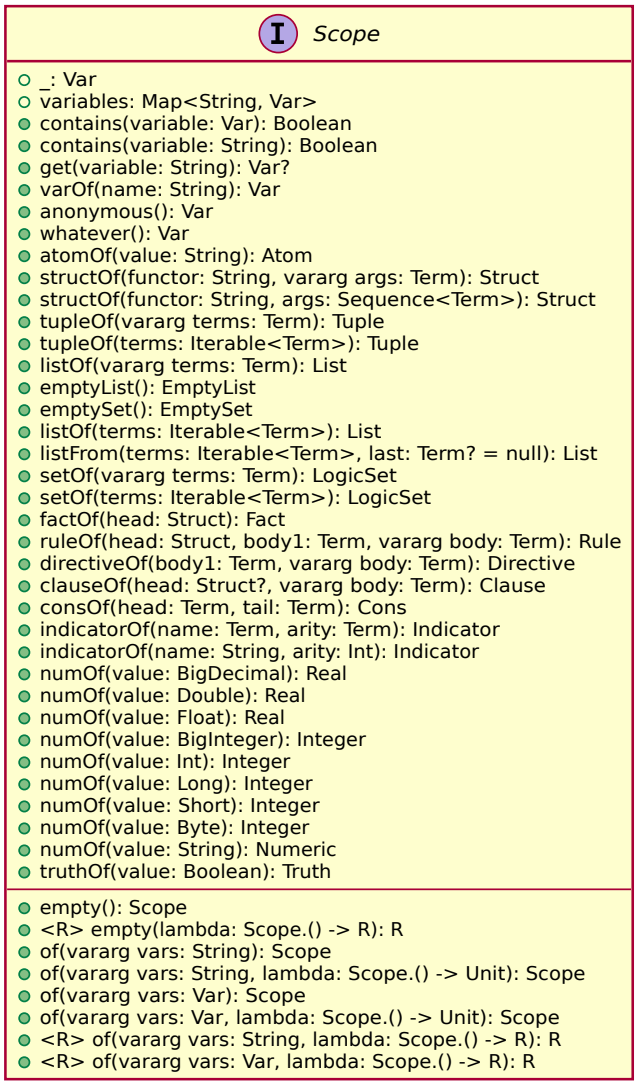
- ...a reference to the variable should be locally saved somehow



Conventions for Variables II

```
Rule.of(
    Struct.of("anchestor", Var.of("X"), Var.of("Y")),
    Struct.of("parent", Var.of("X"), Var.of("Z")),
    Struct.of("parent", Var.of("Z"), Var.of("Y"))
) // anchestor(X_1, Y_2) :- parent(X_3, Z_4), parent(Z_5, Y_6).

// is logically equivalent to

Rule.of(
    Struct.of("anchestor", Var.of("A"), Var.of("B")),
    Struct.of("parent", Var.of("C"), Var.of("D")),
    Struct.of("parent", Var.of("E"), Var.of("F"))
) // anchestor(A, B) :- parent(C, D), parent(E, F).

// THIS IS WRONG. Correct way below:

val X = Var.of("X"); val Y = Var.of("Y"); val Z = Var.of("Z")

Rule.of(
    Struct.of("anchestor", X, Y),
    Struct.of("parent", X, Z),
    Struct.of("parent", Z, Y)
) // anchestor(X_1, Y_2) :- parent(X_1, Z_3), parent(Z_3, Y_2).
```

The need for Scopes I



The need for Scopes II

Scopes

Scopes are factories of terms allowing re-use of variables

Why Scopes

- Instances of `Scope` expose methods to create `Terms`
- Alternative to create `Terms` via static factories
- Instances of variables are cached by simple name

→ Purpose: reusing `Variables` while creating `Terms`

`variables: Map<String, Var>` returns all the `Variables` created so far in this `Scope`, indexed by *simple* name

`varOf(String: Var` returns a `Var` having the provided simple, name possibly re-using some entry of `variables`, if possible, or creating a new one, otherwise

The need for Scopes III

`anonymous(): Var` alias for `Var.anonymous`

`atomOf(String): Atom` alias for `Atom.of`

`structOf(String, ⟨Container⟩): Struct` alias for `Struct.of`

- where ⟨*Container*⟩ may be an iterable, a sequence, or a varargs of `Terms`

⋮

- (one method exists in Scope for each static factory described before)

How to create Scopes

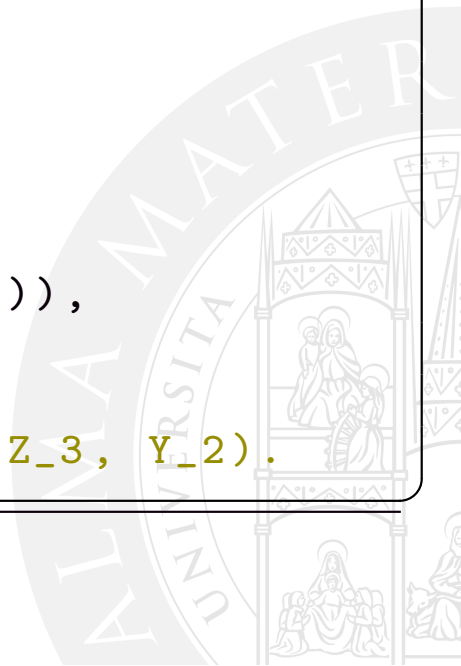
`Scope.empty()` creates a new empty Scope

`Scope.empty(Scope.()->R): R` creates a new empty Scope and accepts a lambda with receiver aimed at creating an object of type `R` using that Scope

- can be used via the syntax `Scope.empty { ⟨Code⟩ }` in Kotlin

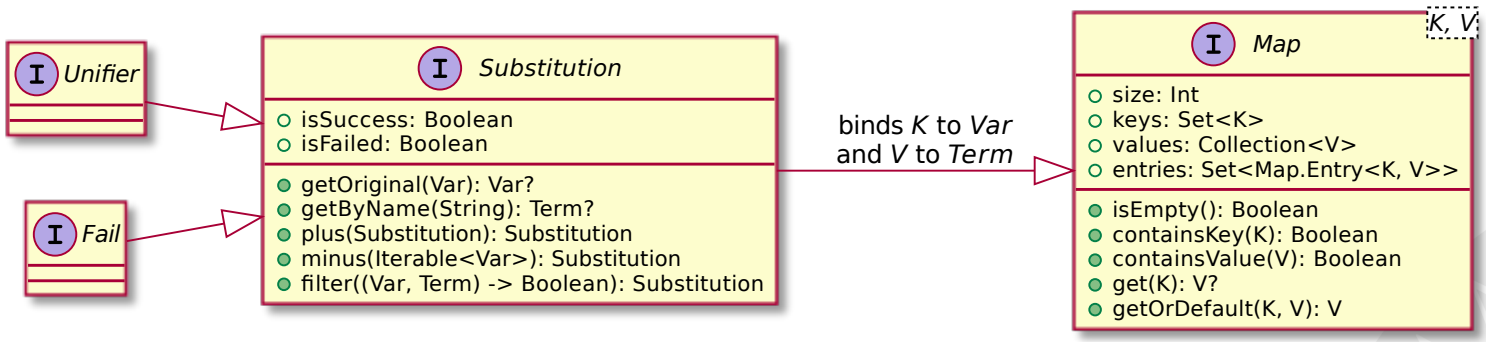
The need for Scopes IV

```
val rule: Rule = Scope.empty {
    ruleOf(
        structOf("anchestor", varOf("X"), varOf("Y")),
        structOf("parent", varOf("X"), varOf("Z")),
        structOf("parent", varOf("Z"), varOf("Y"))
    ) // anchestor(X_1, Y_2) :- parent(X_1, Z_3), parent(Z_3, Y_2).
}

// or, alternatively:

val s = Scope.empty()
val rule: Rule = s.ruleOf(
    s.structOf("anchestor", s.varOf("X"), s.varOf("Y")),
    s.structOf("parent", s.varOf("X"), s.varOf("Z")),
    s.structOf("parent", s.varOf("Z"), s.varOf("Y"))
) // anchestor(X_1, Y_2) :- parent(X_1, Z_3), parent(Z_3, Y_2).
```



Focus on...

The Substitution Type Hierarchy I



The Substitution Type Hierarchy II

The Substitution type

Base type for all substitutions i.e. assignments of Variables to Terms

- Substitution is a sub-type of Map<Var, Term>
- There are two sub-types of Substitution:
  - Unifiers, actually mapping a (possibly empty) set of Vars to their corresponding Term
    - ie an object of the form $\{v_1{=}t_1, \ldots, v_n{=}t_n\}$ where $t \geq 0$, all v_i are Variables, and all t_i are Terms
    - ! the empty mapping is an admissible unifier
  - Fail, representing the failed substitution
- Both Unifier and Fail are nested types of Substitution

isSuccess: Boolean checks whether the current Substitution is a Unifier

The Substitution Type Hierarchy III

`isFail: Boolean` checks whether the current `Substitution` is `Failed`

`getByName(String): Term?` gets the value of a `Variable` by its *simple* name

`plus(Substitution): Substitution` merges the current `Substitution` with the provided one (a third one is created)

`minus(`⟨*Container*⟩`): Substitution` returns a knew `Substitution` not containing any `Var` contained in ⟨*Container*⟩

- where ⟨*Container*⟩ is an iterable, sequence, or varargs of `Variables`

`minus(Substitution): Substitution` returns a knew `Substitution` not containing any `Var` contained in the provided `Substitution`

`filter(`⟨*Container*⟩`): Substitution` returns a knew `Substitution` containing only the `Variables` contained in the provided ⟨*Container*⟩

The Substitution Type Hierarchy IV

`filter(`⟨*Predicate*⟩`):` `Substitution` returns a knew `Substitution` containing only the `Variables` contained for which ⟨*Predicate*⟩ is `true`

+ methods & properties of `Map` such as `get`, `size`, etc.

The Substitution Type Hierarchy V

Substitutions can be merged

Assuming that:

- $U_1 = \{X_1{=}t_1\,,\ \ldots,\ X_n{=}t_n\}$ is a unifier
- $U_2 = \{Y_1{=}t'_1\,,\ \ldots,\ Y_m{=}t'_m\}$ is another unifier
- $\bot$ denotes the failed substitution

$$U_1 + U_2 = \begin{cases} \{X_1{=}t_1\,,\ \ldots,\ X_n{=}t_n\,,\ Y_1{=}t'_1\,,\ \ldots,\ Y_m{=}t'_m\} \\ \qquad \text{if } \nexists i,j : X_i = Y_j \\ \{X_1{=}t_1\,,\ \ldots,\ Y_i{=}t'_j\,,\ \ldots,\ Y_m{=}t'_m\} \\ \qquad \text{if } \forall i,j : X_i = Y_j \implies t_i = t'_j \\ \bot \text{ otherwise} \end{cases}$$

$$U_1 + \bot = \bot + U_2 = \bot$$

The Substitution Type Hierarchy VI

```
val X = Var.of("X"); val Y = Var.of("Y"); val Z = Var.of("Z")

val u1 = Substitution.unifier(X to Integer.of(1), Y to Atom.of("a")) // {X=1, Y=a}
val u2 = Substitution.unifier(Z to Struct.of(Integer.of(1), Atom.of("a"))) // {Z=f(1, a)}
val f = Substitution.failed();

val s1: Substitution = u1 + u2 // {X=1, Y=a, Z=f(1, a)}

println(s1 is Substitution.Unifier) // true
println(s1.isSuccess) // true
println(s1[X]) // 1
println(s1.get(Y)) // a
println(s1.getByName("Z")) // f(1, a)

println(s1 - Y) // {X=1, Z=f(1, a)}
println(s1.filter { it.name == "X" }) // {X=1}

val s2: Substitution = u1 + f
println(s2 is Substitution.Fail) // true

val s3 = u1 + Substitution.of(X to Integer.of(2)) // {X=1, Y=a} + {X=2}
println(s3 is Substitution.Fail) // true
```

Creating Substitutions

Static factories for Substitutions

`Substitution.failed()` return an instance of `Fail`

`Substitution.of(`⟨*KeyValuePairs*⟩`)` creates a new `Substitution` using the provided key-value pairs, where

- keys can be either strings or `Vars`
- values can be arbitrary `Terms`
- multiple pairs can be provided
- if the same `Var` is assigned to different `Terms`, an instance of `Fail` is returned

`Substitution.unifier(`⟨*KeyValuePairs*⟩`)` creates a new `Unifier` using the provided key-value pairs, where

- if the same `Var` is assigned to different `Terms`, a `SubstitutionException` is thrown

Applying Substitutions to Terms I

Definition and Notation

Let

- t be a term
- $U = \{X_1{=}t_1\,,\ \ldots,\ X_n{=}t_n\}$ be a unifier

then we denote by

$$t' = t[U]$$

the term attained by applying U to t.

- t' has the same structure of t...
- ...except each occurrence of X_i is replaced by t_i

Applying Substitutions to Terms II

Notable Cases

- Applying the failed substitution $\bot$ to any term t makes no sense
- If x is ground then $x[U] = x$, $\forall U$

Applying Substitutions to Terms III

In practice

Let *t* denote a `Term` and *s* denote a `Substitution`:

`t.apply(Substitution): Term` returns a new `Term` attained by applying the provided `Substitution` to *t*

- may throw a `SubstitutionApplicationException` if the `Substitution` is an instance of `Fail` at run time

`t.get(Substitution): Term` is an alias for *t*`.apply`

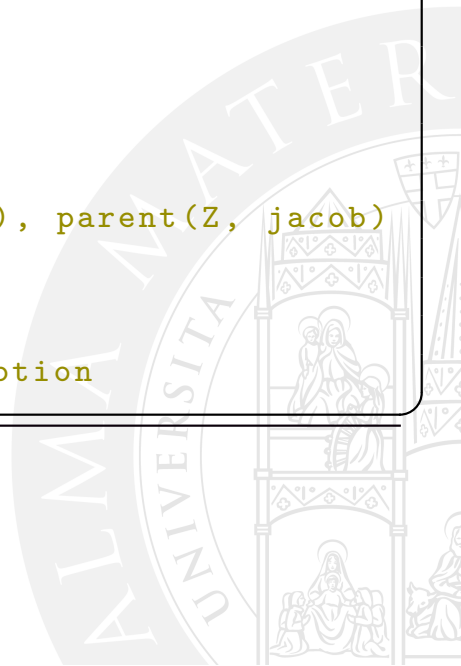
- this can be invoked via the syntax *t*`[`*s*`]` in Kotlin

s`.applyTo(Term): Term` returns a new `Term` attained by applying *s* to the provided `Term`

- may throw a `SubstitutionApplicationException` if the *s* is an instance of `Fail` at run time

Applying Substitutions to Terms IV

```
val X = Var.of("X"); val Y = Var.of("Y"); val Z = Var.of("Z")

val r1 = Rule.of(
    Struct.of("anchestor", X, Y),
    Struct.of("parent", X, Z),
    Struct.of("parent", Z, Y)
) // anchestor(X, Y) :- parent(X, Z), parent(Z, Y).

val u = Substitution.unifier(
    X to Atom.of("abraham"),
    Y to Atom.of("jacob")
) // {X=abraham, Y=jacob}

val r2 = r1.apply(u) // anchestor(abraham, jacob) :- parent(abraham, Z), parent(Z, jacob)
    .
val r3 = u.applyTo(r1) // same as above

val r4 = r2[Substitution.fail()] // throws SubstitutionApplicationException
```



Refreshing Terms I

Definition and Notation

Let t be a `Term` and suppose that t:

- is compound (i.e. it is a `Struct`)
- it contains some `Variables` $X_1, \ldots, X_n$

then refreshing t means creating another term t' such that:

- t' has the same structure of t
- each occurrence of X_i is replaced by X_i', where
  - X_i' is a fresh variable s.t. $X_i' \neq X_i$
  - X_i' has the same name of X_i

Notable Cases

- If t is ground then refreshing t leaves it unaffected

Refreshing Terms II

In practice

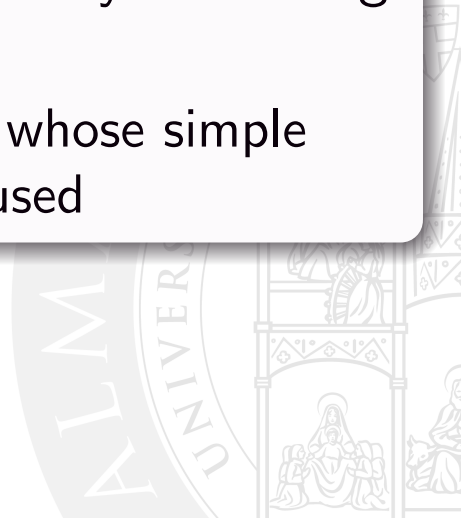
Let t denote a `Term`:

`t.freshCopy(): Term` returns a new `Term` attained by refreshing t

- the new `Term` is guaranteed to only contain `Variables` which have never been used before

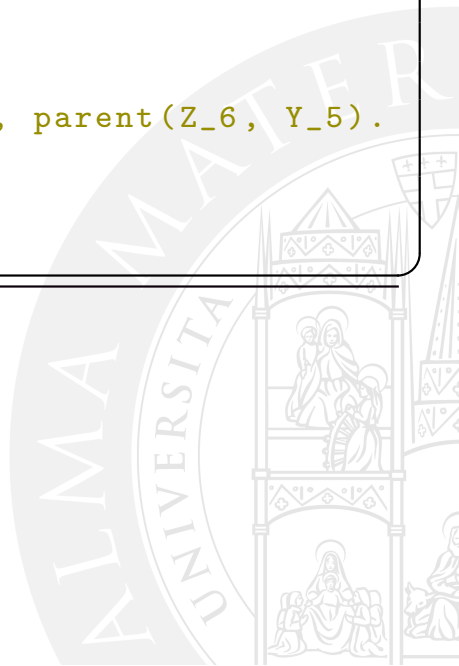
`t.freshCopy(Scope): Term` returns a new `Term` attained by refreshing t using the provided `Scope`

- if the provided `Scope` already contains some `Variable` whose simple name is exploited within t, then is that `Variable` is reused



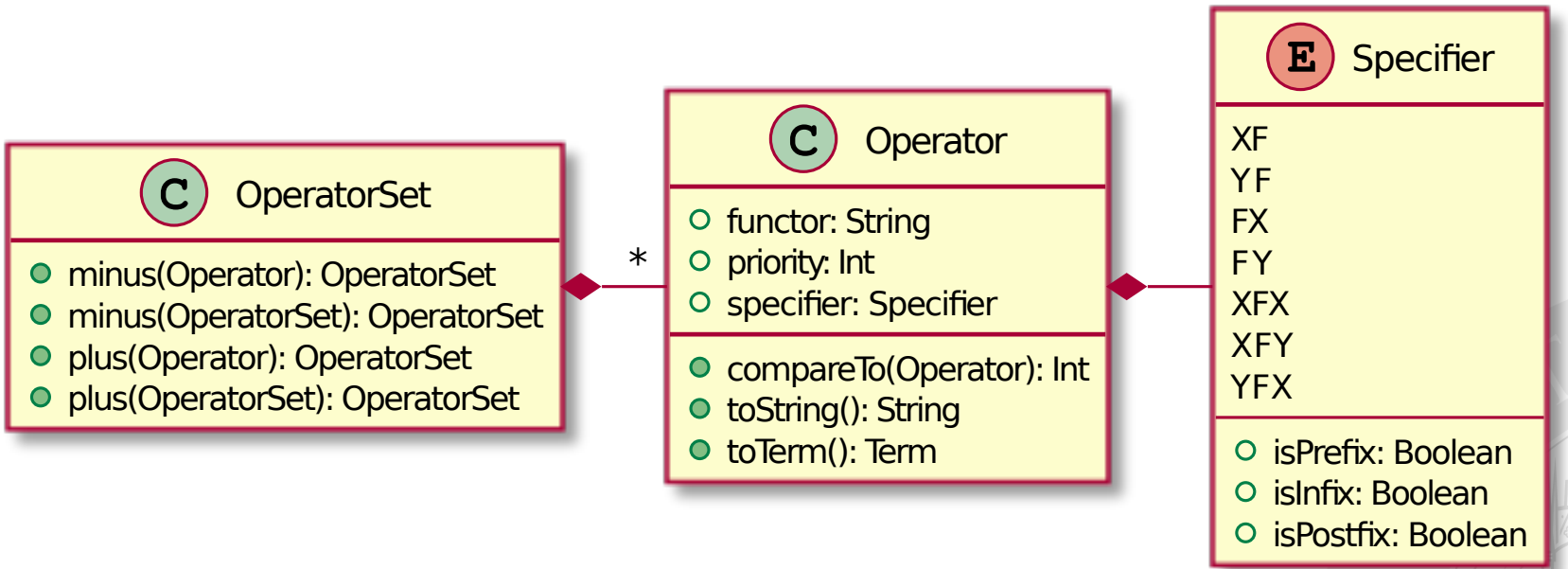
Refreshing Terms III

```
val rule: Rule = Scope.empty {
    ruleOf(
        structOf("anchestor", varOf("X"), varOf("Y")),
        structOf("parent", varOf("X"), varOf("Z")),
        structOf("parent", varOf("Z"), varOf("Y"))
    ) // anchestor(X_1, Y_2) :- parent(X_1, Z_3), parent(Z_3, Y_2).
}

val copy = rule.freshCopy() // anchestor(X_4, Y_5) :- parent(X_4, Z_6), parent(Z_6, Y_5).

println(rule == copy) // false
println(rule.equals(copy, useVarCompleteName = false)) // true
```



Focus on...

Prolog-like Operators I



Prolog-like Operators II

Definition

A Prolog-like `Operator` is an algebraic object composed by 3 fields:

`functor:` `String` i.e. the operator's symbol

`priority:` `Int` where lower integers correspond to higher priorities

- conventionally, 0 is max priority, 1200 is min priority

`specifier:` `Specifier` is a descriptor of the operator's

- arity i.e. the amount of arguments it takes (either binary or unary)
- position w.r.t. arguments (prefix, postfix, or infix)
- associativity e.g. either right- or left-associative, or non-associative

Prolog-like Operators III

About Specifiers

- `Specifier` is an *enum* with 7 constants
- Each one is a combination of 3 letters:
  - F denoting the operator
  - X denoting an expression having strictly higher priority
  - Y denoting an expression having higher or equal priority
- Each one describes an operator's arity, position, and associativity:

	Left-associative	**Non-associative**	**Right-associative**
Prefix	—	`FX`	`FY`
Infix	`YFX`	`XFX`	`XFY`
Postifx	`YF`	`XF`	—

Prolog-like Operators IV

Creating Operators

`Operator(String, Specifier, Int)` creates a new `Operator` having the provided functor, specifier, and priority

`Operator.fromTerm(Integer, Atom, Atom): Operator?` creates a new `Operator` having the provided priority, specifier, and functor—provided as `Terms`

- the 2^{nd} argument must be one `Atom` among `fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, or `xfy`
- returns `null` otherwise

`Operator.fromTerm(Struct): Operator` creates a new `Operator` out of a `Struct` of the form `op(`p`, `s`, `f`)`

- where p must be an `Integer`
- and s be one `Atom` among `fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, or `xfy`
- returns `null` otherwise

The meaning of Specifiers I

How two interpret **prefix** specifiers FX and FY

- If f and g are operators of type FY, and T is a term:
  - expressions of the form `f(g(T))`
    - can be written as `f g T` only if *priority*(g) $\leq$ *priority*(f)
    - can be written as `f(g T)` otherwise
  - expressions of the form `f(f(T))` or `g(g(T))`
    - can be written as `f f T` or `g g T`
- Otherwise, if f and g are operators of type FX:
  - expressions of the form `f(g(T))`
    - can be written as `f g T` only if *priority*(g) $<$ *priority*(f)
    - can be written as `f(g T)` otherwise
  - expressions of the form `f(f(T))`, or `g(g(T))`
    - can never be written as `f f T` or `g g T`
    - but can be written as `f(f T)` or `g(g T)`

The meaning of Specifiers II

How two interpret **postfix** specifiers XF and YF

- If f and g are operators of type YF, and T is a term:
  - expressions of the form f(g(T))
    - can be written as T g f only if *priority*(g) $\leq$ *priority*(f)
    - can be written as (T g) f otherwise
  - expressions of the form f(f(T)) or g(g(T))
    - can be written as T f f or T g g
- Otherwise, if f and g are operators of type XF:
  - expressions of the form f(g(T))
    - can be written as T g f only if *priority*(g) $<$ *priority*(f)
    - can be written as (T g) f otherwise
  - expressions of the form f(f(T)), or g(g(T))
    - can never be written as T f f or T g g
    - but can be written as (T f)f or (T g)g

The meaning of Specifiers III

How two interpret **infix** specifiers YFX and XFY

- If f and g are operators of type YFX, and A, B, and C are terms:
  - expressions of the form `f(g(A, B), C)`
    - can be written as `A f B g C` only if *priority*(g) $\leq$ *priority*(f)
    - can be written as `(A f B) g C` otherwise
  - expressions of the form `f(f(A, B), C)` or `g(g(A, B), C)`
    - can be written as `A f B f C` or `A g B g C`
- Conversely, if f and g are operators of type XFY:
  - expressions of the form `f(A, g(B, C))`
    - can be written as `A f B g C` only if *priority*(g) $\leq$ *priority*(f)
    - can be written as `A f (B g C)` otherwise
  - expressions of the form `f(A, f(B, C))` or `g(A, g(B, C))`
    - can be written as `A f B f C` or `A g B g C`

The meaning of Specifiers IV

How two interpret the **infix** specifier XFX

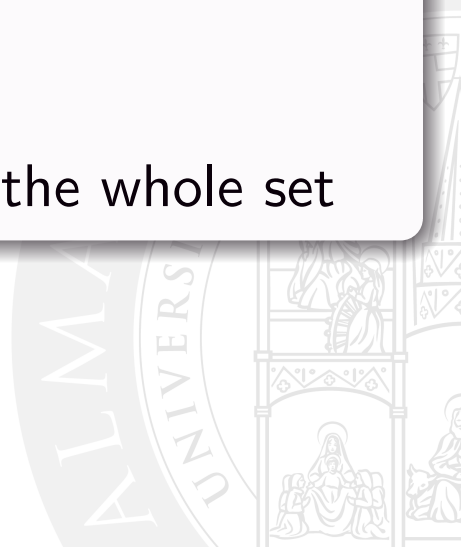
- If f and g are operators of type XFX, and A, B, and C are terms:
  - expressions of the form `f(g(A, B), C)`
    - can be written as `A f B g C` only if *priority*(g) $<$ *priority*(f)
    - can be written as `(A f B) g C` otherwise
  - expressions of the form `f(f(A, B), C)` or `g(g(A, B), C)`
    - can never be written as `A f B f C` or `A g B g C`
    - but can be written as `(A f B) f C` or `(A g B) g C`
  - expressions of the form `f(A, g(B, C))`
    - can be written as `A f B g C` only if *priority*(g) $<$ *priority*(f)
    - can be written as `A f (B g C)` otherwise
  - expressions of the form `f(A, f(B, C))` or `g(A, g(B, C))`
    - can never be written as `A f B f C` or `A g B g C`
    - but can be written as `A f (B f C)` or `A g (B g C)`

Sets of Operators I

The OperatorSet typ is a subtype of Set<Operator>

An OperatorSet is simply a set of Operators (no duplicates admitted)

- the OperatorSet is immutable
- operations exist to add/remove/check Operators within OperatorSets
- operations exists to merge OperatorSets
- adding/removing operators implies creating a copy of the whole set

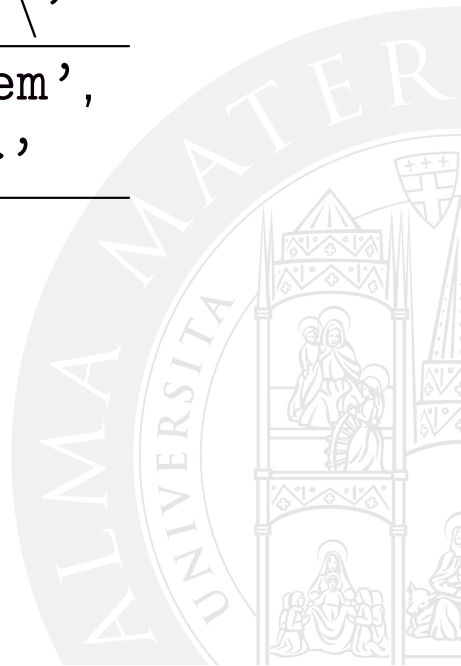


Sets of Operators II

Several notable `OperatorSet` exists:

eg one for arithmetic operators:

Priority	Specifier	Operators
500	YFX	'+', '-', '\/', '/\'
400	YFX	'*', '/', '//', 'rem', 'mod', '<<', '>>'
200	FX	'+', '-', '\'
	XFX	'**'
	XFY	'^'



Sets of Operators III

eg one for arithmetic comparison operators:

Priority	Specifier	Operators
700	XFX	'=:=', '=\=', '<', '>', '=<', '>='

Sets of Operators IV

eg one for term comparison operators:

Priority	Specifier	Operators
700	XFX	'=', '\\=', '==', '\\==' '@<', '>@', '@=<', '@>=' '=..', 'is'

Sets of Operators V

eg one for logic connectors:

Priority	Specifier	Operators
1100	XFY	';'
1050	XFY	'->'
1000	XFY	','
900	FY	'\+'

Sets of Operators VI

eg one for clause-related operators:

Priority	Specifier	Operators
1200	FX	':-', '?-'
1200	XFX	':-', '-->'

Sets of Operators VII

eg one containing all the aforementioned (ISO Prolog Standard compliant) operators:

Priority	Specifier	Operators
1200	FX	‘:-’, ‘?-’
1200	XFX	‘:-’, ‘-->’
1100	XFY	‘;’
1050	XFY	‘->’
1000	XFY	‘,’
900	FY	‘\+’
700	XFX	‘=’, ‘\\=’, ‘==’, ‘\\==’ ‘@<’, ‘>@’, ‘@=<’, ‘@>=’ ‘=..’, ‘is’
700	XFX	‘=:=’, ‘=\=’, ‘<’, ‘>’, ‘=<’, ‘>=’
500	YFX	‘+’, ‘-’,‘\/’, ‘/\’
400	YFX	‘*’, ‘/’,‘//’, ‘rem’, ‘mod’,‘<<’, ‘>>’
200	FX	‘+’, ‘-’,‘\’
	XFX	‘**’
	XFY	‘^’

Sets of Operators VIII

Creating an OperatorSet

`OperatorSet(`⟨*Container*⟩`)` creates a novel set out of some operators

- where ⟨*Container*⟩ may be an iterable, sequence, or varargs of `Operators`

`OperatorSet.EMPTY` references an empty set

`OperatorSet.ARITHMETIC` references a set containing arithmetic operators

`OperatorSet.ARITHMETIC_COMPARISON` references a set containing arithmetic comparison operators

`OperatorSet.TERM_COMPARISON` references a set containing term comparison operators

`OperatorSet.CONTROL_FLOW` references a set containing logic connectors

`OperatorSet.CLAUSES` references a set containing clause-related operators

`OperatorSet.STANDARD` references a set containing all the aforementioned operators

Focus on...

TermVisitors and their Use Cases I

About TermVisitors

- Visitor pattern[a] applied to Term
- Supports use cases where a particular operation must be performed depending on the actual type of a Term
- Essentially, a form of dynamic dispatch[b]

[a] https://en.wikipedia.org/wiki/Visitor_pattern
[b] https://en.wikipedia.org/wiki/Dynamic_dispatch

TermVisitors and their Use Cases II

```
interface TermVisitor<T> {
    fun defaultValue(term: Term): T

    fun visitTerm(term: Term): T = defaultValue(term)
    fun visitVar(term: Var): T = defaultValue(term)
    fun visitConstant(term: Constant): T = defaultValue(term)
    fun visitStruct(term: Struct): T = defaultValue(term)
    fun visitCollection(term: Collection): T = defaultValue(term)
    fun visitAtom(term: Atom): T = defaultValue(term)
    fun visitTruth(term: Truth): T = defaultValue(term)
    fun visitNumeric(term: Numeric): T = defaultValue(term)
    fun visitInteger(term: Integer): T = defaultValue(term)
    fun visitReal(term: Real): T = defaultValue(term)
    fun visitSet(term: Set): T = defaultValue(term)
    fun visitEmpty(term: Empty): T = defaultValue(term)
    fun visitEmptySet(term: EmptySet): T = defaultValue(term)
    fun visitList(term: List): T = defaultValue(term)
    fun visitCons(term: Cons): T = defaultValue(term)
    fun visitEmptyList(term: EmptyList): T = defaultValue(term)
    fun visitTuple(term: Tuple): T = defaultValue(term)
    fun visitIndicator(term: Indicator): T = defaultValue(term)
    fun visitClause(term: Clause): T = defaultValue(term)
    fun visitRule(term: Rule): T = defaultValue(term)
    fun visitDirective(term: Directive): T = defaultValue(term)
}
```

TermVisitors and their Use Cases III

TermVisitors' contract

- An instance of `TermVisitor<`T`>` v can be applied to a term t via

$$t\texttt{.accept(}v\texttt{)}$$

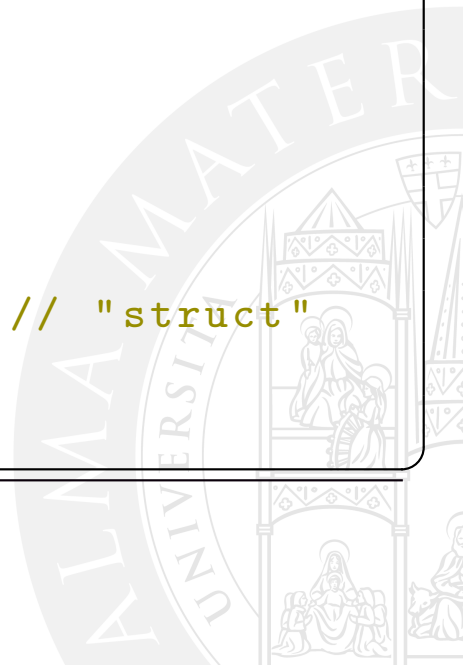
- This returns an object of type T ...
- ... attained via the most adequate method `visit`$\langle Type \rangle$ of v
  - where $\langle Type \rangle$ is the most specific type matching the run-time type of t

Implementors of TermVisitors

- May override `defaultValue` to handle the general case
- Then override some `visit`$\langle Type \rangle$ method to handle the particular case of $\langle Type \rangle$

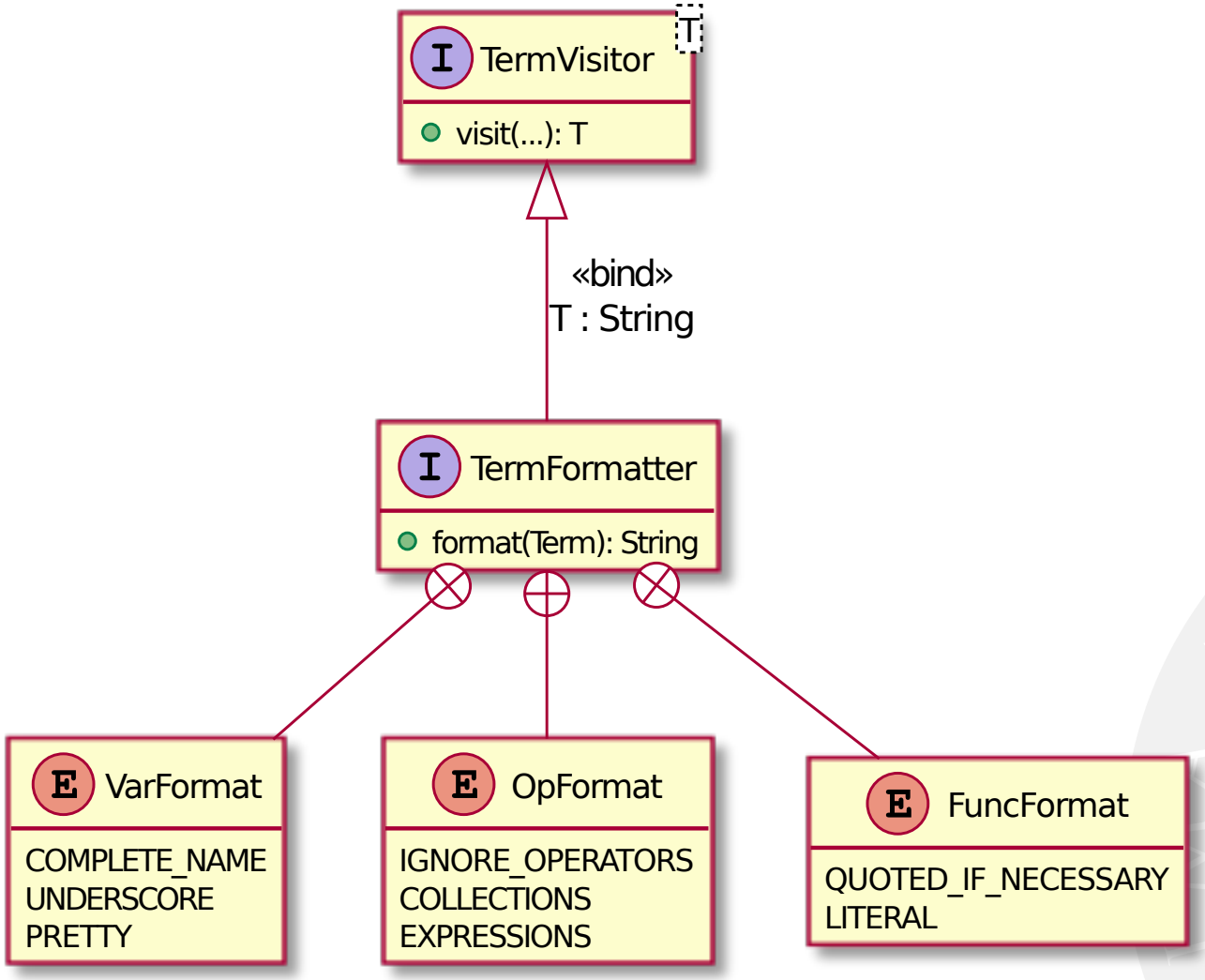
TermVisitors and their Use Cases IV

```
class MyVisitor : TermVisitor<String> {
    override fun defaultValue(term: Term): String = "?"

    override fun visitInteger(term: Integer): String = "integer"
    override fun visitStruct(term: Struct): String = "struct"
    override fun visitAtom(term: Atom): String = "atom"
}

val visitor = MyVisitor()

println(Atom.of("a").accept(visitor)) // "atom"
println(Integer.ONE.accept(visitor)) // "integer"
println(Struct.of("f", Integer.ONE).accept(visitor)) // "struct"
println(List.empty().accept(visitor)) // ?
println(Real.ONE.accept(visitor)) // ?
```



Focus on...

The TermFormatter Type I



The TermFormatter Type II

About TermFormatters

- TermFormatters aim at presenting terms and clauses as strings
- Many predefined formatters exist, serving disparate use cases
  - configurable via enums

TermFormatter enums I

VarFormat dictates how Variables are represented

COMPLETE_NAME represent variables by complete name

UNDERSCORE represent all variables in the form $\_\langle Integer \rangle$

PRETTY represent all variables by simple name

- progressively renames homonymous variables
  - eg X, X1, X2, etc

TermFormatter enums II

OpFormat dictates how Operators are treated

IGNORE_OPERATORS all Structures are represented in canonical form:

$$\langle Functor \rangle(\langle Args \rangle)$$

no matter what, including Collections

COLLECTIONS like the above, except

- lists are represented in square-brackets form
- tuples are represented in round-parentheses form
- sets are represented in curly-brackets form

EXPRESSIONS like the above except that Structures matching some operator definition are represented in infix, prefix, or infix form, using parentheses only if required

- this requires an OperatorSet to be specified

TermFormatter enums III

FuncFormat dictates how Structures' functors are represented

QUOTED_IF_NECESSARY enquotes functors if they are not well formed, escaping non-readable characters

LITERAL represents functors literally, without any enquoting or escaping

The numberVars: Boolean flag

Some TermFormatter initialisers come with a numberVars argument:

when true structures of the form

$$\texttt{'\$VAR'}(i)$$

where i is a non-negative Integer are represented as the Variable named after the i^{th} letter of the alphabet

- '$VAR'(0) → A, '$VAR'(1) → B, etc.

when false they are represented as ordinary Structures

Creating TermFormatters I

Static factories for `TermFormatters`:

`TermFormatter.of(VarFormat,OpFormat,FuncFormat,Boolean,OperatorSet)`
returns a new formatter using the provided options and operators

`TermFormatter.default(OperatorSet)` creates a new formatter using the provided operators and the following settings:

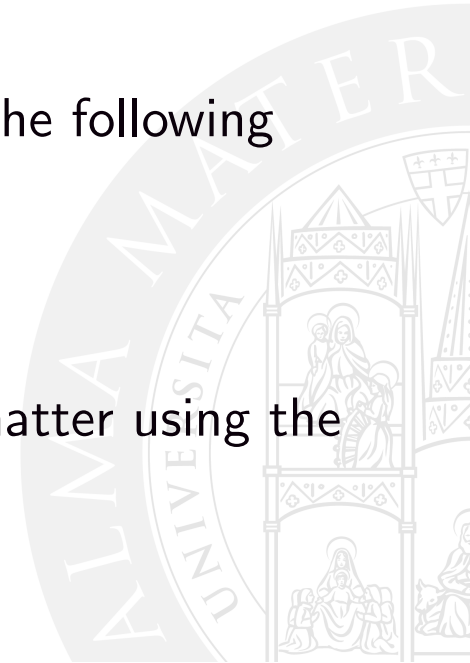
- `VarFormat.UNDERSCORE`, `OpFormat.EXPRESSIONS`, `FuncFormat.LITERAL`
- `numberVars = true`

`TermFormatter.canonical()` creates a new formatter using the following settings:

- `UNDERSCORE`, `IGNORE_OPERATORS`, `QUOTED_IF_NECESSARY`
- `numberVars = false`

`TermFormatter.readable(OperatorSet)` creates a new formatter using the provided operators and the following settings:

- `PRETTY`, `EXPRESSIONS`, `QUOTED_IF_NECESSARY`
- `numberVars = false`



Creating TermFormatters II

`TermFormatter.prettyVariables()` like `canonical`, but with:

- `VarFormat.PRETTY`, `OpFormat.COLLECTIONS`

`TermFormatter.prettyExpressions(Boolean, OperatorSet)` creates a new formatter using the provided operators and the following settings:

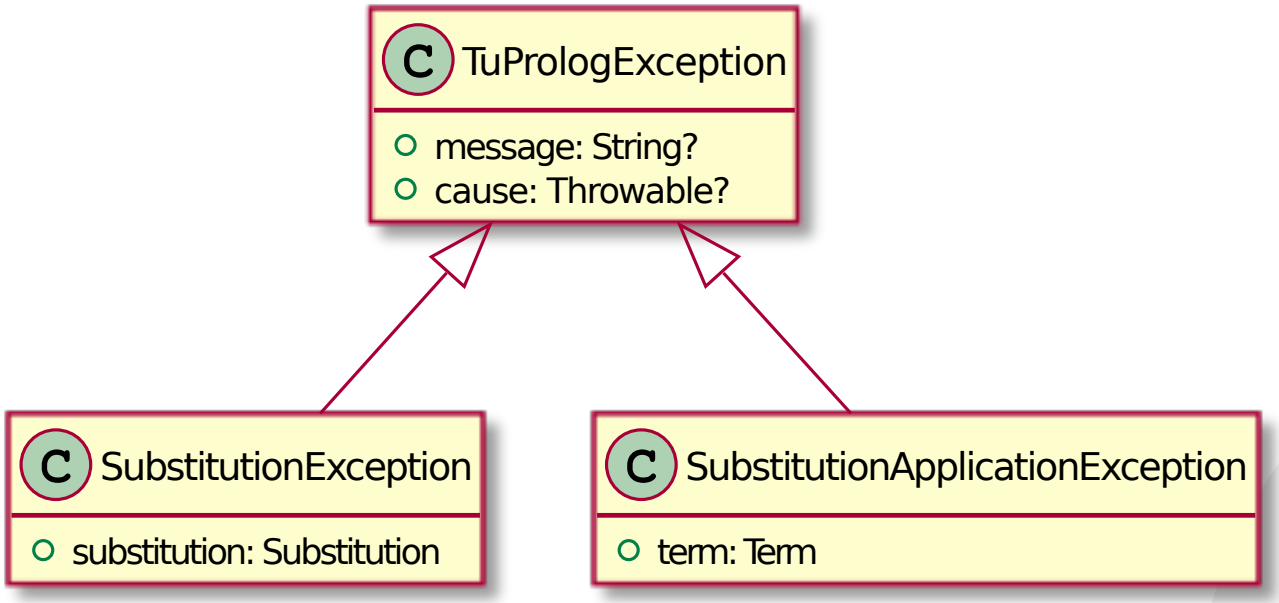
- `VarFormat.PRETTY` if the 1^{st} argument is `true`, `COMPLETE_NAMES` otherwise
- `OpFormat.EXPRESSIONS`, `FuncFormat.QUOTED_IF_NECESSARY`
- `numberVars = false`

Formatting Terms

```
val expression = Scope.empty {
    structOf("+",
        listOf(varOf("A"), varOf("B")),
        setOf(
            structOf("-", varOf("X"), intOf(1)),
            structOf("\$VAR", intOf(2))
        )
    )
} // [A, B] + (X - 1, '$VAR'(2))

println( TermFormatter.canonical().format(expression) )
    // '+'('.'(_0, '.'(_0, [])), '{}'(','('-'(_0, 1), '$VAR'(2))))
println( TermFormatter.default(OperatorSet.DEFAULT).format(expression) )
    // [_0, _0] + {_0 - 1, C0}
println( TermFormatter.readable(OperatorSet.DEFAULT).format(expression) )
    // [A, B] + {X - 1, C0}
println( TermFormatter.prettyVariables().format(expression) )
    // '+'([A, B], {'-'(X, 1), '$VAR'(2)})
println( TermFormatter.prettyExpressions(true, OperatorSet.DEFAULT).format(expression) )
    // [A, B] + {X - 1, '$VAR'(2)}
```

Focus on...

Exceptions in 2P-Kt I



Exceptions in 2P-Kt II

The `TuPrologException` type is a sub-type of `RuntimeException`

Base type for all custom exceptions possibly occurring in 2P-Kt

Important convention

Custom exceptions possibly defined into any 2P-Kt module must directly or indirectly be sub-types of `TuPrologException`

Exceptions in 2P-Kt III

Sub-types of `TuPrologException` in *:core*

`SubstitutionException` thrown when attempting to create a `Unifier` out of contradictory assignments

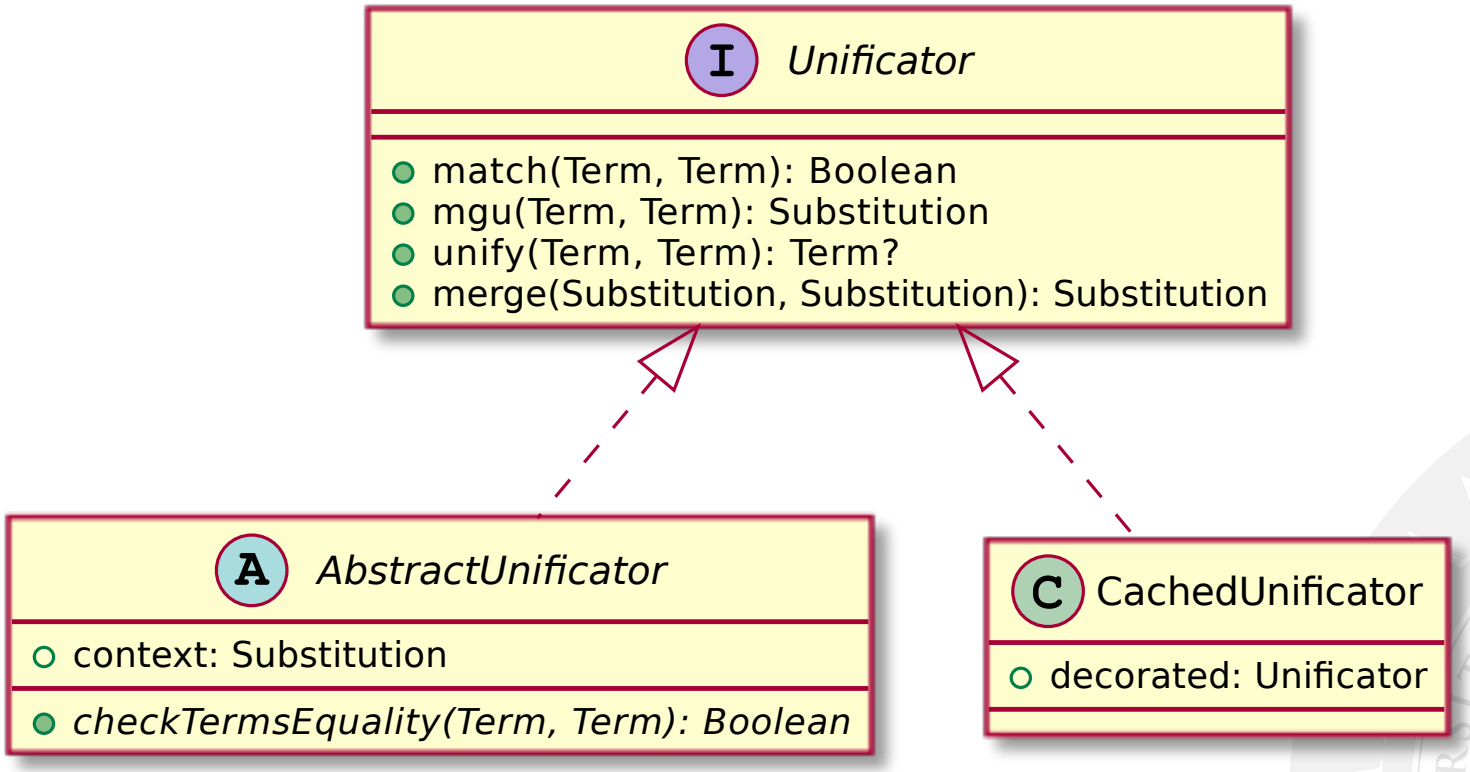
- e.g. the same `Var` is assigned to different `Terms`

`SubstitutionApplicationException` thrown when attempting to apply a `Failed` substitution to a `Term`

Next in Line...

Focus on...

The Unificator type I



The Unificator type II

The Unificator type

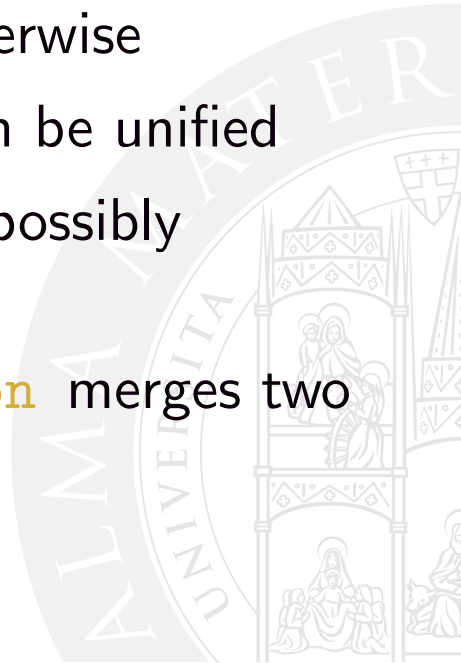
Matches two terms by finding a suitable substitution to their variables, i.e. the *most general unifier* (MGU)

`mgu(Term, Term): Substitution` returns the MGU between two `Terms` if a substitution is found, or a `Fail` object otherwise

`match(Term, Term): Boolean` checks if two `Terms` can be unified

`unify(Term, Term): Term?` tries to unify two `Terms`, possibly returning the unified Term as a result

`merge(Substitution, Substitution): Substitution` merges two `Substitutions` with occurs check enabled



Focus on...

Creating Unificators I

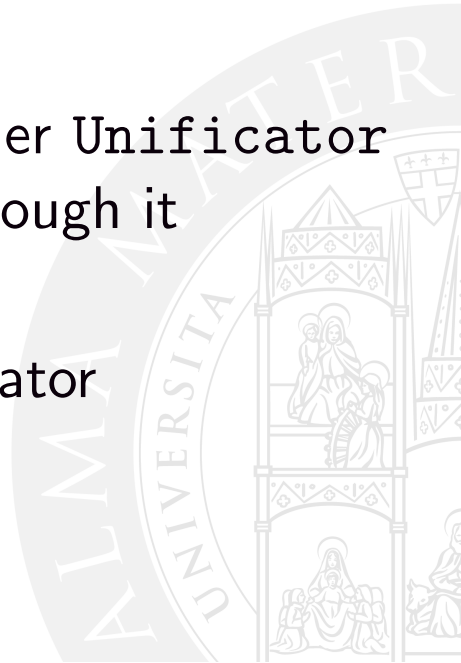
Main `Unificator` static factories:

`naive(): Unificator` unifies terms based on their `Term.equals` method, while numbers are compared by value

`strict(): Unificator` unifies terms based uniquely on their `Term.equals` methods

`cached(Unificator, Int): Unificator` makes another `Unificator` cached, memorizing the most recent `Terms` unified through it (according to the given cache size)

`default: Unificator` returns a `cached`, `strict` unificator



Using Unificators I

General unification strategy:

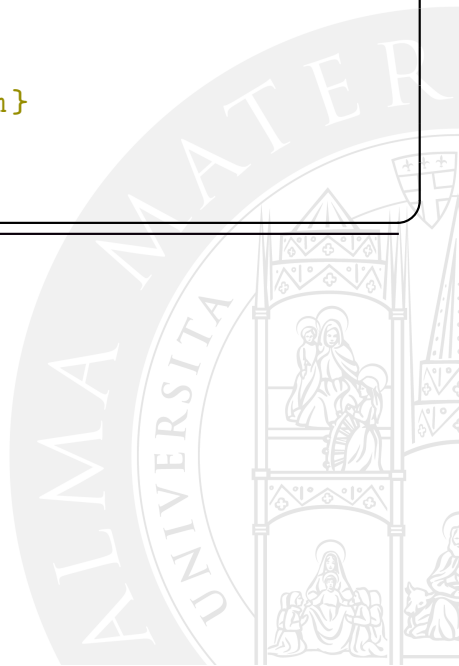
$MGU(\cdot,\cdot)$	c	X	$f(t_1,\ldots,t_n)$
k	$c = k \to \emptyset$ $c \neq k \to \bot$	$\{X = k\}$	$\bot$
Y	$\{Y = c\}$	$\{X = Y\}$	$\{Y = f(t_1,\ldots,t_n)\}$
$g(x_1,\ldots,x_m)$	$\bot$	$\{X = g(x_1,\ldots,x_m)\}$	$f = g \wedge n = m \to \bigcup_i MGU(t_i, x_i)$ $f \neq g \vee n \neq m \to \bot$

! implementations may alter the way terms are compared

Using Unificators II

Successful unification with `default`:

```
val unificator = Unificator.default

val t1 = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))
val t2 = Struct.of("father", Var.of("X"), Atom.of("isaac"))

val substitution: Substitution = unificator.mgu(t1, t2) // {X_0=abraham}
val match: Boolean = unificator.match(t1, t2) // true
val result: Term? = unificator.unify(t1, t2) // father(abraham, isaac)
```



Using Unificators III

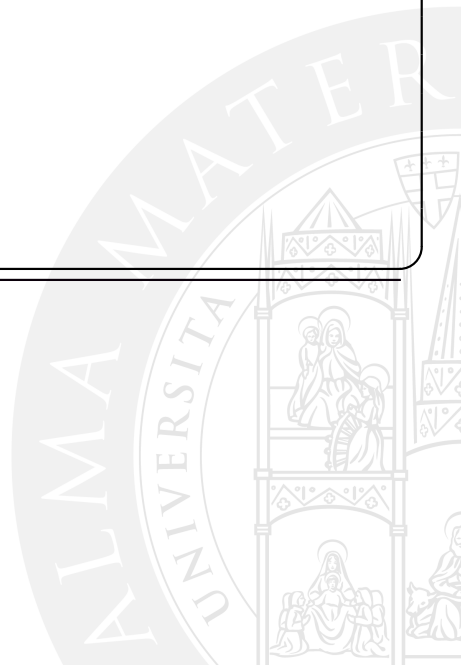
Successful unification with `default` (no bindings):

```
val unificator = Unificator.default

val t1 = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))
val t2 = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))

val substitution: Substitution = unificator.mgu(t1, t2) // {}
```

Using Unificators IV

Failed unification with default:

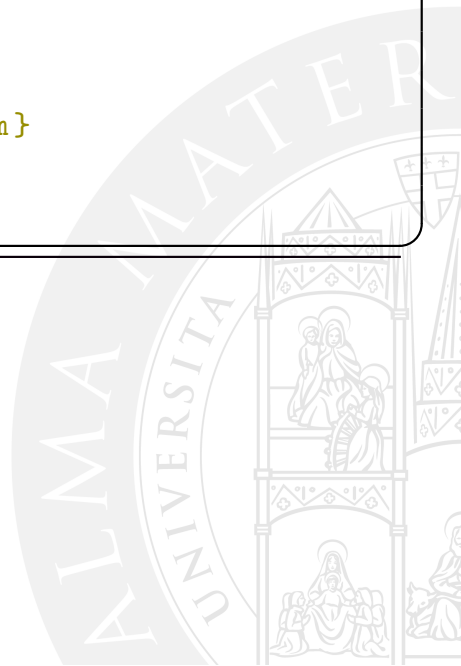
```
val unificator = Unificator.default

val t1 = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))
val t2 = Struct.of("father", Atom.of("isaac"), Atom.of("abraham"))

val substitution: Substitution = unificator.mgu(t1, t2)
val isFail: Boolean = substitution is Substitution.Fail // true
val isFailed: Boolean = substitution.isFailed // true
val match: Boolean = unificator.match(t1, t2) // false
val result: Term? = unificator.unify(t1, t2) // null
```



Using Unificators V

Unification with cached unificator:

```
val unificator = Unificator.default
val cached = Unificator.cached(unificator, capacity = 5)

val term = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))
val goal = Struct.of("father", Var.of("X"), Atom.of("isaac"))

val substitution: Substitution = cached.mgu(term, goal) // {X_0=abraham}
val match: Boolean = cached.match(term, goal) // true
val result: Term? = cached.unify(term, goal) // father(abraham, isaac)
```



Focus on...

Custom Unificators I

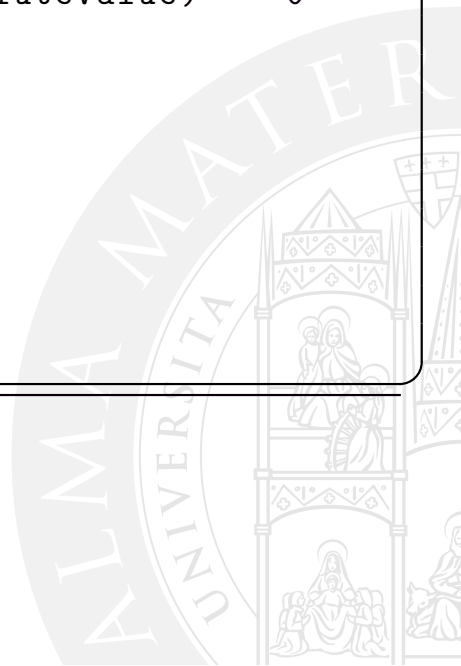
Writing custom Unificators

- create an instance of the `AbstractUnificator` type
- override `checkTermsEquality` method
  - ! checking whether the two provided `Terms` are matching

Custom Unificators II

E.g. unifying numbers by their absolute value:

```
val unificator = object : AbstractUnificator() {
    override fun checkTermsEquality(x: Term, y: Term): Boolean = when {
        x is Integer && y is Integer ->
            x.value.absoluteValue.compareTo(y.value.absoluteValue) == 0
        x is Numeric && y is Numeric ->
            x.decimalValue.absoluteValue.compareTo(y.decimalValue.absoluteValue) == 0
        else -> x == y
    }
}

val t1 = Struct.of("f", Integer.of(1)) // f(1)
val t2 = Struct.of("f", Integer.of(-1)) // f(-1)

val match = unificator.match(t1, t2) // true
val result = unificator.unify(t1, t2) // f(1)
```



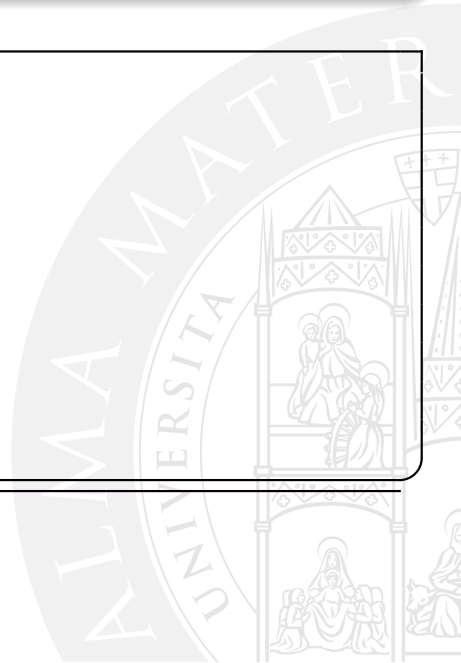
Focus on...

Infix operators I

Infix operators: `unifyWith`, `matches`, `mguWith`

- defined in `Unificator`'s companion object
- built as functions with receivers
- enable streamlined, DSL-like unification

```
import it.unibo.tuprolog.unify.Unificator.Companion.matches
import it.unibo.tuprolog.unify.Unificator.Companion.mguWith
import it.unibo.tuprolog.unify.Unificator.Companion.unifyWith

val t1 = Struct.of("father", Atom.of("abraham"), Atom.of("isaac"))
val t2 = Struct.of("father", Var.of("X"), Atom.of("isaac"))

val substitution: Substitution = t1 mguWith t2 // {X_0=abraham}
val match: Boolean = t1 matches t2 // true
val result: Term? = t1 unifyWith t2 // father(abraham, isaac)
```



Next in Line...

Focus on...

Collections Design I

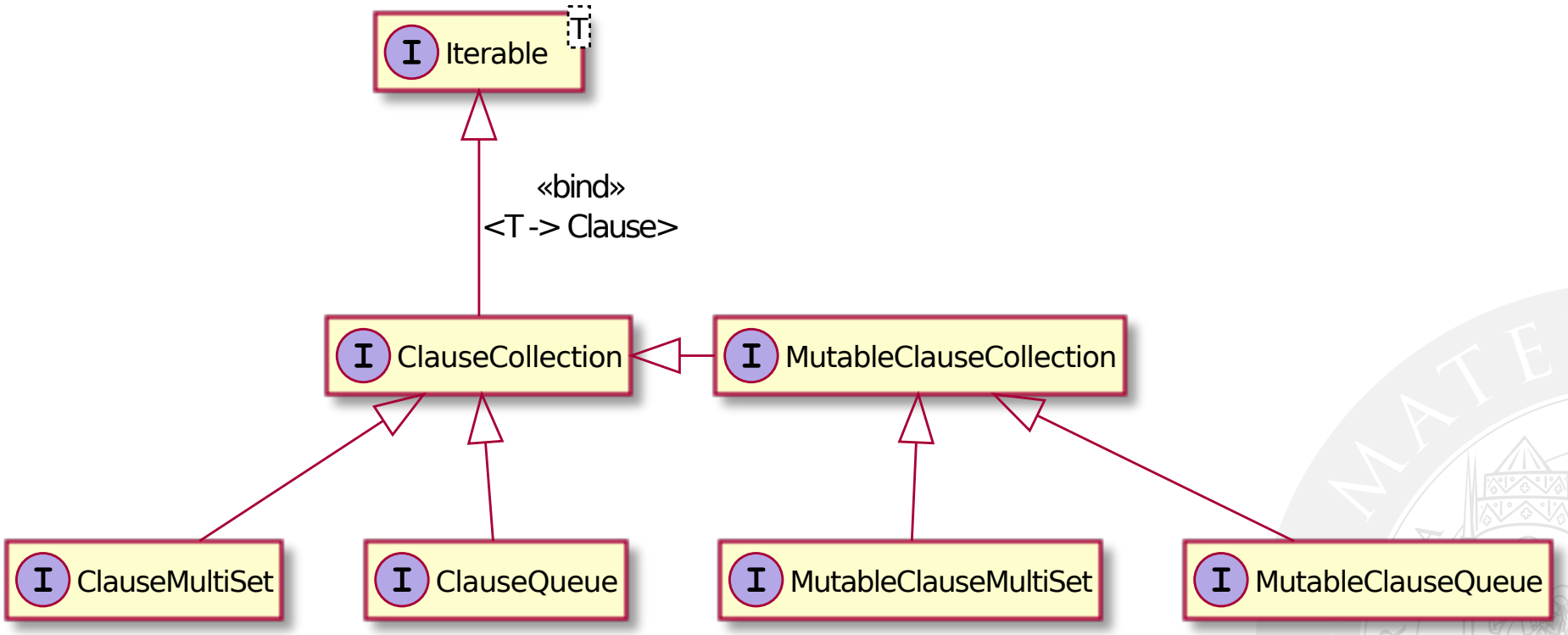
Clause Collections

- manage the storage of `Clauses`
- provide methods for addition and removal of clauses
- designed as `Iterable` subtypes

Two dimensions: *mutability* and *ordering*

- our design tackles both issues
- *mutable* collections can be modified in-place
- *ordered* collections keep track of the order of insertion

The `ClauseCollection` Type Hierarchy I



The ClauseCollection Type Hierarchy II

The ClauseCollection interface

- defines typical insertion and removal operations
- method names similar to Java Collections
  - ! following the *least astonishment principle*

`add(Clause): ClauseCollection` returns a new collection containing the given clause and the existing ones

`retrieve(Clause): RetrieveResult<ClauseCollection>` tries to remove the given clause from the collection

`contains(Clause): Boolean` checks if the collection contains the given clause

`size: Int` computes the size of the collection

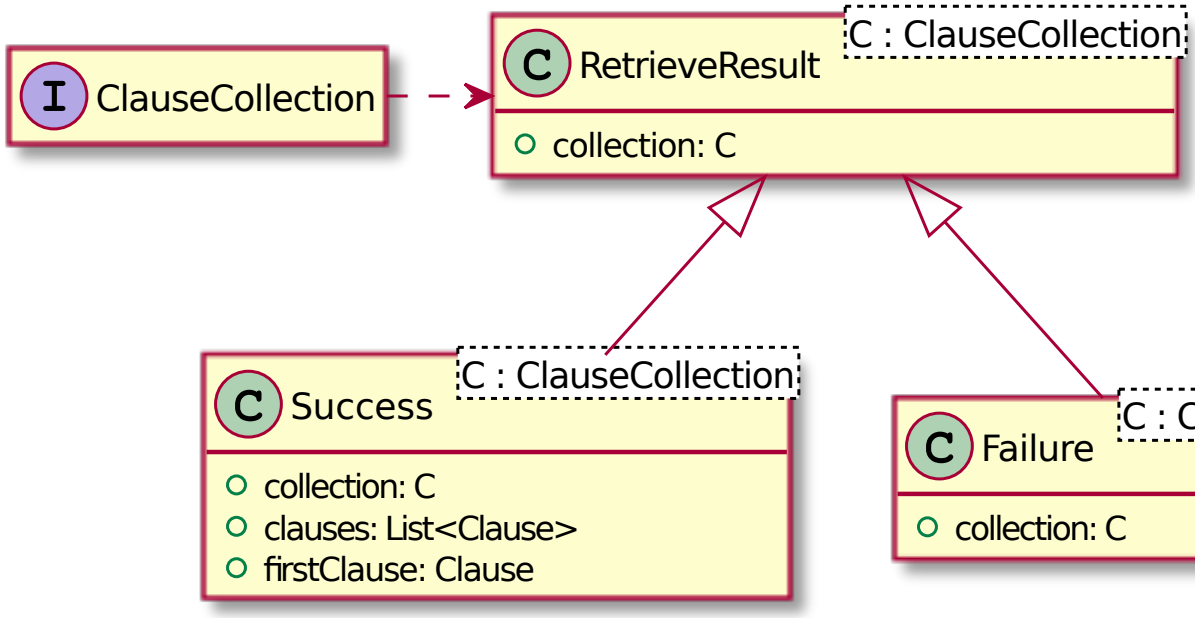
`isEmpty(): Boolean` checks if the collection contains any clauses

`iterator(): Iterator<Clause>` returns the collection's iterator

The ClauseCollection Type Hierarchy III

The RetrieveResult type

- represents the output of a `retrieve(Clause)` call
- can be a `Success` or `Failure`



The `ClauseCollection` Type Hierarchy IV

The `MutableClauseCollection` interface

- subtype of `ClauseCollection`
- insertions and removals are performed **in-place**
  - ! same collection is returned after any operation

`add(Clause): MutableClauseCollection` adds a new clause to this collection

`retrieve(Clause): RetrieveResult<MutableClauseCollection>` tries to remove the given clause from this collection

The `ClauseCollection` Type Hierarchy V

Clause Queues

- **preserve** the **ordering** of clauses
  - ! according to their *insertion* order
- store clauses in a *queue*-like structure
- defined by the `ClauseQueue` and `MutableClauseQueue` types

`addFirst(Clause): ClauseQueue` adds a clause in the *first* position to the collection

`add/addLast(Clause): ClauseQueue` adds a clause in the *last* position to the collection

`getFifoOrdered(Clause): Sequence<Clause>` produces a sequence of clauses unifying with the given one, searching *from first to last*

`getLifoOrdered(Clause): Sequence<Clause>` produces a sequence of clauses unifying with the given one, searching *from last to first*

The `ClauseCollection` Type Hierarchy VI

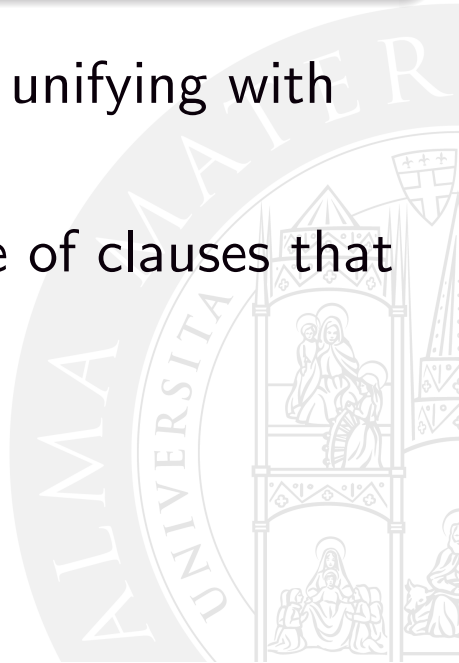
Clause Multisets

- ignore clause ordering
- useful when many clauses may match the goal
- defined by the `ClauseMultiSet` and `MutableClauseMultiSet` types

`count(Clause): Long` computes the number of clauses unifying with the given one

`get(Clause): Sequence<Clause>` produces a sequence of clauses that unify with the given one

! equivalent to the `[]` operator



Creating ClauseCollections I

Main `ClauseCollection` static factories:

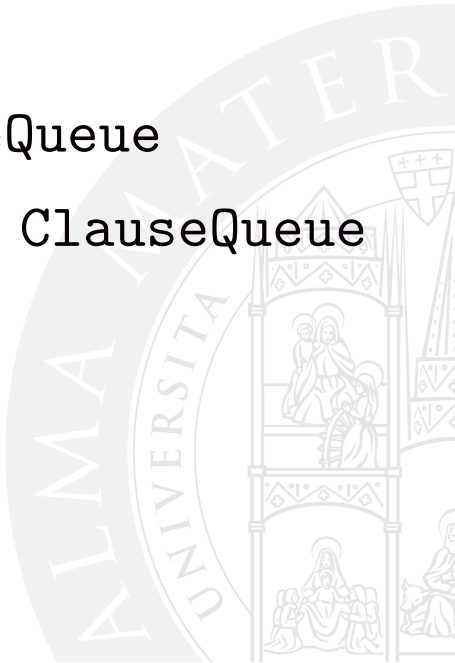
`emptyMultiSet(): ClauseMultiSet` creates an empty `ClauseMultiset`

`multiSetOf(Iterable<Clause>): ClauseMultiSet` creates a `ClauseMultiSet` from the given clauses

`emptyQueue(): ClauseQueue` creates an empty `ClauseQueue`

`queueOf(Iterable<Clause>): ClauseQueue` creates a `ClauseQueue` from the given clauses

These factories return *immutable* collections by default



Creating ClauseCollections II

`ClauseQueue` and `ClauseMultiSet` provide three main static factories:

`empty(): C` creates an empty `C`

`of(Iterable<Clause>): C` creates a `C` from the given clauses

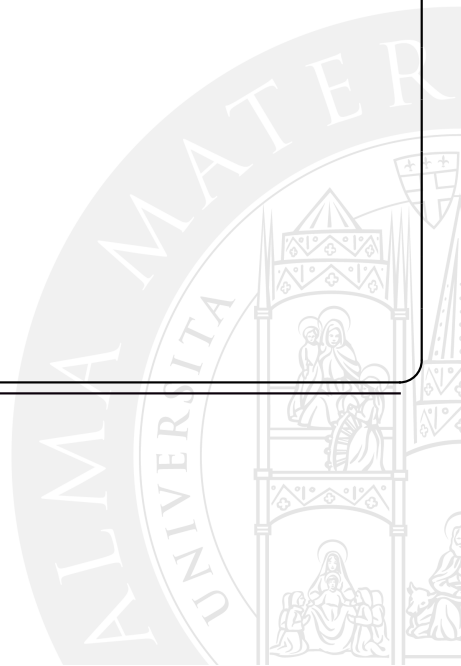
`of(vararg Clause): C` creates a `C` from the given clauses

with `C` being either `ClauseQueue` or `ClauseMultiSet`

Using ClauseCollections I

Ordering a ClauseQueue:

```
val queue = ClauseCollection.queueOf(
    Fact.of(Struct.of("f", Integer.of(1))),
    Fact.of(Struct.of("f", Integer.of(2))),
) // f(1), f(2)

val fifoOrdered = queue.getFifoOrdered(
    Fact.of(Struct.of("f", Var.of("X")))
) // f(1), f(2)
val lifoOrdered = queue.getLifoOrdered(
    Fact.of(Struct.of("f", Var.of("X")))
) // f(2), f(1)
```



Using ClauseCollections II

Populating a ClauseQueue:

```
val queue = ClauseCollection.queueOf(
    Fact.of(Struct.of("f", Integer.of(1)))
) // f(1)

val queue2 = queue.addLast(
    Fact.of(Struct.of("f", Integer.of(2)))
) // f(1), f(2)

val queue3 = queue2.add(
    Fact.of(Struct.of("f", Integer.of(3)))
) // f(1), f(2), f(3)

val queue4 = queue3.addFirst(
    Fact.of(Struct.of("f", Integer.of(4)))
) // f(4), f(1), f(2), f(3)
```

Using ClauseCollections III

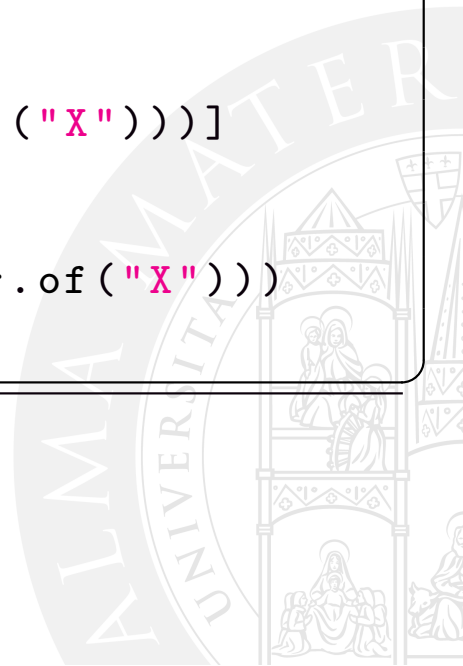
Retrieving clauses from a ClauseQueue:

```
val queue = ClauseCollection.queueOf(
    Fact.of(Struct.of("f", Integer.of(1))),
    Fact.of(Struct.of("f", Integer.of(2))),
    Fact.of(Struct.of("f", Integer.of(3))),
) // f(1), f(2), f(3)

val successfulRetrieve = queue.retrieveFirst(
    Fact.of(Struct.of("f", Integer.of(1)))
)

println(successfulRetrieve.firstClause) // f(1)

val queue2 = successfulRetrieve.collection // f(2), f(3)

val failedRetrieve = queue2.retrieveFirst(
    Fact.of(Struct.of("f", Integer.of(1)))
)
val queue3 = failedRetrieve.collection // f(2), f(3)
```

Using ClauseCollections IV

Working with ClauseMultiSets:

```
val multiSet = ClauseCollection.multiSetOf(
    Fact.of(Struct.of("f", Integer.of(1))),
    Fact.of(Struct.of("f", Integer.of(2))),
    Fact.of(Struct.of("f", Integer.of(3))),
)

val matching = multiSet[Fact.of(Struct.of("f", Var.of("X")))]
    // f(1), f(2), f(3)

val count = multiSet.count(Fact.of(Struct.of("f", Var.of("X"))))
    // 3
```



Focus on...

Theories Design and Implementations I

The `Theory` type

- provides high-level management of `Clauses`
- meant as a *façade* through which client code can access a knowledge base
- design and implementation mirror the ones devised for `Clause` collections
- designed as `Iterable` sub-types

Theories Design and Implementations II

Mutable vs. immutable theories

Just like `ClauseCollections`,

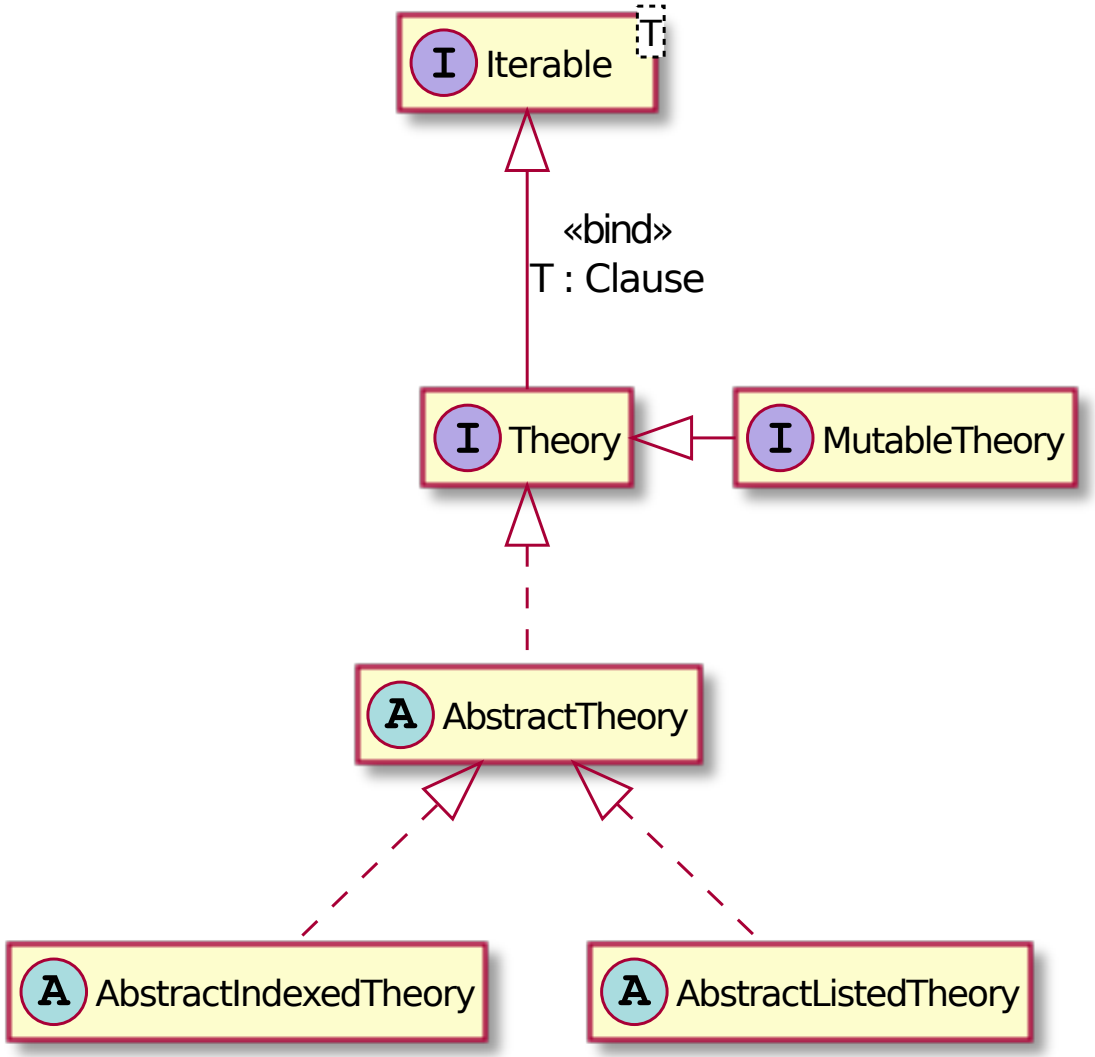
- *mutable* theories can be modified in place
- *immutable* theories cannot be changed: their operations build new ones containing the modified clauses

Indexed vs. listed Theories

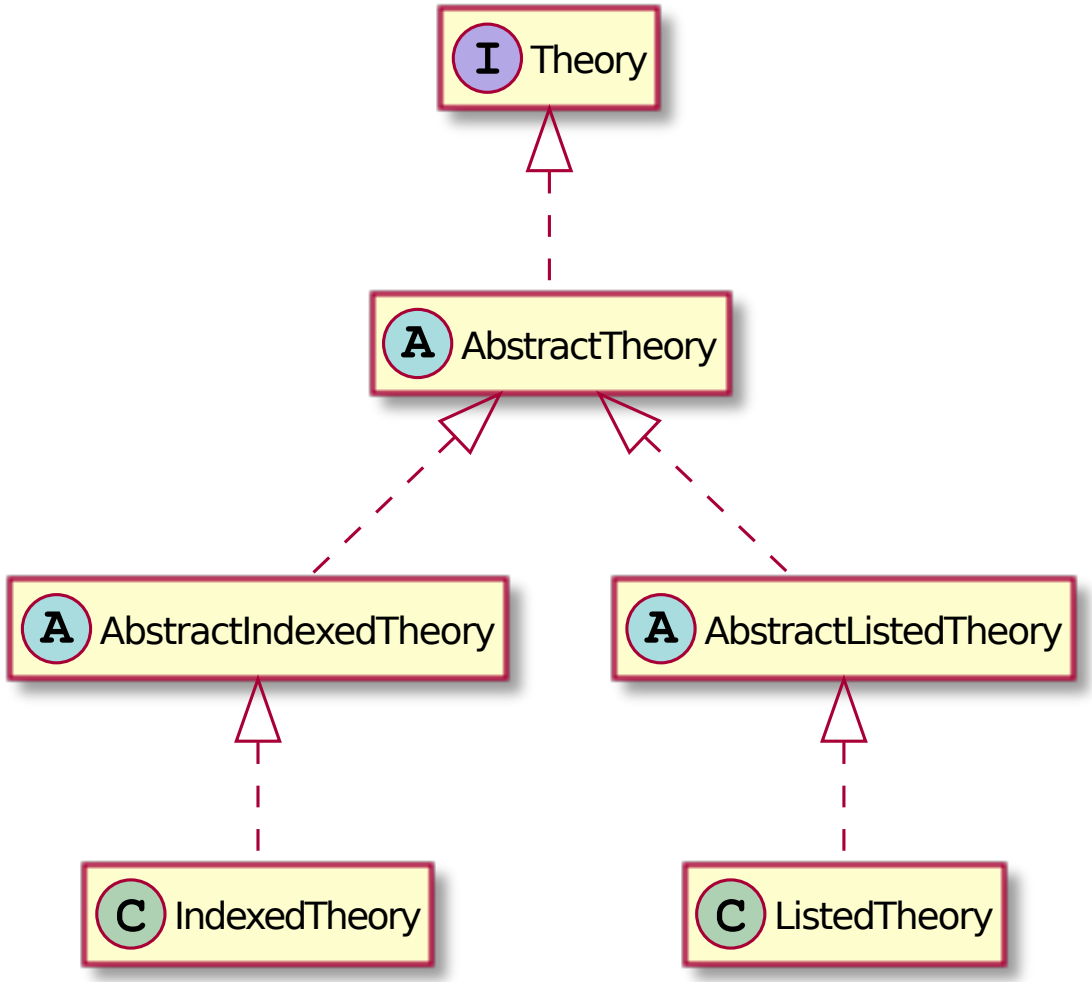
Theories come into two main implementations:

- *indexed* theories are backed by an indexed data structure
  - ! providing faster look-up, but taking up more space
- *list-based* theories are backed by lists
  - ! slower but memory-efficient
  - ! preserve order of `Clauses`

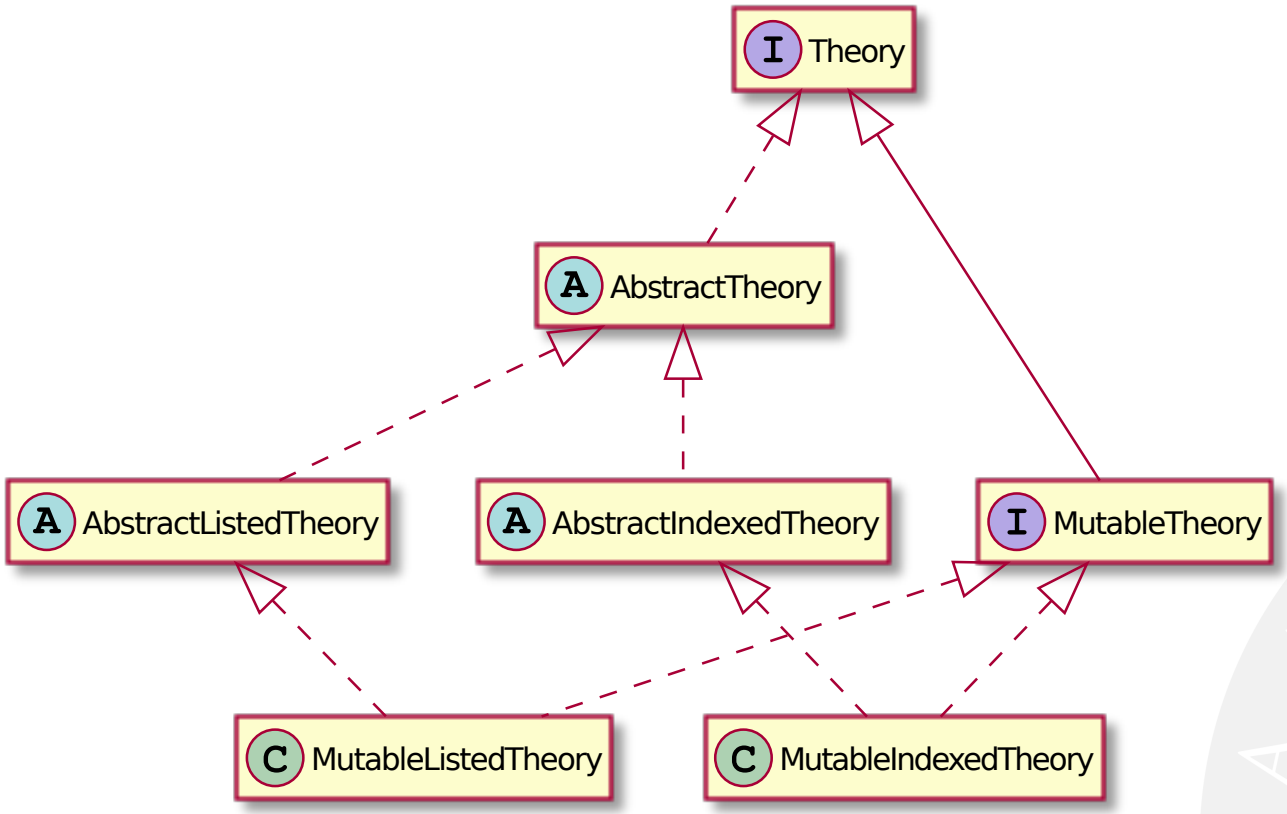
The Theory Type Hierarchy I



The Theory Type Hierarchy II



The Theory Type Hierarchy III



The Theory Type Hierarchy IV

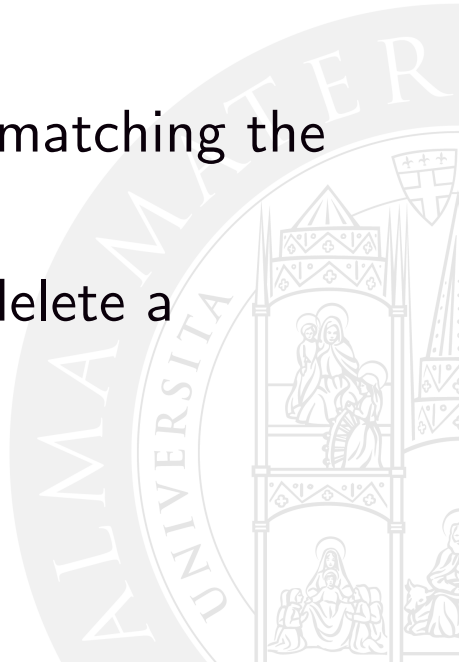
Main Theory methods:

`assertA(Clause): Theory` adds the given clause *before* all other clauses in this theory

`assertZ(Clause): Theory` adds the given clause *after* all other clauses in this theory

`abolish(Indicator): Theory` removes all the clauses matching the given Indicator

`retract(Clause): RetractResult<Theory>` tries to delete a matching clause from this theory



The Theory Type Hierarchy V

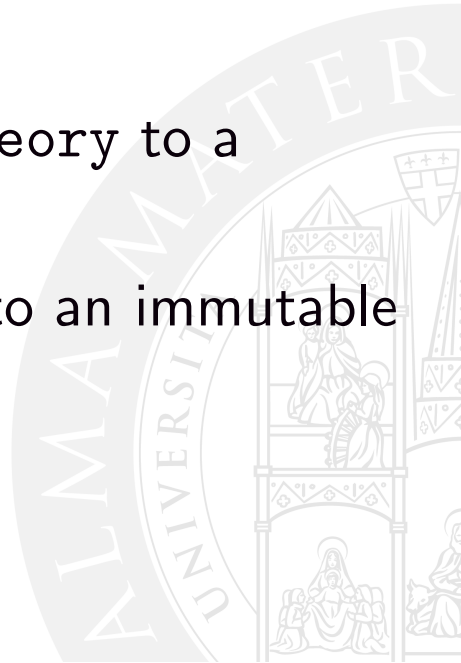
`plus(Theory): Theory` adds the given `Theory` to this

`contains(Clause): Boolean` checks if the given `Clause` is contained in this `Theory`

`get(Clause): Sequence<Clause>` retrieves all matching clauses from this `Theory`

`toMutableTheory(): MutableTheory` converts this `Theory` to a mutable one

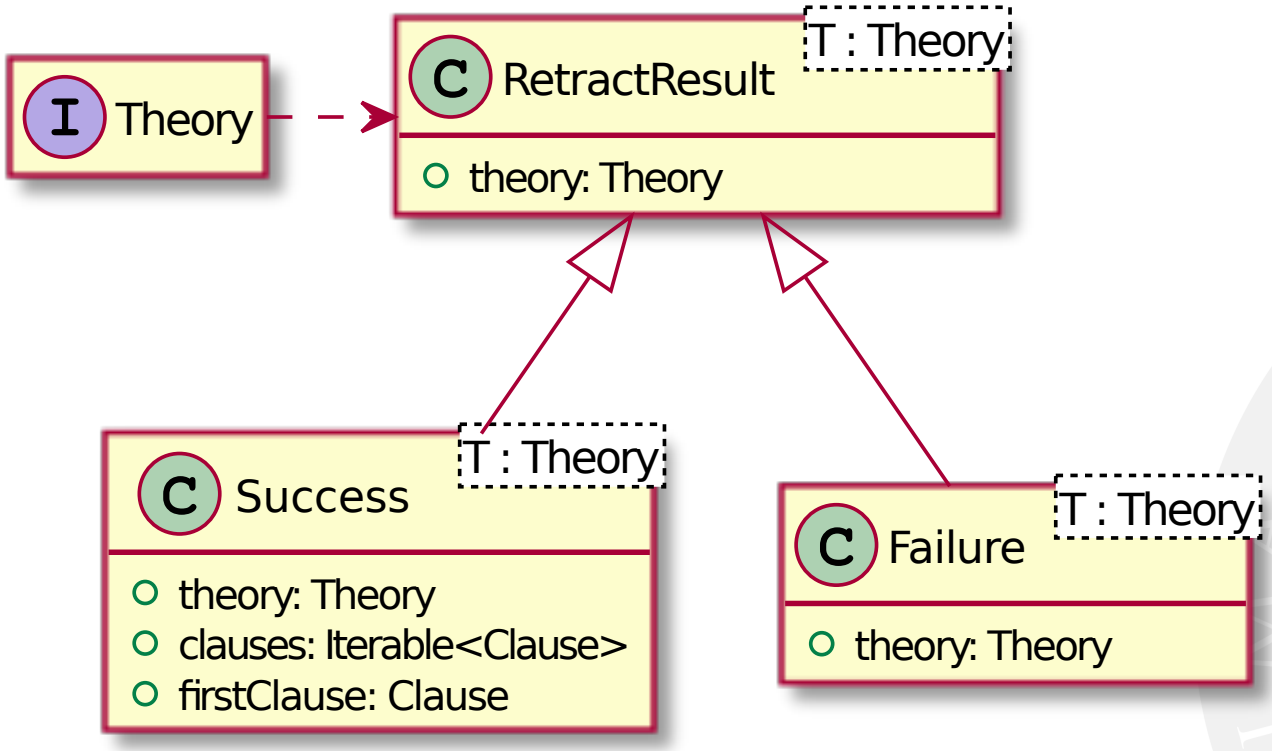
`toImmutableTheory(): Theory` converts this `Theory` to an immutable one



The Theory Type Hierarchy VI

The RetractResult type

- represents the output of a `retract(Clause)` call
- can be a `Success` or a `Failure`



Creating a Theory I

Static factories for `Theory` (same methods apply for `MutableTheory`):

`empty(): Theory` creates an empty `Theory` (indexed by default)

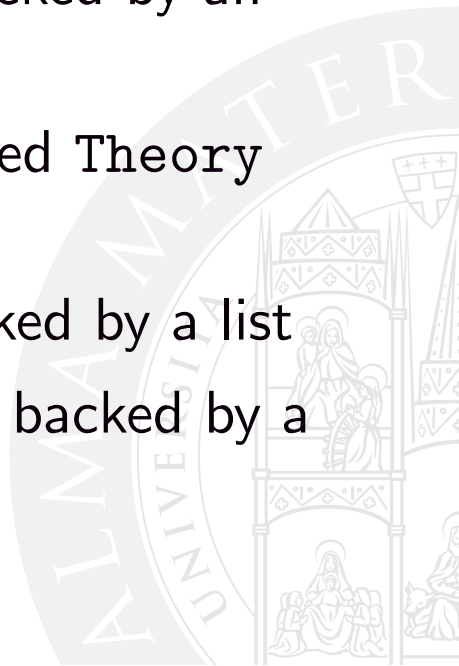
`of(vararg Clause): Theory` creates a `Theory` containing the given clauses

`emptyIndexed(): Theory` creates an empty `Theory` backed by an indexed data structure

`indexedOf(vararg Clause): Theory` creates an indexed `Theory` containing the given `Clauses`

`emptyListed(): Theory` creates an empty `Theory` backed by a list

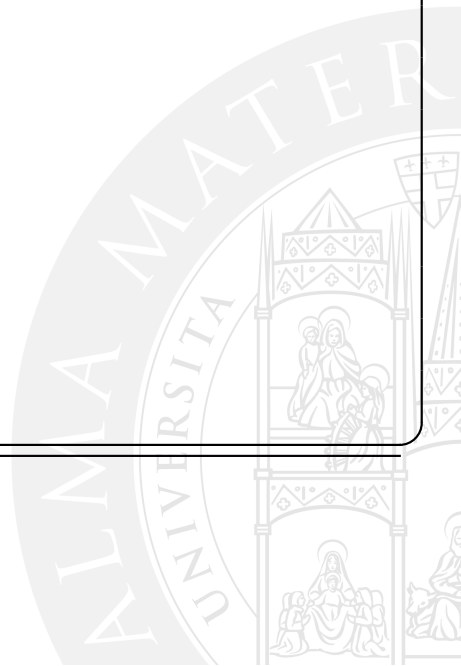
`listedOf(vararg Clause): Theory` creates a `Theory` backed by a list, containing the given clauses



Using a Theory I

Asserting clauses into a Theory:

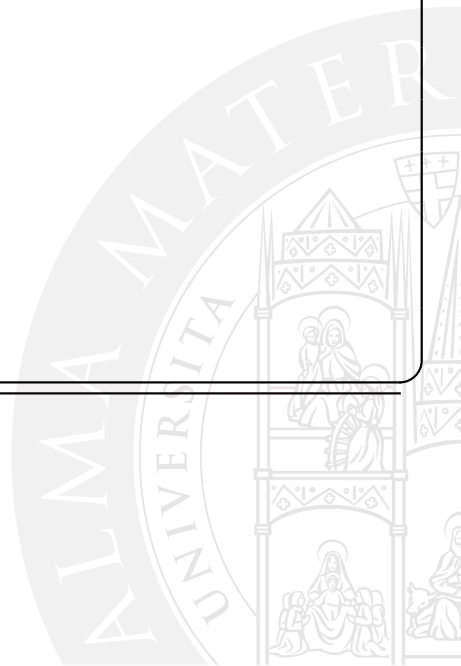
```
val theory = Theory.empty()

val theory2 = theory.assertA(
    Fact.of(Struct.of("f", Integer.of(1)))
) // f(1)

val theory3 = theory2.assertA(
    Fact.of(Struct.of("f", Integer.of(2)))
) // f(2), f(1)

val theory4 = theory3.assertZ(
    Fact.of(Struct.of("f", Integer.of(3)))
) // f(2), f(1), f(3)
```



Using a Theory II

Abolishing Indicators from a Theory:

```
val theory = Theory.of(
    Fact.of(Struct.of("f", Integer.of(1))),
    Fact.of(Struct.of("f", Integer.of(2))),
    Fact.of(Struct.of("f", Integer.of(3))),
    Fact.of(Struct.of("g", Integer.of(1))),
    Fact.of(Struct.of("g", Integer.of(2))),
) // f(1), f(2), f(3), g(1), g(2)

val theory2 = theory.abolish(
    Indicator.of("f", 1) // retracts all f/1 clauses
) // g(1), g(2)
```



Using a Theory III

Retracting clauses from a Theory:

```
val theory = Theory.of(
    Fact.of(Struct.of("f", Integer.of(1))),
    Fact.of(Struct.of("f", Integer.of(2))),
    Fact.of(Struct.of("f", Integer.of(3))),
) // f(1), f(2), f(3)

val successfulRetract = theory.retract(
    Fact.of(Struct.of("f", Integer.of(1)))
)

println(successfulRetract.firstClause) // f(1)

val theory2 = successfulRetract.theory // f(2), f(3)

val failedRetract = theory2.retract(
    Fact.of(Struct.of("f", Integer.of(4)))
)
val theory3 = failedRetract.theory // f(2), f(3)
```

Next in Line...

The *:solve* Module: Overview I

Purpose

- Provide generic support to logic resolution
- Re-use as much as possible of Prolog standard
  - without committing to any particular resolution strategy
  - nor to any inference procedure

The *:solve* Module: Overview II

Thus, *:solve* provides the following abstractions:

Solvers and Solutions

Solvers are reactive entities capable of answering to users' queries by producing one or more solutions, exploiting logic resolution

The *:solve* Module: Overview III

Execution Contexts

An execution context encapsulates some solver's internal state. Resolution affects and is affected by the solver's execution context.

Composed by:

libraries — containers of built-in functionalities exploitable by resolution

flags — configurable aspects of a solver

knowledge bases (KB) — containers of the logic knowledge used by resolution

operators — set of operators used to parse queries and to present solutions

channels — I/O facilities to communicate with the external world

+ any resolution-specific aspect
  - (this may vary depending on how resolution is implemented)

The *:solve* Module: Overview IV

Libraries

Containers of built-in functionalities exploitable by resolution

Possibly, including:

some **clauses** containing logic predicates provided by the library

some **primitives** i.e., logic predicates, implemented in Kotlin, and provided by the library

some **functions** i.e., custom ways to reduce expressions of terms

some **operators** automatically imported into the solver along with the library

The *:solve* Module: Overview V

Flags

Key-value pairs aimed at configuring a solver behaviour

Where:

- the key is a string
- the value is an arbitrary term

The *:solve* Module: Overview VI

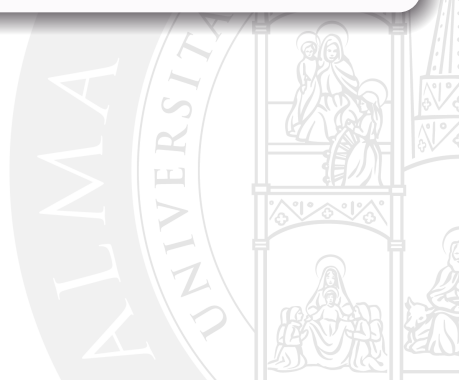
Knowledge bases (KB)

Containers of the logic knowledge taken into account by resolution to answer users' queries

Following Prolog's conventions, KB are of two sorts:

static KB — which cannot be altered during resolution

dynamic KB — which can be altered during resultion



The *:solve* Module: Overview VII

Operators

Each solver may operate upon a dynamic set of operators

That:

- may be loaded along with libraries or be dynamically altered during resolution
- may affect the way users' queries are parsed
- may affect the way solutions are presented to users

The *:solve* Module: Overview VIII

Channels

Communication facilities among the solver and the external world

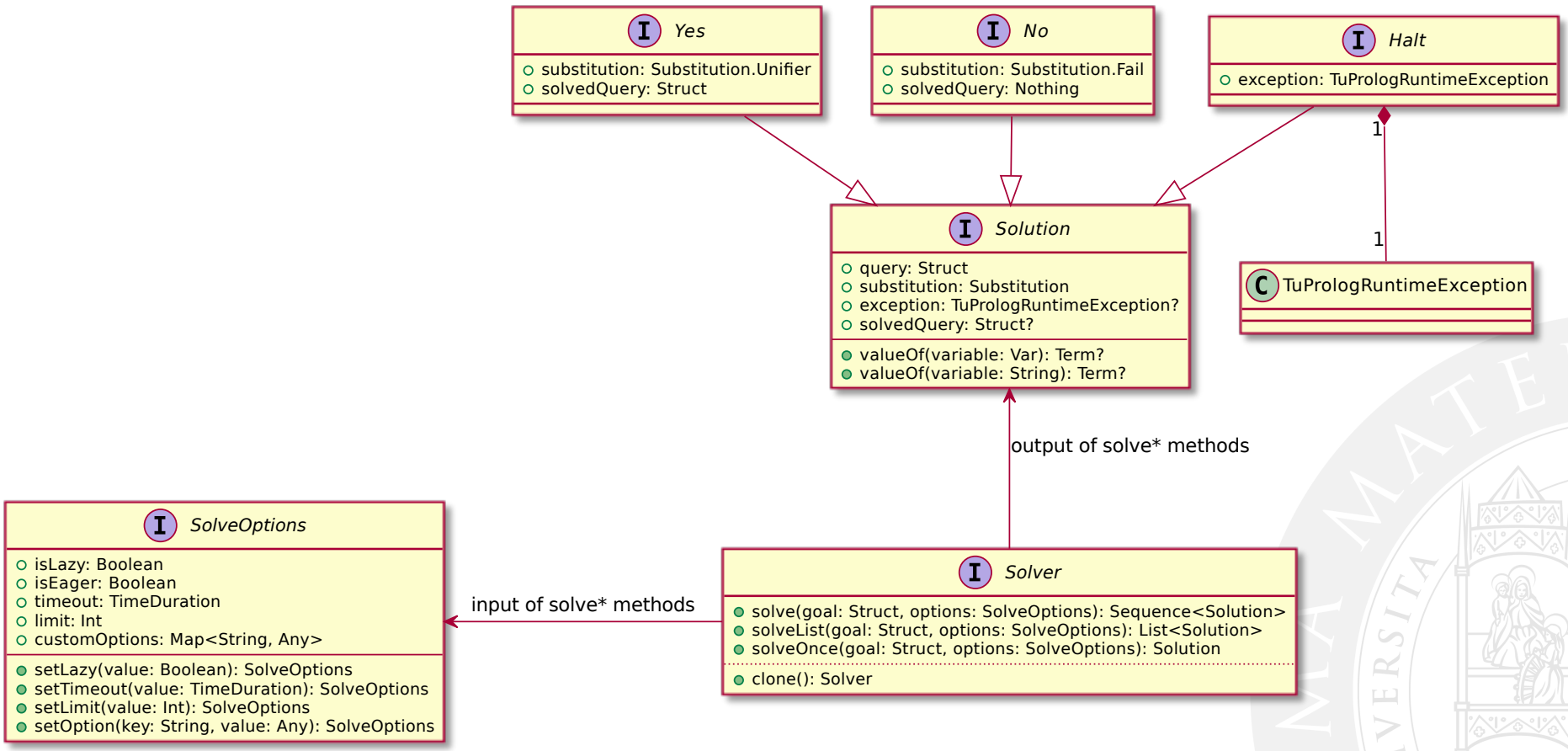
Channels are of two sorts:

input channels (a.k.a. sources) let a solver receive inputs messages

output channels (a.k.a. sinks) let a solver provide output messages

Focus on...

The Solver, Solution, and SolveOptions Types I



The Solver, Solution, and SolveOptions Types II

The Solver type

Solvers expose many solve* methods, accepting:

- a goal, i.e. a Struct representing the query to be answered via resolution
- an instance of SolveOptions describing/constraining the way answers are provided

and returning one or more Solutions

The Solver, Solution, and SolveOptions Types III

The Solution type

A Solution represents an answer provided by a Solver w.r.t. the user's query It can be of three actual sorts:

Solution.Yes representing a positive answer

- carrying a Unifier assigning some variable from the user's query

Solution.No representing a negative answer

Solution.Halt representing an exceptional answer

- carrying a TuPrologRuntimeException locating the error w.r.t. the resolution process

The `Solver`, `Solution`, and `SolveOptions` Types IV

The `SolveOptions` type

Instances of `SolveOptions` are provided along with users' queries and aim at constraining the way solutions are computed by the solver:

`isEager: Boolean` — if `true` all solutions are computed ASAP

- may saturate memory in case of many/infinite solutions!

`isLazy: Boolean` — if `true` solutions are *lazily* computed just before being consumed

- this is `true` by default

`timeout: TimeDuration` — dictates the maximum amount of time resolution may use to compute a single solution

- defaults to infinite time

`limit: Int` — limits the maximum amount of computed solutions

- defaults to infinite

The Solver, Solution, and SolveOptions Types V

The solve* methods in Solver

`solve` — returns a Sequence (i.e., a stream) of Solutions

`solveList` — returns a List of Solutions

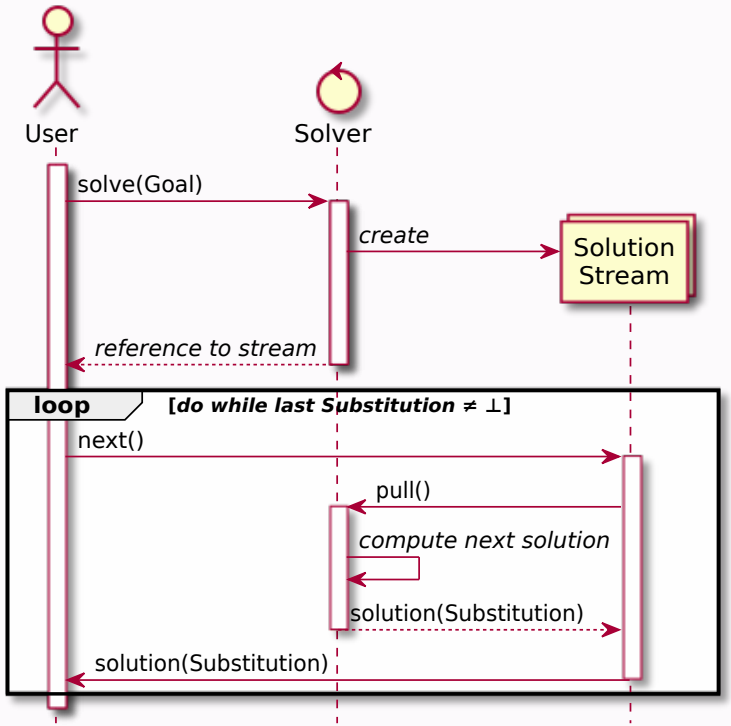
- implies `solveOpts.isEager == true`

`solveOnce` — returns a single Solution

- implies `solveOpts.limit == 1`

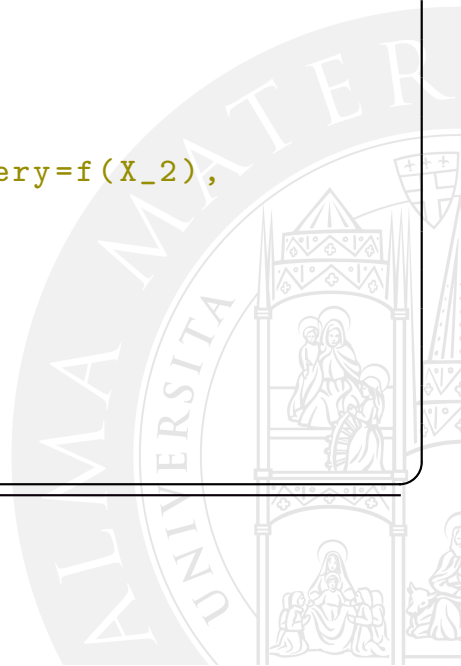
Using a Solver I

Concept



Using a Solver II

```
val prolog = Solver.classic.solverWithDefaultBuiltins(
    staticKb = Theory.indexedOf(
        Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
        Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
        Fact.of(Struct.of("f", Atom.of("c")))  // f(c).
    )
)

val goal = Struct.of("f", Var.of("X"))

val solutions: List<Solution> = prolog.solveList(goal)

println(solutions.size) // 3
println(solutions) // [Yes(query=f(X_2), substitution={X_2=a}), Yes(query=f(X_2),
     substitution={X_2=b}) Yes(query=f(X_2), substitution={X_2=c})]

for (solution in solutions) {
    println(solution.query) // f(X_2), f(X_2), f(X_2)
    println(solution.isYes) // true, true, true
    println(solution.substitution) // {X_2=a}, {X_2=b}, {X_2=c}
}
```



Using a Solver III

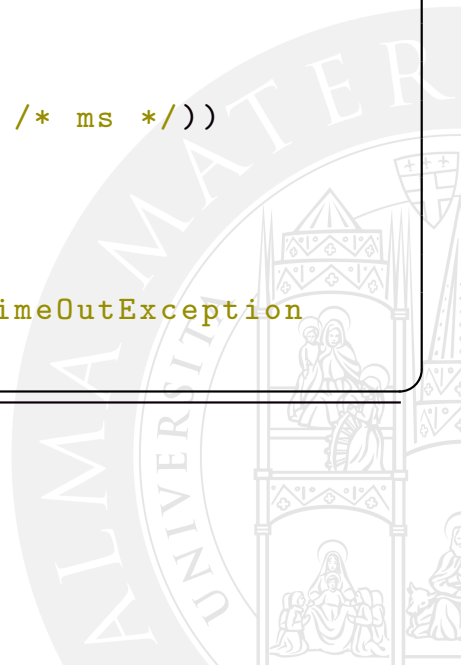
```
val prolog = Solver.classic.solverWithDefaultBuiltins(
    staticKb = Theory.indexedOf(
        Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
        Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
        Fact.of(Struct.of("f", Atom.of("c")))  // f(c).
    )
)

val goal = Struct.of("f", Var.of("X"))

val solutions: Sequence<Solution> = prolog.solve(goal)

println(solutions) // kotlin.sequences.ConstrainedOnceSequence@7f362f22

val i = solutions.iterator()

while (i.hasNext()) {
    val solution = i.next()
    println(solution.query) // f(X_2), f(X_2), f(X_2)
    println(solution.isYes) // true, true, true
    println(solution.substitution) // {X_2=a}, {X_2=b}, {X_2=c}
}
```

Using a Solver IV

```
val prolog = Solver.classic.solverOf()

val goal = Struct.of("f", Var.of("X"))

val solution = prolog.solveOnce(goal)

println(solution) // No(query=f(X_0))
```

Using a Solver V

```
val prolog = Solver.classic.solverWithDefaultBuiltins(
    staticKb = Theory.indexedOf(
        Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
        Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
        Fact.of(Struct.of("f", Atom.of("c")))  // f(c).
    )
)

val goal = Struct.of("f", Var.of("X"))

val solutions = prolog.solve(goal, SolveOptions.allLazilyWithTimeout(1 /* ms */))

for (solution in solutions) {
    println(solution.query) // f(X_2)
    println(solution.isHalt) // true
    println(solution.exception) // it.unibo.tuprolog.solve.exception.TimeOutException
}
```

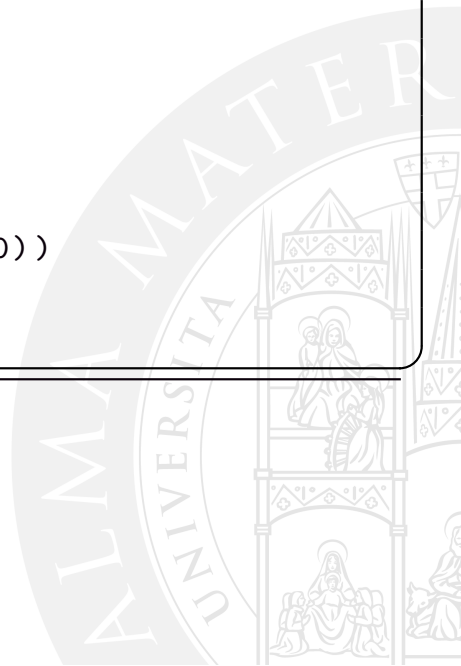


Using a Solver VI

```
val prolog = Solver.classic.solverWithDefaultBuiltins()

val solution = prolog.solveOnce(Struct.of("halt", Integer.of(2)))

println(solution) // Halt(query=halt(2), exception=it.unibo.tuprolog.solve.exception.
    HaltException)
println((solution.exception as HaltException).exitStatus) // 2
```

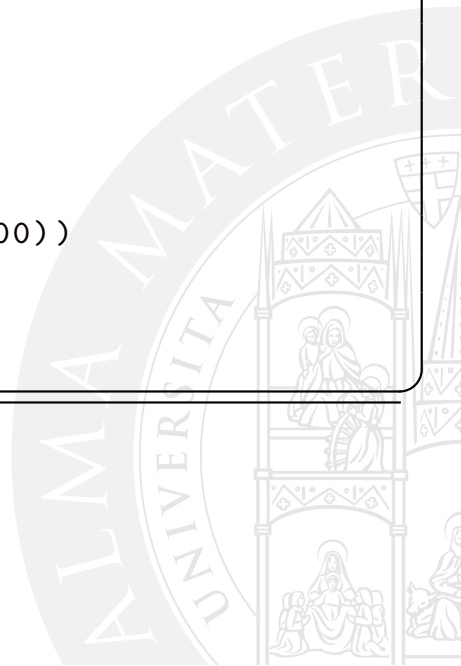
Using a Solver VII

```
val prolog = Solver.classic.solverOf(
    staticKb = Theory.of(
        { factOf(structOf("nat", atomOf("z"))) },           // nat(z).
        {
            ruleOf(
                structOf("nat", structOf("s", varOf("Z"))), // nat(s(Z)) :-
                structOf("nat", varOf("Z"))                 //   nat(Z).
            )
        }
    )
)

val goal = Struct.of("nat", Var.of("X"))
val solutions = prolog.solve(goal, SolveOptions.someLazily(limit = 1000))

println(solutions)
```



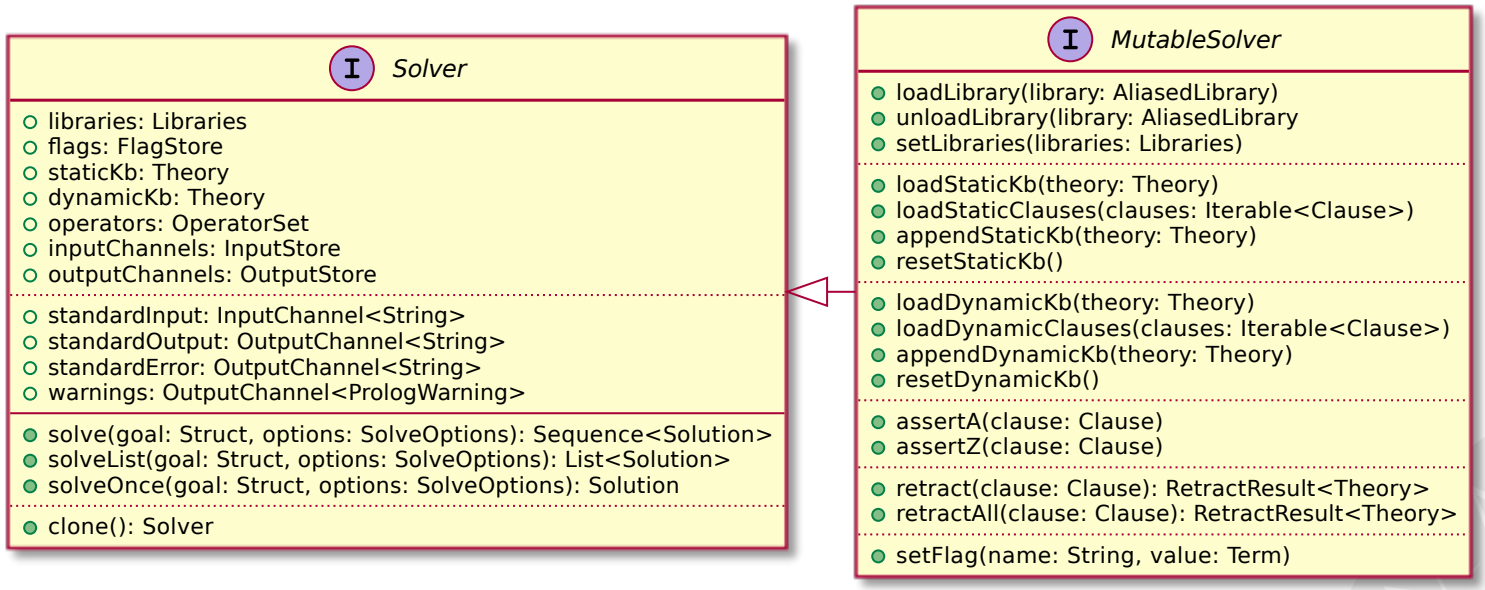
Using a Solver VIII

```
val prolog = Solver.classic.solverOf(
    staticKb = Theory.of(
        { factOf(structOf("nat", atomOf("z"))) },            // nat(z).
        {
            ruleOf(
                structOf("nat", structOf("s", varOf("Z"))), // nat(s(Z)) :-
                structOf("nat", varOf("Z"))                 //   nat(Z).
            )
        }
    )
)

val goal = Struct.of("nat", Var.of("X"))
val solutions = prolog.solve(goal, SolveOptions.someEagerly(limit = 1000))

// AFTER A LONG WHILE
println(solutions)
```



Solver vs. MutableSolver I

Notice that there are 2 types for solvers:



- *solvers* expose properties supporting inspection of their state
- mutable solvers expose methods to alter their state
  - ! the state of *non*-mutable solvers may change too, because of resolution
    - → they are not immutable

Solver vs. MutableSolver II

```
import it.unibo.tuprolog.solve.classic.stdlib.DefaultBuiltins

val theory = Theory.of(
    Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
    Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
    Fact.of(Struct.of("f", Atom.of("c")))  // f(c).
)
val fact = Struct.of("g", Integer.ONE)  // g(1).

// solvers require information to be provided at instantiation time:
val solver: Solver = Solver.classic.solverOf(
    libraries = Libraries.of(DefaultBuiltins),
    staticKb = theory
)
// ... side effects must be provoked through resolution
solver.solveOnce(Struct.of("assert", fact))
println(solver.dynamicKb) // MutableIndexedTheory{ g(1) :- true }

// mutable solvers can be lately configured:
val mutableSolver: MutableSolver = Solver.classic.mutableSolverOf()
mutableSolver.loadLibrary(DefaultBuiltins)
mutableSolver.loadStaticKb(theory)
// ... side effects can be provoked via ad-hoc methods
mutableSolver.assertZ(fact)
println(solver.dynamicKb) // MutableIndexedTheory{ g(1) :- true }
```

Creating Solvers via a SolverFactory I

About solver factories

Objects of type `SolverFactory` aim at instantiating `Solvers`

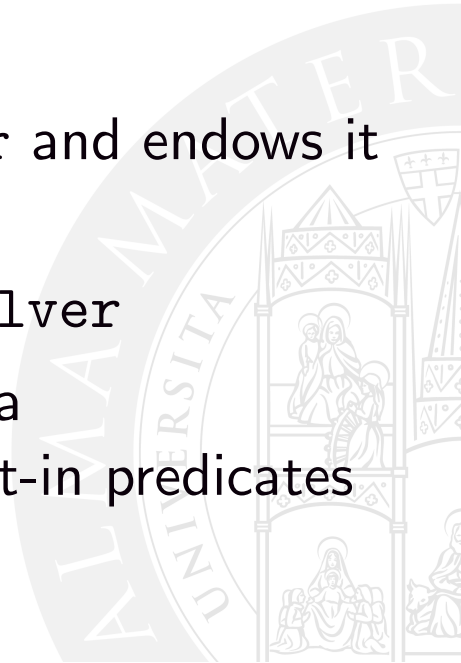
Main sorts of methods:

`solverOf(...)` — creates an empty `Solver`

`solverWithDefaultBuiltins(...)` — creates a `Solver` and endows it with some default built-in predicates

`mutableSolverOf(...)` — creates an empty `MutableSolver`

`mutableSolverWithDefaultBuiltins(...)` — creates a `MutableSolver` and endows it with some default built-in predicates



Creating Solvers via a SolverFactory II

- All such methods come with various overloads...
- ...letting the caller specify the new solver's:
  1. libraries to be loaded upon construction
     - (possibly, other than the default ones)
  2. initial values of specific flags
  3. content of the static KB
  4. initial content of the dynamic KB
  5. operators to be loaded upon construction
  6. input/output channels to be used for the solver
  7. or some sub-set of the above

Creating Solvers via a SolverFactory III

Important convention

All implementations of `Solver` should be packed with a corresponding implementation of `SolveFactory`, possibly as a singleton

eg *:solve-classic* exposes object `ClassicSolverFactory`

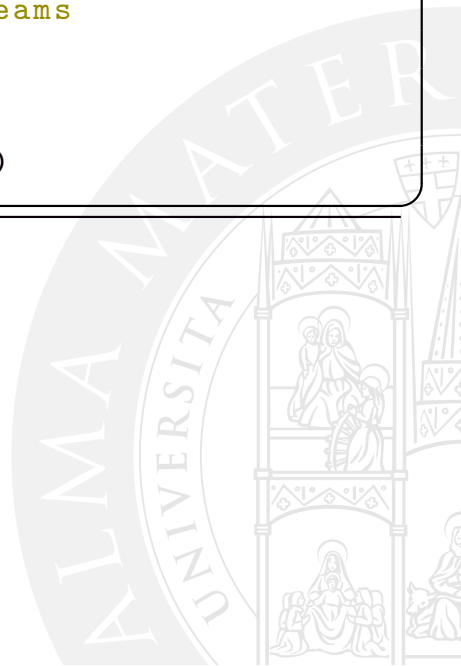
- also accessible via `Solver.classic`

eg *:solve-streams* exposes object `StreamsSolverFactory`

- also accessible via `Solver.streams`

Creating Solvers via a SolverFactory IV

```
import it.unibo.tuprolog.solve.classic.ClassicSolverFactory
import it.unibo.tuprolog.solve.streams.StreamsSolverFactory

val classicFactory = ClassicSolverFactory // alternatively, Solver.classic
val streamsFactory = StreamsSolverFactory // alternatively, Solver.streams

val classicSolver: Solver = classicFactory.solverWithDefaultBuiltins()

val streamsFactory: Solver = streamsFactory.solverWithDefaultBuiltins()
```



Creating Solvers via a SolverFactory V

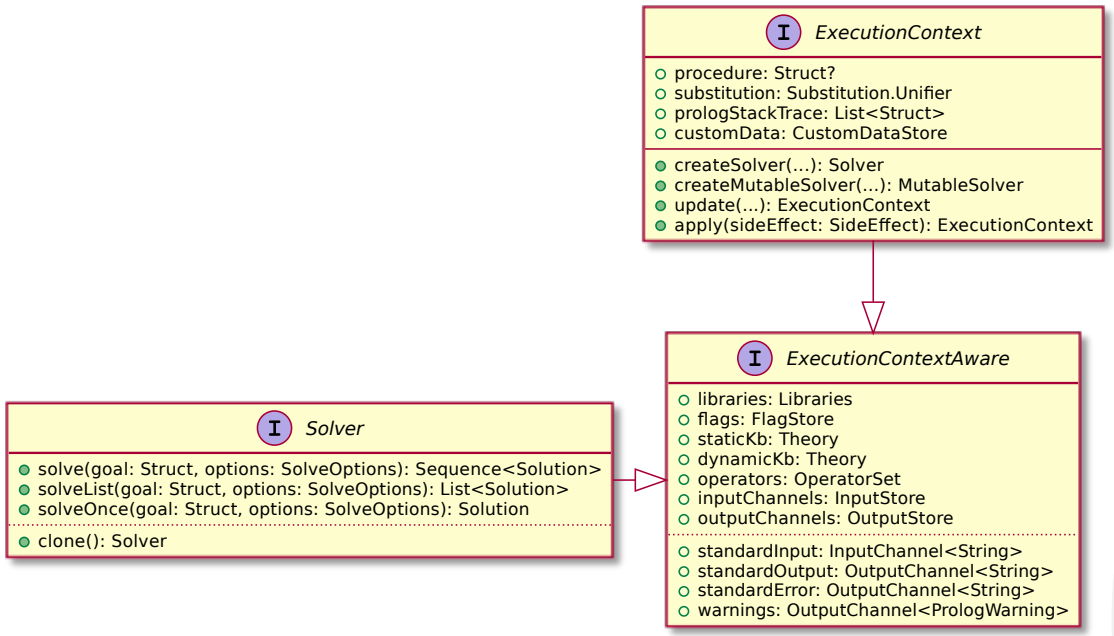
Solver factories support writing implementation-agnostic tests for Solvers:

```
import kotlin.test.Test

class GeneralTestSuite(private val solverFactory: SolverFactory) {
    fun testCase() {
        val solver = solverFactory.solverOf()
        // assertions here
    }
}

import it.unibo.tuprolog.solve.classic.ClassicSolverFactory
class ClassicTestSuite : GeneralTestClass(ClassicSolverFactory) {
    @Test
    override fun testCase() = super.testCase()
}

import it.unibo.tuprolog.solve.classic.StreamsSolverFactory
class StreamsTestSuite : GeneralTestClass(StreamsSolverFactory) {
    @Test
    override fun testCase() = super.testCase()
}
```

Focus on...

Overview on the general purpose ExecutionContext I

- Module *:solve* exposes a general interface for ExecutionContexts
- This must be exploited & customised by implementers of Solvers



Overview on the general purpose ExecutionContext II

An ExecutionContext keeps track of (at least)

- all inspectable aspects of a Solver
  - ie libraries, flags, KB, operators, and channels
- \+ the procedure resolution is currently focussing upon
- \+ the substitution resolution has constructed so far
- \+ other fields possibly specified/customised by implementers

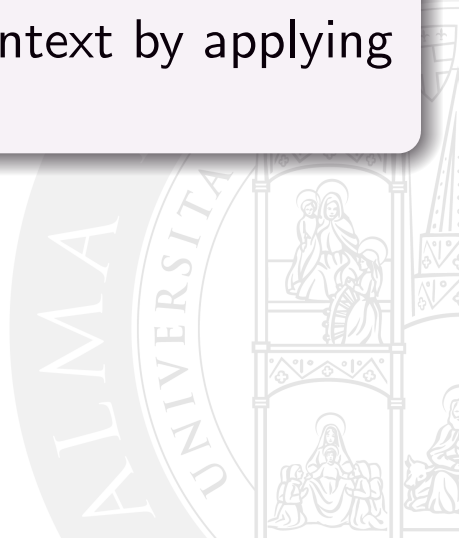
Overview on the general purpose ExecutionContext III

Conventions

- ExecutionContexts are immutable
- An ExecutionContext can be attained from another one via:

  `update(...)` — creating a copy of the current context, only differing for the provided fields

  `apply(SideEffect)` — creating a copy of the current context by applying the provide SideEffect to it

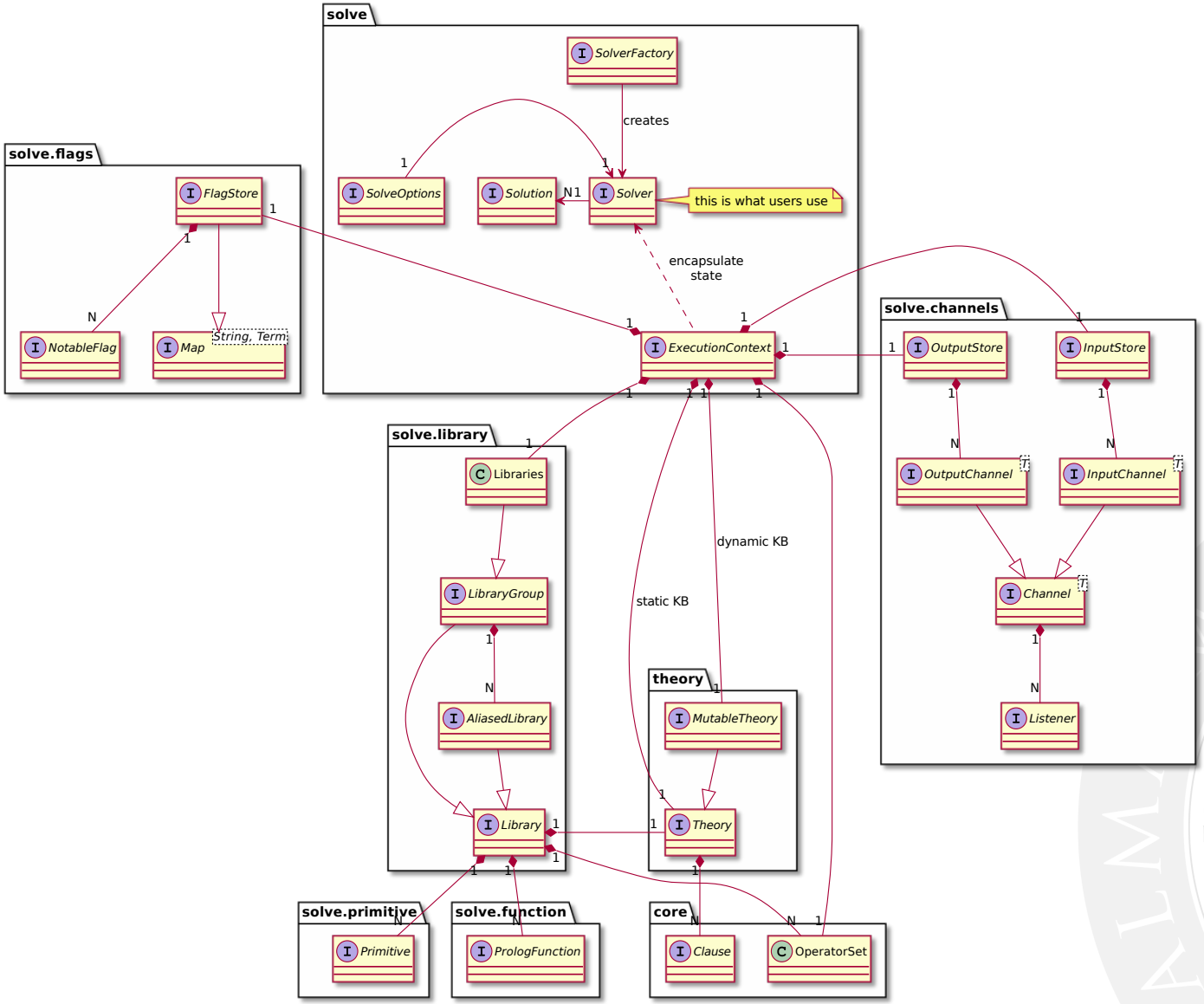


Overview on the general purpose ExecutionContext IV

Notice that

- It is always possible to create a novel `(Mutable)Solver` via
  `createSolver(...)`
  `createMutableSolver(...)`
- The new solver is initialised using the current context
  - ! this is fundamental to realise complex functionalities

ExecutionContext is where everything is linked together



Static vs. Dynamic Knowledge bases I

Conventions

- The `staticKb` is assumed to reference an immutable `Theory`
  - ie read efficient
- The `dynamicKb` is assumed to reference a `MutableTheory`
  - ie write efficient
- Resolution is generally only allowed to alter the dynamic KB
  - eg via `assert/1`, or `retract/1`
- However, notable exceptions exist
  - eg via `consult/1`, or `register/2` (from the *:oop-lib* module)

Example: Using `Solvers` with custom KB I

Endowing `Solvers` with their KB

- `Solvers` can be provided with their KB upon construction
  - cf. `SolverFactory`'s methods
- `MutableSolvers` can be provided with their KB dynamically

  eg via the many methods of `MutableSolver`

Example: Using Solvers with custom KB II

```
val theory1 = Theory.of(
    Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
    Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
)
val theory2 = Theory.of(
    Fact.of(Struct.of("g", Integer.of(1))), // g(1).
    Fact.of(Struct.of("g", Integer.of(2))), // g(2).
)

val solver = Solver.classic.mutableSolverWithDefaultBuiltins()
println(solver.staticKb) // IndexedTheory{  }
println(solver.dynamicKb) // MutableIndexedTheory{  }

solver.loadStaticKb(theory1)
solver.loadDynamicKb(theory2)
println(solver.staticKb) // IndexedTheory{ f(a) :- true. f(b) :-
    true }
println(solver.dynamicKb) // MutableIndexedTheory{ g(1) :- true. g
    (2) :- true }
```

Example: Using Solvers with custom KB III

```
val theory1 = Theory.of(
    Fact.of(Struct.of("f", Atom.of("a"))), // f(a).
    Fact.of(Struct.of("f", Atom.of("b"))), // f(b).
)
val theory2 = Theory.of(
    Fact.of(Struct.of("g", Integer.of(1))), // g(1).
    Fact.of(Struct.of("g", Integer.of(2))), // g(2).
)

val solver = Solver.classic.solverWithDefaultBuiltins(
    staticKb = theory1,
    dynamicKb = theory2
)

println(solver.staticKb) // IndexedTheory{ f(a) :- true. f(b) :-
    true }
println(solver.dynamicKb) // MutableIndexedTheory{ g(1) :- true. g
    (2) :- true }
```

Custom KB and Directives I

About directives

- KB may contain some `Directives`
- Implementations of `Solver` should react to particular `Directives`
  - upon KB provisioning (either by construciton or otherwise)
  - other sorts of directives are ignored
- Notice that `Directives` are just headless `Clauses`
  - unless they are provided to a `Solver`
  - at that point, they acquire procedural meaning

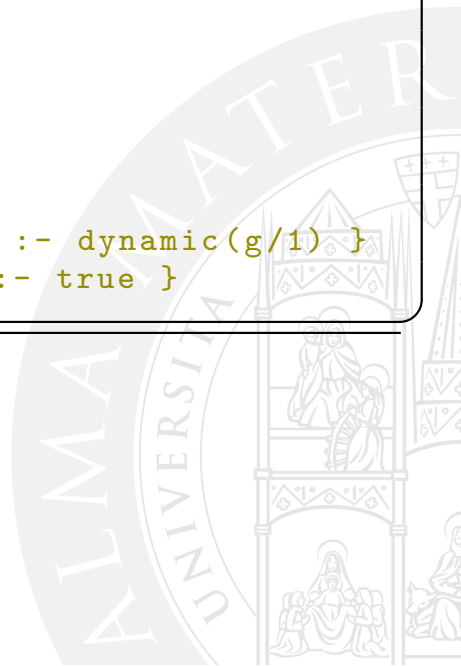
Custom KB and Directives II

Notable directives:

- `:- dynamic(F / N).` makes any subsequent clause whose head indicator is *F/N* be loaded into the dynamic KB, event if the theory is loaded as static
- `:- static(F / N).` dual w.r.t. `dynamic(F/N)`
- `:- op(P, S, F).` loads a new operator into the `Solver`, having the provided propriety (*P*), specifier (*S*), and functor (*F*)
- `:- solve(G).` executes goal *G* as soon as the theory has been loaded
- `:- initialization(G).` alias for `solve(G)`
- `:- set_prolog_flag(N, V).` (re)sets the flag named *N* to the value *V*
- `:- include(P).` loads the theory whose path is *P* as if it was part of the currently loading one
  - ! requires the *:io-lib* module to be available at run-time
- `:- load(P).` alias for `include(P)`

Custom KB and Directives III

```
val theory = Theory.of(
    Fact.of(Struct.of("f", Atom.of("a"))),                    // f(a).
    Fact.of(Struct.of("f", Atom.of("b"))),                    // f(b).
    Directive.of(Struct.of("dynamic", Indicator.of("g", 1))), // :- dynamic(g/1).
    Fact.of(Struct.of("g", Integer.of(1))),                   // g(1).
    Fact.of(Struct.of("g", Integer.of(2))),                   // g(2).
)

val solver = Solver.classic.solverWithDefaultBuiltins(
    staticKb = theory
)

println(solver.staticKb) // IndexedTheory{ f(a) :- true. f(b) :- true. :- dynamic(g/1) }
println(solver.dynamicKb) // MutableIndexedTheory{ g(1) :- true. g(2) :- true }
```



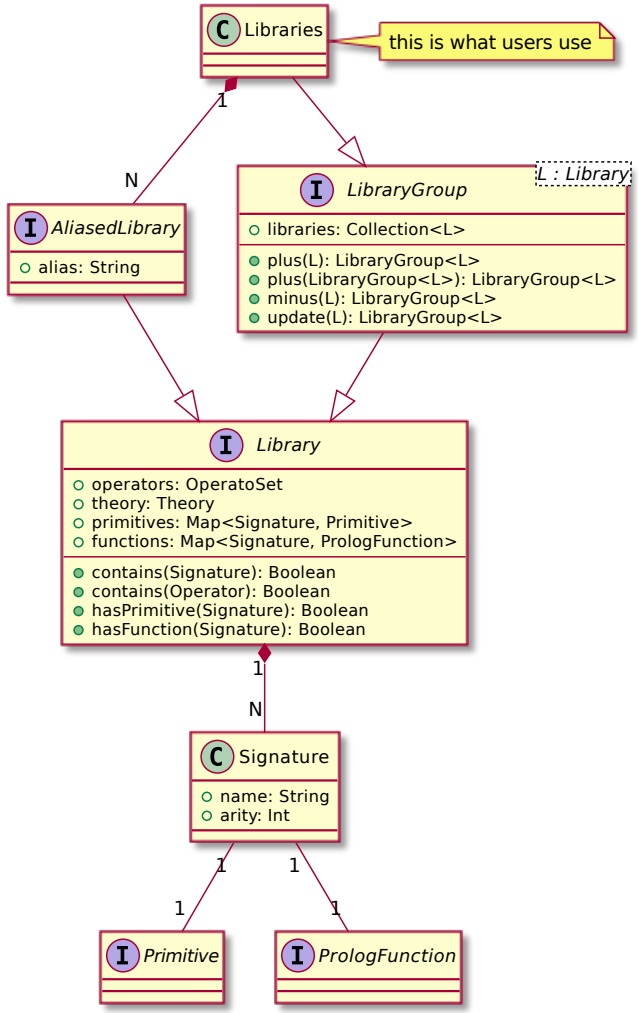
Overview on Libraries I

Why Libraries

- Need to extend Solvers via built-in functionalities

eg operators, rules, (logic) functions, or primitives

Overview on Libraries II



Overview on Libraries III

Definition: Library

A type for immutable containers of:

operators to be loaded along with the Library

theory in turn containing pre-built rules and facts

primitives i.e. logic predicates, written in Kotlin

functions i.e. logic functions, written in Kotlin

Definition: Signature

Primitives and functions are indexed in Librarys via Signatures,

ie name-arity pairs

Overview on Libraries IV

Definition: AliasedLibrary

A Library with an alias

- think about aliases as Java packages names

Conventions on **aliases**

- Same of Java packages names

ie camel case, dot separated, space-less

eg prolog.lang, prolog.io

Overview on Libraries V

Definition: LibraryGroup

A Library attained by collecting several Librarys

Overview on Libraries VI

Definition: Libraries

A group of AliasedLibrary, indexed by alias

- this is what Solvers' users use, in practice

Overview on Libraries VII

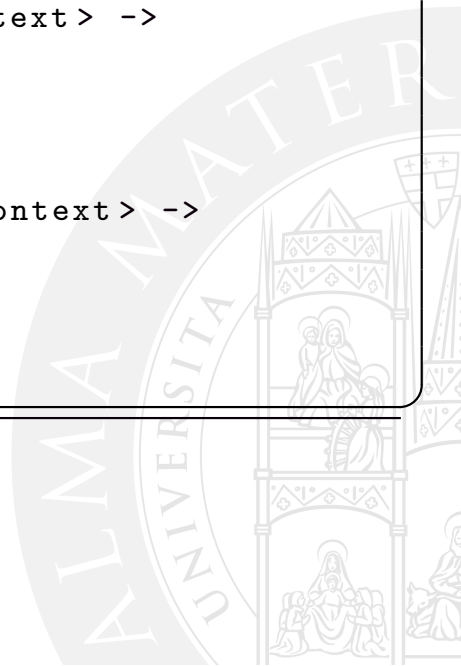
Example 1 – Using Libraries:

```
import it.unibo.tuprolog.solve.classic.stdlib.DefaultBuiltins
import it.unibo.tuprolog.solve.libs.io.IOLib
import it.unibo.tuprolog.solve.libs.oop.OOPLib

val solver = Solver.classic.solverOf(
    libraries = Libraries.of(DefaultBuiltins, IOLib, OOPLib)
)

println(solver.libraries.keys) // [prolog.lang, prolog.io, prolog.
    oop]
```

Overview on Libraries VIII

Example 2 – Defining custom Libraries:

```
object MyLibrary : AliasedLibrary by
    Library.aliased(
        alias = "alias.of.the.lib",
        operatorSet = OperatorSet(/* custom operators here */),
        theory = Theory.of(/* provide rules/facts here */),
        primitives = mapOf(
            Signature("f", 1) to { request: Solve.Request<ExecutionContext> ->
                // implement primitive here
            }
        ),
        functions = mapOf(
            Signature("+", 2) to { request: Compute.Request<ExecutionContext> ->
                // implement function here here
            }
        )
    )
```



The Primitive Type I

Problem

- Some built-in logic functionalities are easier to write in Kotlin
- . . . as they require altering the Solver's ExecutionContext
  eg `assert/1`, `retract/1`, `write/1`
- Some logic functionalities may exploit external computational facilities

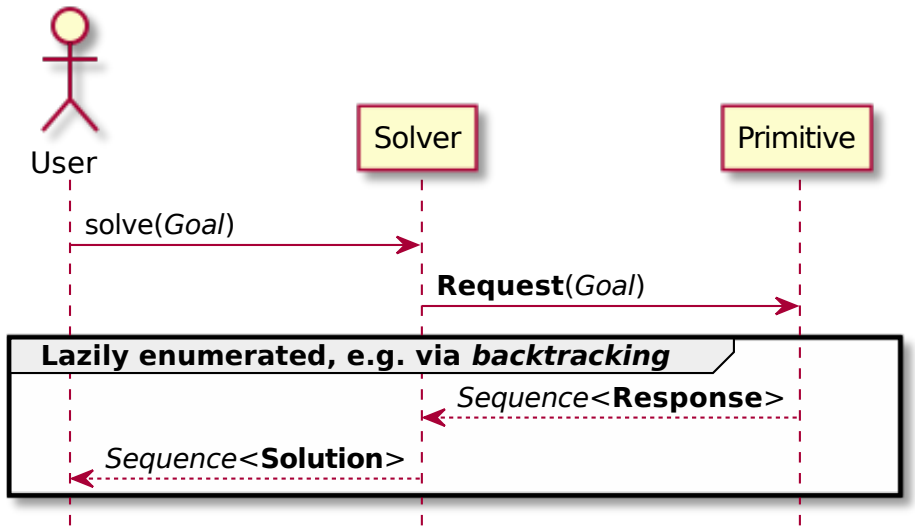
Why Primitives

- A means for calling Kotlin code from LP
- A means for writing logic relations via OOP+FP+IP

The Primitive Type II

Client-server metaphor:

- users are clients for Solvers, which act as servers
- Solvers are clients for Primitives, which act as servers



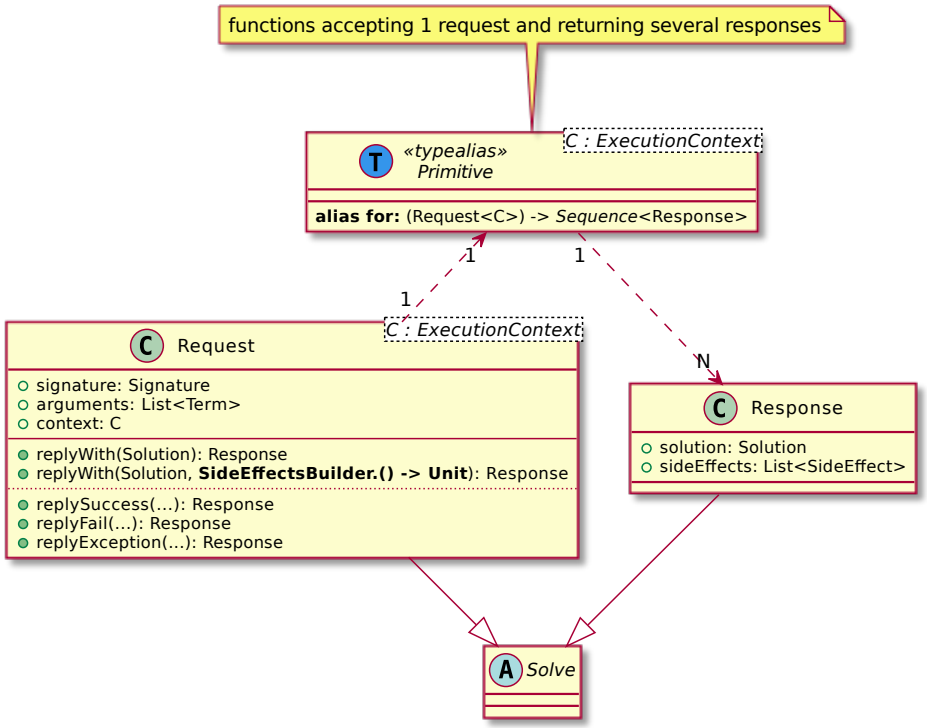
The Primitive Type III

When needing to solve a (sub-)goal G, a Solver may

- look into its KB, or
- pick a Primitive P, compliant with G, from a Library
  1. send a request, describing G, to P
  2. receive several responses, describing solutions to G, from P
     - possibly containing some *side effect*, to be reified

The Primitive Type IV

In code:



The Primitive Type V

```
val gt = Signature("gt", 2)

fun greaterThan(req: Solve.Request<ExecutionContext>): Sequence<Solve.Response> {
    val arg1: Term = req.arguments[0] ; val arg2: Term = req.arguments[1]

    if (arg1 !is Numeric)
        throw TypeError.forArgument(req.context, req.signature, TypeError.Expected.NUMBER
     , arg1, 0)
    if (arg2 !is Numeric)
        throw TypeError.forArgument(req.context, req.signature, TypeError.Expected.NUMBER
     , arg2, 1)

    return if (arg1.castToNumeric().decimalValue > arg2.castToNumeric().decimalValue)
        sequenceOf(req.replySuccess())
    else
        sequenceOf(req.replyFail())
}

val mylib = Library.aliased(alias = "prolog.mylib", primitives = mapOf(gt to ::
     greaterThan))

val solver = Solver.classic.solverOf(Libraries.of(mylib))

println(solver.solveOnce(Struct.of("gt", Integer.ONE, Integer.ZERO))) // yes
println(solver.solveOnce(Struct.of("gt", Integer.ZERO, Integer.ONE))) // no
println(solver.solveOnce(Struct.of("gt", Integer.ONE, Atom.of("a")))) // type_errors
```

The Primitive Type VI

```
val nat = Signature("nat", 1)

fun natural(req: Solve.Request<ExecutionContext>): Sequence<Solve.Response> {
    return when (val arg1: Term = req.arguments[0]) {
        is Numeric -> sequenceOf(
            if (arg1.intValue >= BigInteger.ZERO) req.replySuccess() else req.replyFail()
        )
        is Var -> generateSequence(0) { it + 1 }.map { Integer.of(it) }
                .map { it mguWith arg1 }.map { req.replyWith(it) }
        else -> throw TypeError.forArgument(req.context, req.signature, TypeError.
     Expected.NUMBER, arg1, 0)
    }
}

val mylib = Library.aliased(alias = "prolog.mylib", primitives = mapOf(nat to ::natural))

val solver = Solver.classic.solverOf(Libraries.of(mylib))

println(solver.solveOnce(Struct.of("nat", Integer.ONE))) // yes
println(solver.solveOnce(Struct.of("nat", Integer.MINUS_ONE))) // no
println(solver.solveOnce(Struct.of("nat", Atom.of("a")))) // type_error
println(solver.solve(Struct.of("nat", Var.of("X"))).take(3).toList()) // 0, 1, 2
```

The Primitive Type VII

The many sides of the Primitives

Primitive are:

- *N*-ary relations, in the eyes of LP
- servers & data producers, in the eyes of Solvers
- callbacks, in the eyes of libraries implementers

The Primitive Type VIII

Within Primitives, Requests are

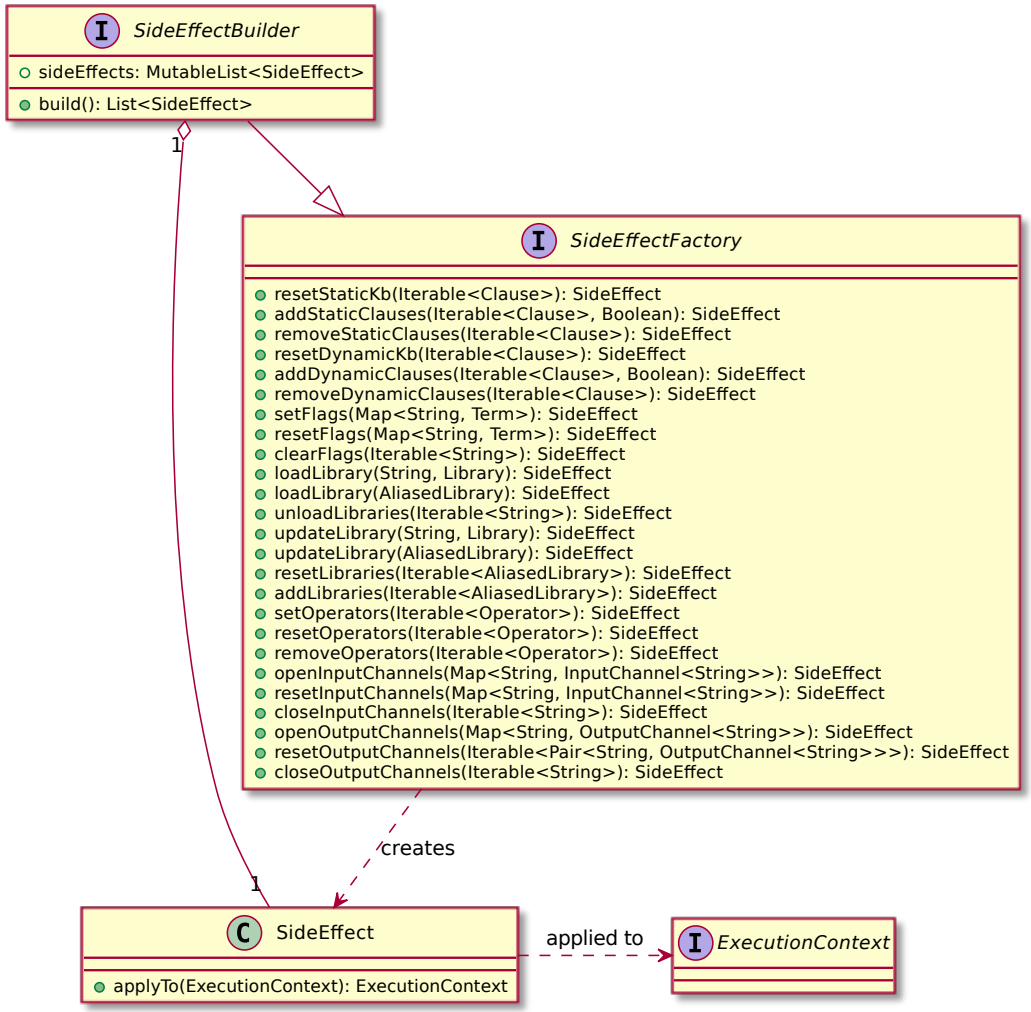
- descriptors of the current ExecutionContext
- descriptors of the actual arguments of the (sub-)goal
- factories of Responses
  - possibility to specify success / failure / exceptions
  - possibility to provoke SideEffects

Primitives provoking SideEffects I

Definition: SideEffects

- Instances of `SideEffect` represent an edit to be perfomed to some `ExecutionContext`
- Only differences are represented
  - eg clauses to be added / removed to the static / dynamic KB
  - eg flags to be set / removed by name
  - eg channels to be opened / closed by name
- `SideEffects` can be applied to `ExecutionContext`
  - producing another `ExecutionContext`
  - only differing from the former one for the edit represented by the `SideEffect`
- `SideEffect` creation $\neq$ `SideEffect` application
  - analogy with lambda expressions

Primitives provoking SideEffects II



Primitives provoking SideEffects III

SideEffects creation

- Side effects are created along with Responses
- ...via SideEffectsBuilders
  - overloads exist for each Response.reply* method
  - supporting the syntax:

    request.reply*(solution) { ⟨*add side effects here*⟩ }

Primitives provoking SideEffects IV

```
val assert_all = Signature("assert_all", 1)

fun assertAll(req: Solve.Request<ExecutionContext>): Sequence<Solve.Response> {
    return when (val arg1: Term = req.arguments[0]) {
        is List -> req.replySuccess {
            addDynamicClauses(arg1.toList().map { Rule.of(it as Struct) })
        }
        else -> throw TypeError.forArgument(req.context, req.signature, TypeError.
     Expected.LIST, arg1, 0)
    }
    .let { sequenceOf(it) }
}

val mylib = Library.aliased(
    alias = "prolog.mylib",
    primitives = mapOf(assert_all to ::assertAll))

val solver = Solver.classic.solverOf(Libraries.of(mylib))

val factsToAdd = List.of(Atom.of("a"), Atom.of("b"), Atom.of("c"))

println(solver.solveOnce(Struct.of("assert_all", factsToAdd))) // yes
println(solver.dynamicKb) // a. b. c.
```

The PrologFunction Type I

Problem

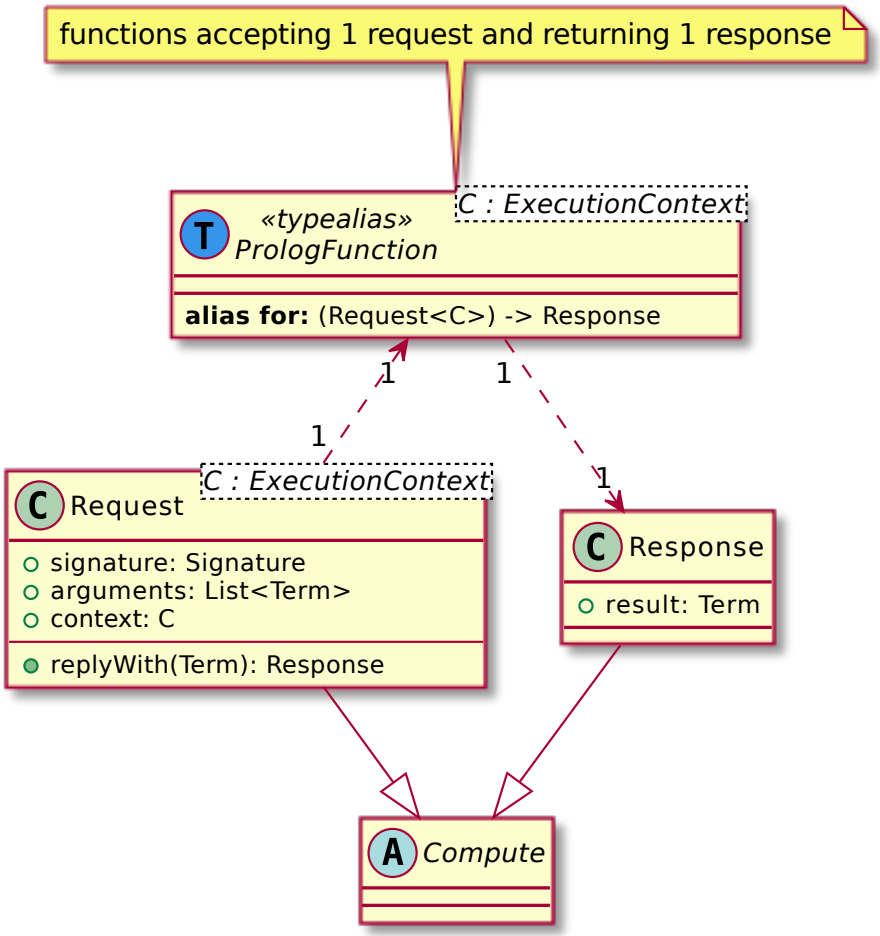
- Some predicates may require logic expressions to be evalued / reduced
  - eg `X is Y + 1` (which is equal to `is(X, '+'(Y, 1))`)
- Evaluation may involve several functions
  - ie objects consuming $N \geq$ terms a producing a result
    - which is, in turn, a `Term`
- Plugging custom functions into a `Solver` should be possible

Why PrologFunctions

- A means for calling Kotlin code from LP
  - with the purpose of evaluating an expression

The PrologFunction Type II

Design very similar to Primitives:

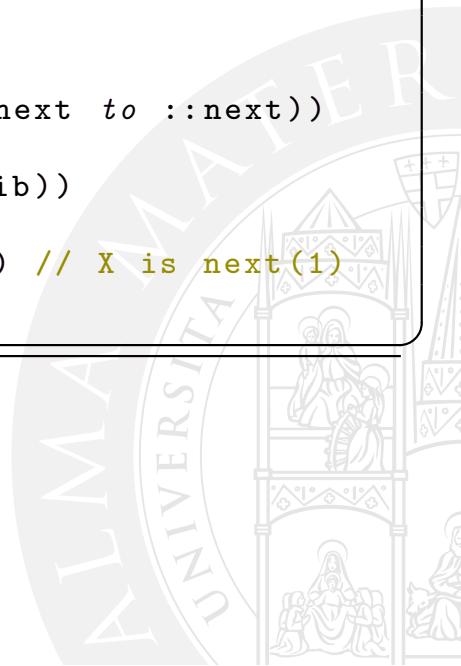


The `PrologFunction` Type III

! no support for side effects, nor multiple responses

The PrologFunction Type IV

```
val next = Signature("next", 1)

fun next(req: Compute.Request<ExecutionContext>): Compute.Response {
    return when(val arg1: Term = req.arguments[0]) {
        is Integer -> req.replyWith(Integer.of(arg1.intValue + BigInteger.ONE))
        else -> req.replyWith(arg1)
    }
}

val mylib = Library.aliased(alias = "prolog.mylib", functions = mapOf(next to ::next))

val solver = Solver.classic.solverWithDefaultBuiltins(Libraries.of(mylib))

val goal = Struct.of("is", Var.of("X"), Struct.of("next", Integer.ONE)) // X is next(1)
println(solver.solveOnce(goal)) // X=2
```



The PrologFunction Type V

About PrologFunctions usage

Expression evaluation:

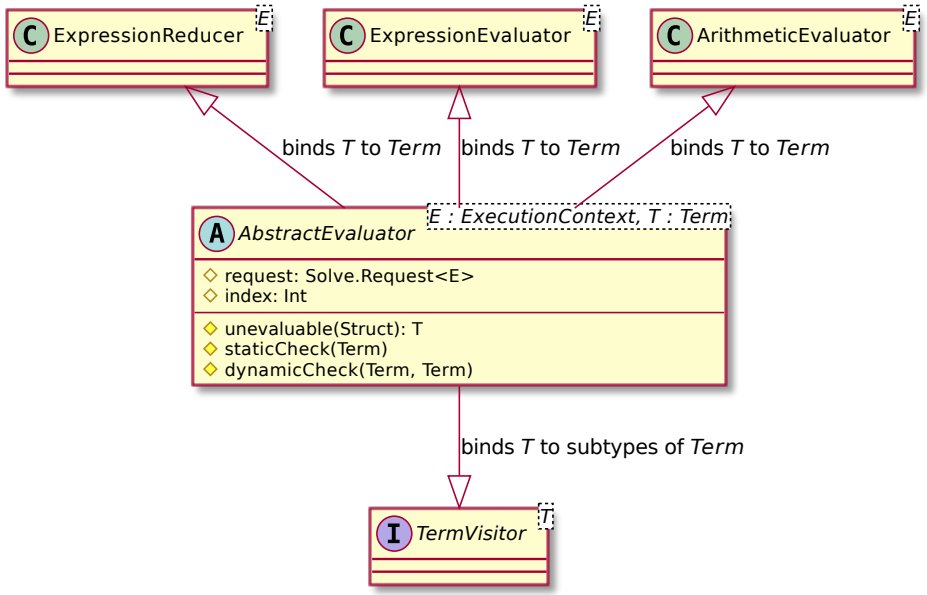
- is expected to only occur as part of some Primitive execution
- is likely to involve several PrologFunctions
- must be atomic in the eyes of the Solver

Example: the is/2 predicate

- Prolog's is/2 is both an operator and a primitive in 2P-Kt
- Whenever a goal of the form is(X, Expr) is met:
  - Expr is evalued, by applying all possible functions
  - the resulting term is bound to X

The PrologFunction Type VI

Evaluators aim to "apply all possible functions" to an expression:



where

- "all possible functions" = *"all functions currently loaded into the `Solver`"*

The PrologFunction Type VII

Sorts of evaluators

`ExpressionReducer` reduces all sub-expressions matching some function, leaving others unreduced

- sub-expressions are evaluated using a post-order visit

eg `g(f(1 + 2), 3 + 4)` → `g(f(3), 7)`

`ExpressionEvaluator` attempts to evaluate all sub-expressions matching some function, throws an error if some does not

- sub-expressions are evaluated using a post-order visit

eg `(1 + 2) - (3 + f(x))` → error as there is no function for `f/1`

`ArithmeticEvaluator` like `ExpressionEvaluator`, but only leverages on arithmetic functions

ie functions accepting and returning `Numeric` arguments

The PrologFunction Type VIII

```
val reduce = Signature("reduce", 2)

fun reduce(req: Solve.Request<ExecutionContext>): Sequence<Solve.Response> {
    val arg1: Term = req.arguments[0]
    return when (val arg2: Term = req.arguments[1]) {
        is Var -> throw InstantiationError.forArgument(req.context, req.signature, arg2,
     1)
        else -> {
            val reducer = ExpressionReducer(req, 1)
            req.replyWith(arg1 mguWith arg2.accept(reducer))
        }
    }.let { sequenceOf(it) }
}

val mylib = Library.aliased(
    alias = "prolog.mylib",
    primitives = mapOf(reduce to ::reduce))

val solver = Solver.classic.solverWithDefaultBuiltins(Libraries.of(mylib))

val expression = Struct.of("f", Struct.of("+", Integer.ONE, Integer.of(2))) // f(1 + 2)
val goal = Struct.of("reduce", Var.of("X"), expression) // reduce(X, f(1 + 2))
println(solver.solveOnce(goal)) // X=f(3)
```

- implementation of `is/2` is analogous, except `ExpressionEvaluator` is used

Overview on Channels I

Why Channels

- Solvers are prosumers of data streams
- Data represented in the form of characters / strings
- I/O facilities similar to Java's Readers and Writers are needed
  - ! unfortunately, Kotlin MP does not support I/O

Overview on Channels II

Design choices

- Channels are of 2 sorts: either InputChannels or OutputChannels
  - InputChannels (aka sources) let Solvers read data of type T
  - OutputChannels (aka sinks) let Solvers write data of type T
- Channels are identified by unique terms, in the eyes of Solvers
  - or by some alias, i.e. an atom
- Channels provide basic support to the features of *:io-lib*
- Each Solver contains an InputStore and an OutputStore
  - keeping track of currently available Channels
  - and their aliases

Overview on Channels III

Default Channels

Each `Solver` is initialised with 4 default `Channels`:

`stdIn:` `InputChannel<String>` main input channel

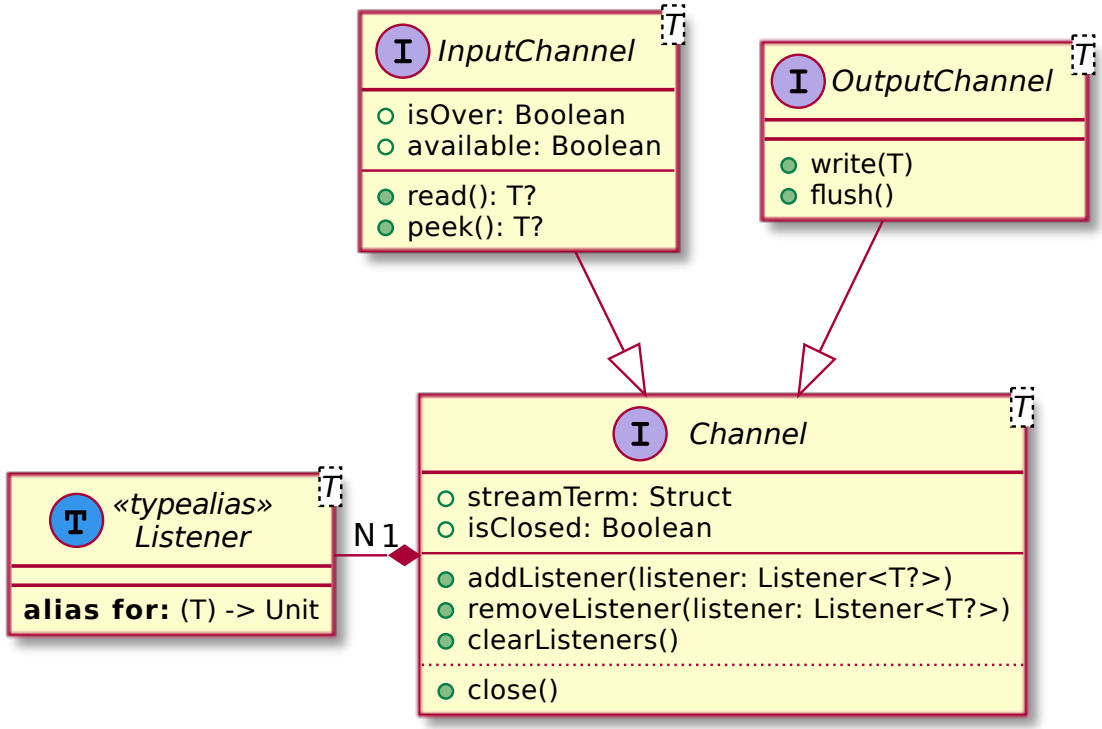
`stdOut:` `OutputChannel<String>` main output channel

`stdErr:` `OutputChannel<String>` error channel

`warnings:` `OutputChannel<PrologWarning>` warning channel

- does not output strings, but instances of `PrologWarning`

The Channel Type Hierarchy I



The Channel Type Hierarchy II

The Channel<T> type

Base type for both input/output channels. All channels are observable entities supporting the (un)registration of listeners.

`addListener(Listener<T>)` registers a new listener

`removeListener(Listener<T>)` unregisters a listener

`clearListeners(Listener<T>)` unregisters all listeners

`close()` closes the channel (no more write/read allowed)

`isClosed: Boolean` returns `true` iff the channel has been closed

`streamTerm: Struct` returns the `Struct` identifying this channel

- is always a structure of the form:

```
'$stream'(InOut, Id)
```

  where
  - `InOut` is either '`in`' or '`out`', depending on the sort of channel
  - `Id` is a number uniquely identifying the channel instance

The Channel Type Hierarchy III

The InputChannel<T> type is a sub-type of Channel<T>

Base type for input channels. Support reading an instance of T at a time, from some data source.

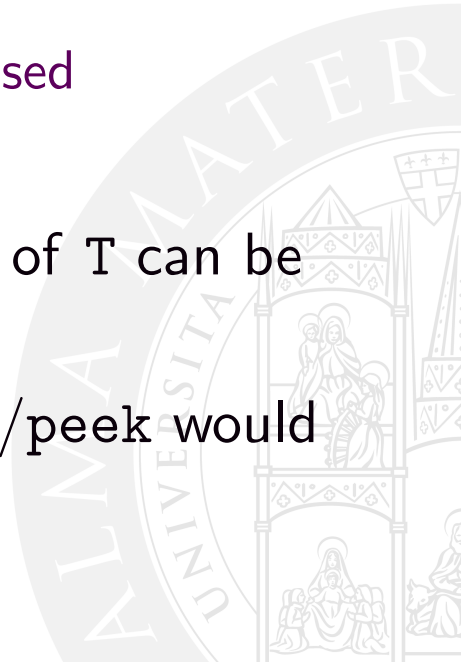
`read(): T?` consumes and returns the next instance of T in the channel

- returns `null` if the channel is over
- throws `IllegalStateException` if the channel is closed

`peek(): T?` like `read`, but does not consume

`available: Boolean` returns `true` iff the next instance of T can be read/peeked without blocking

`isOver: Boolean` returns `tree` iff the next call to `read`/`peek` would return `null`



The Channel Type Hierarchy IV

The OutputChannel<T> type is a sub-type of Channel<T>

Base type for output channels. Support writing an instance of T at a time, onto some data sink.

`write(T)` writes an instance of `T`

- throws `IllegalStateException` if the channel is closed

`flush()` ensures data is actually written to the channel

- (useful in case of buffering / caching)

Creating Channels I

Design choices – InputChannels

InputChannels may be created out of:

- a generator function (of type `() -> T`)
- a String—supporting its consumption character-wise
- an InputStream or Reader (JVM only)
  - eg System.in

`stdIn(): InputChannel<String>` returns a wrapper of the actual standard input of the current program

- eg System.in on the JVM
- ! makes no sense on JS Browser-side

`of(() -> T): InputChannel<T>` creates an input channel out of a generator function

Creating Channels II

`of(String): InputChannel<String>` creates an input channel consuming a string character-wise

```
val source = InputChannel.stdIn()
println(source.read()) // block until a line is is prompted,
    // then prints the first char
    // may print null in testing
```

```
val source = InputChannel.of("hello")
println(source.read()) // h
println(source.read()) // e
println(source.read()) // l
```

```
var i = 0
val source = InputChannel.of { ('a' + i++).toString() }
println(source.read()) // a
println(source.read()) // b
println(source.read()) // c
```

Creating Channels III

Design choices – InputChannels

`OutputChannels` may be created out of:

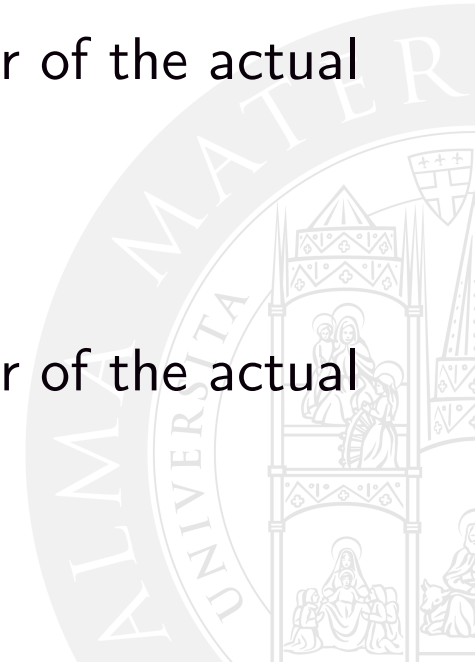
- a consumer function (of type `(T) -> Unit`)
- an `OutputStream` or `Writer` (JVM only)
  - eg `System.out` or `System.err`

`stdOut(): OutputChannel<String>` returns a wrapper of the actual standard output of the program

- eg `System.out` on the JVM
- eg `Console.log` on JS

`stdErr(): OutputChannel<String>` returns a wrapper of the actual standard error of the program

- eg `System.err` on the JVM
- eg `Console.error` on JS



Creating Channels IV

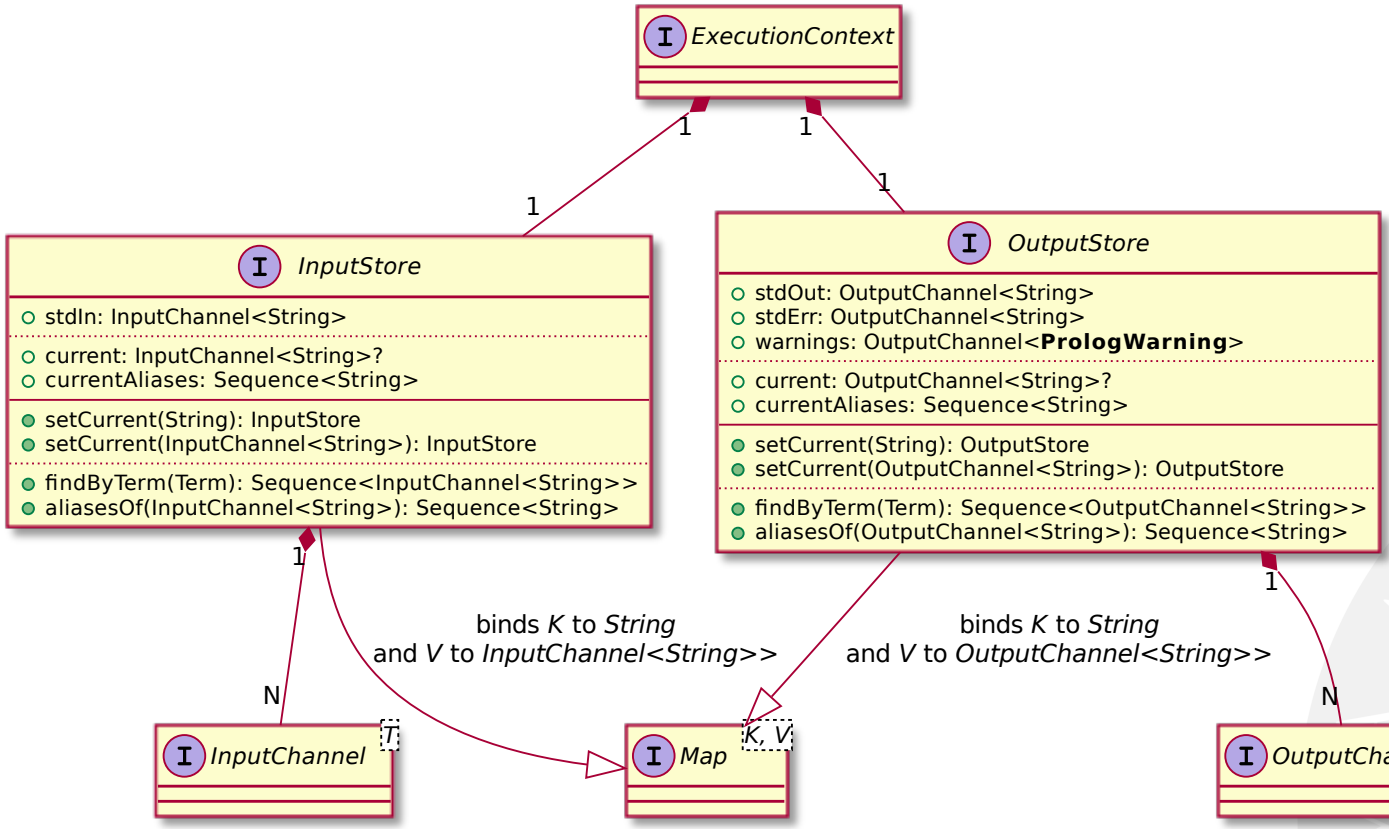
`warn(): OutputChannel<PrologWarning>` returns a channel outputting warnings to the actual standard error of the program

`of((T) -> Unit): OutputChannel<T>` creates an output channel out of a consumer function (i.e. a *callback*)

```
val sink1 = OutputChannel.stdOut<String>()
val sink2 = OutputChannel.stdErr<String>()

sink1.write("Printed on stdout\n")
sink2.write("Printed on stderr\n")
```

```
val messages = mutableListOf<String>()
val sink = OutputChannel.of<String> { messages += it }

sink.write("Appended to messages")

println(messages) // [Appended to messages]
```

The ChannelStore Type Hierarchy I



The ChannelStore Type Hierarchy II

The InputStore type

Immutable map keeping track of currently open InputChannels for a Solver, and their aliases

- References the standard input channel
- Keeps track of the current input channel
  - ie the one the Solver reads from, by default

The ChannelStore Type Hierarchy III

The OutputStore type

Map keeping track of currently open OutputChannels for a Solver, and their aliases

- References the standard input and error channels
- References the warnings channel
- Keeps track of the current output channel
  - ie the one the Solver writes onto, by default

The ChannelStore Type Hierarchy IV

Common methods:

`current: Channel<String>` returns the current channel

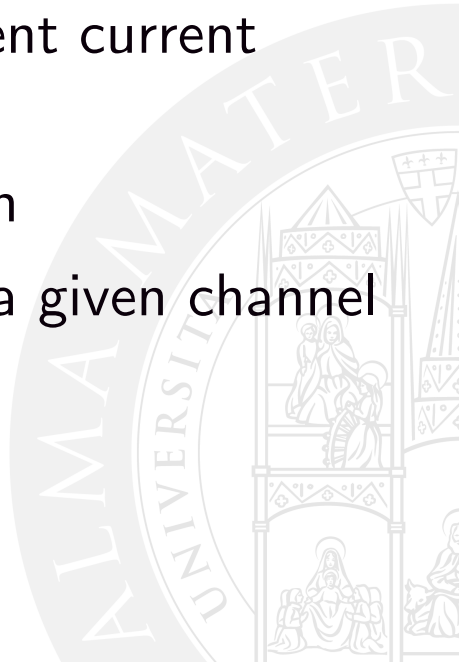
`setCurrent(Channel<String>)` returns a new store having a different current channel

`setCurrent(String)` returns a new store having a different current channel (selected by alias)

`findByTerm(Term)` returns a channel by its identifier term

`aliasesOf(Channel<String>)` returns all the aliases of a given channel

+ all methods of `Map<String, Channel<String>>`



Using Channels

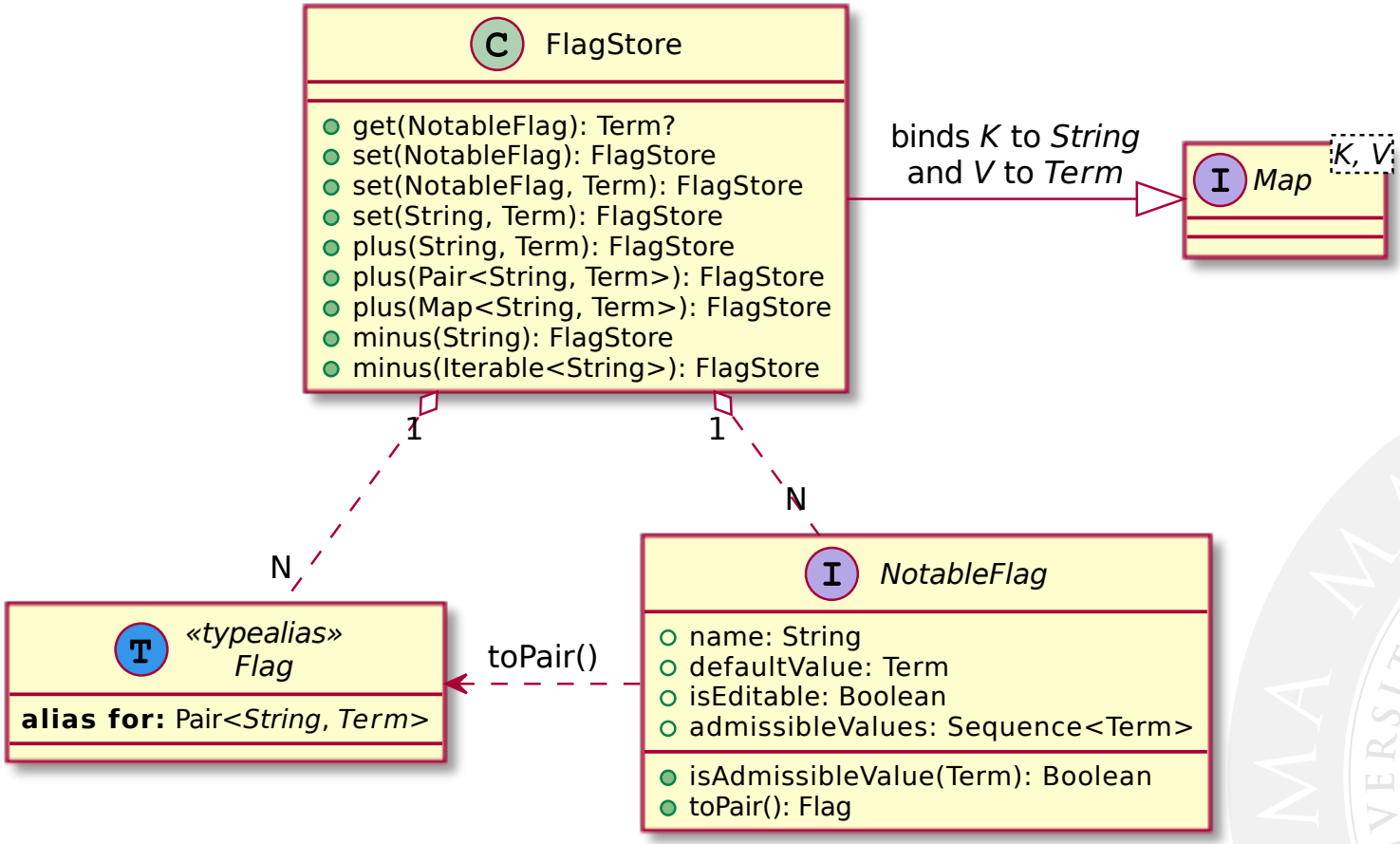
```
val messages = mutableListOf<String>()

val solver = Solver.classic.solverWithDefaultBuiltins(
    otherLibraries = Libraries.of(IOLib),
    stdIn = InputChannel.of("hello"),
    stdOut = OutputChannel.of { messages += it }
)

val goal = Scope.empty {
    tupleOf(
        structOf("get_char", varOf("X")),
        structOf("write", varOf("X"))
    )
} // ?- get_char(X), write(X).

for (i in 0 until "hello".length) {
    println(solver.solveOnce(goal)) // X=h, X=e, X=l, ...
}

println(messages) // [h, e, l, l, o]
```

Overview on flags

About flags

- Flags are key-value pairs carrying configurable aspects of `Solvers`
  - keys are `Strings`, whereas
  - values are `Terms`
- Some flags are editable, while others are `read-only`
  - All flags can be inspected during resolution
  - Editable flags' values can be altered during resolution
- Custom flags may be defined/inspected during resolution

The FlagStore and NotableFlag Types I



The `FlagStore` and `NotableFlag` Types II

No type for flags

- Kotlin's `Pairs` are used instead
- Flag $\equiv$ `Pair<String, Term>`, in the general case
- `NotableFlag` as the convenience type for built-in flags

The FlagStore and NotableFlag Types III

The NotableFlag type

Ad-hoc type representing built-in flags and their admissible values

`name: String` returns the name (i.e. the key) of the flag

`admissibleValues: Sequence<Term>` returns the set of admissible values of the flag

`defaultValue: Term` returns some item of `admissibleValues`

`isEditable: Boolean` returns `true` iff the flag's value can be altered during resolution

`isAdmissibleValue(Term): Boolean` checks whether the provided `Term` is an admissible value for the flag

`toPair(): Pair<String, Term>` converts the notable flag into a key-value pair

The FlagStore and NotableFlag Types IV

The FlagStore type

Immutable map keeping track of currently defined flags' keys and their values

+ some methods for getting/setting NotableFlags

`get(NotableFlag): Term?` returns the *actual* value of some notable flag

`get(String): Term?` returns the *actual* value of some flag

`set(NotableFlag): FlagStore` returns a novel store where the provided notable flag is set to its *default* value

`set(NotableFlag, Term): FlagStore` returns a novel store where the provided notable flag is set to the *provided* value

`set(String, Term): FlagStore` returns a novel store where the provided flag is set to the provided value

The FlagStore and NotableFlag Types V

`plus(`⟨*OtherFlags*⟩`):` `FlagStore` returns a novel store containing the provided flags as well

`minus(`⟨*FlagNames*⟩`):` `FlagStore` returns a novel store not containing the provided flags names

+ methods of `Map`

Example: Default NotableFlags and their Purpose I

unknown

Defines what to do when no rule/primitive is found for solving a (sub-)goal:

- Class: `it.unibo.tuprolog.solve.flags.Unknown`
- Admissible values:

  `fail` the goal is silently considered `false`
  `warning` *(default)* the goal is considered `false` & a warning is raised
  `error` an error is raised and resolution is interrupted

max_arity

Read-only flag dictating the maximum admissible value for a `Struct`

- Class: `it.unibo.tuprolog.solve.flags.MaxArity`
- Admissible values: `Int.MAX_VALUE` (i.e. $2^{31} - 1$)

Example: Default NotableFlags and their Purpose II

last_call_optimization

Dictates whether Prolog Solvers should optimise tail-recursive rules

- Class: `it.unibo.tuprolog.solve.flags.LastCallOptimization`
- Admissible values:
  - `on` *(default)* optimization is performed whenever possible
  - `off` optimization is never performed

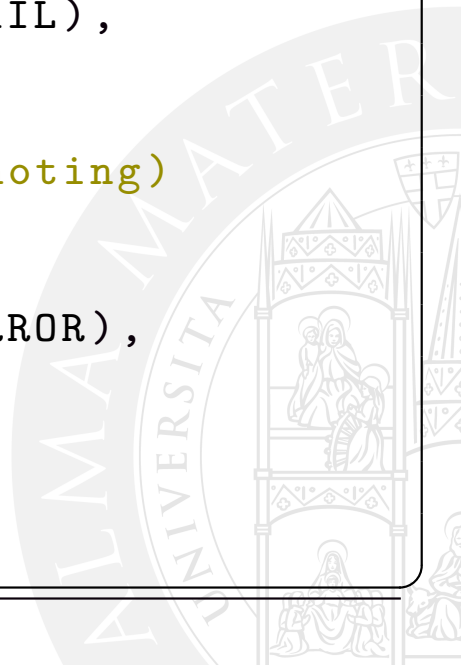
double_quotes

Read-only flag dictating how double-quoted literals should be interpreted

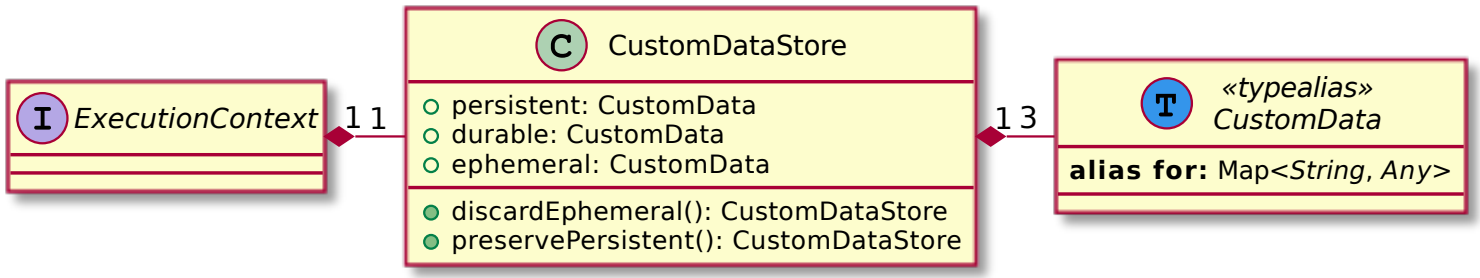
- Class: `it.unibo.tuprolog.solve.flags.DoubleQuotes`
- Admissible values: `atom`
  - ie double-quoted literals are always considered as `Atom`

Using Flags

```
val solver1 = Solver.classic.solverOf(
    flags = FlagStore.DEFAULT,
    warnings = OutputChannel.warn() // default
)
solver1.solveOnce(Atom.of("missing")) // no (prints warning)
    // No such a predicate: missing/0

val solver2 = Solver.classic.solverOf(
    flags = FlagStore.DEFAULT.set(Unknown, Unknown.FAIL),
    warnings = OutputChannel.warn() // default
)
solver2.solveOnce(Atom.of("missing")) // no (prints noting)

val solver3 = Solver.classic.solverOf(
    flags = FlagStore.DEFAULT.set(Unknown, Unknown.ERROR),
    warnings = OutputChannel.warn() // default
)
solver3.solveOnce(Atom.of("missing")) // halt
    // existence_error(procedure, missing/0)
```



Overview on CustomData I



About CustomDataStores

- To ease extensibility, `ExecutionContexts` main contain `CustomData` too
- This is achieved via one more property, of type `CustomDataStore`
- `CustomDataStores` have 3 fields of type `CustomData`
- `CustomData` are key-value store which may keep any sort of data
- This may exploited by `Primitives`/`Solvers` implementers
  - to accumulate/inspect data to affect resolution

Overview on CustomData II

Sorts of CustomData

Custom data are of 3 sorts:

ephemeral — automatically erased after a choice point is selected

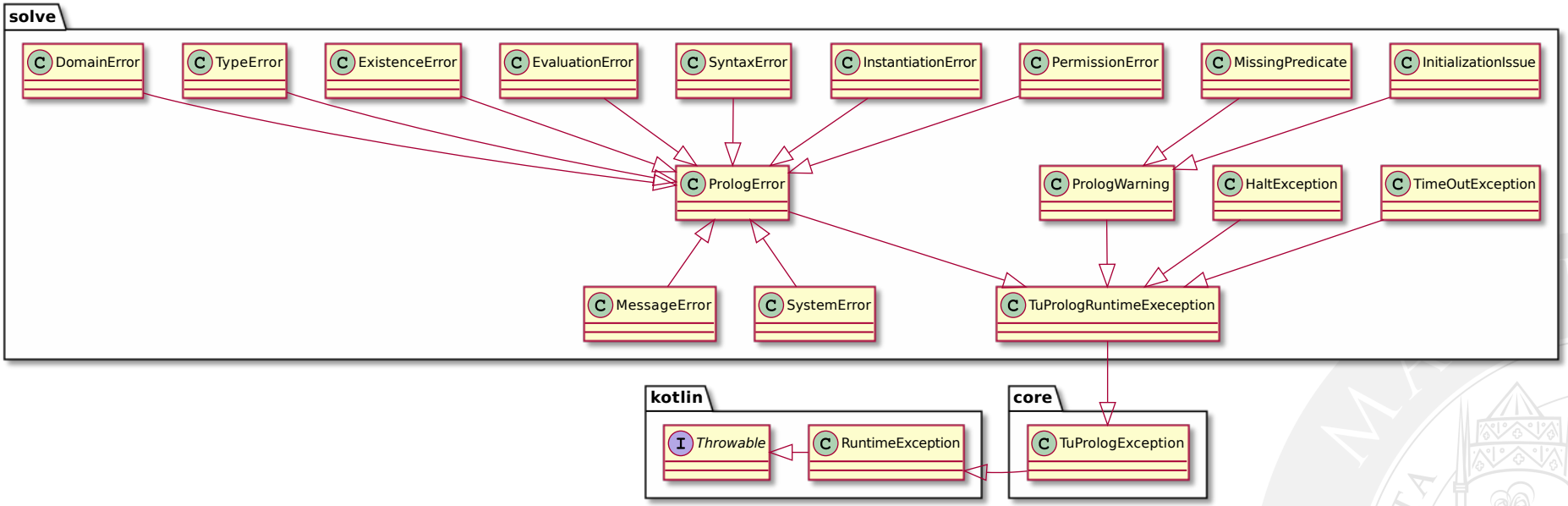
eg in Prolog, once every time backtracking is performed

durable — automatically erased after a solution is reached

persistent — never erased automatically

Focus on...

The TuPrologRuntimeException Type Hierarchy I



The TuPrologRuntimeException Type Hierarchy II

`TuPrologRuntimeException` base type for all issues which may occur during resolution

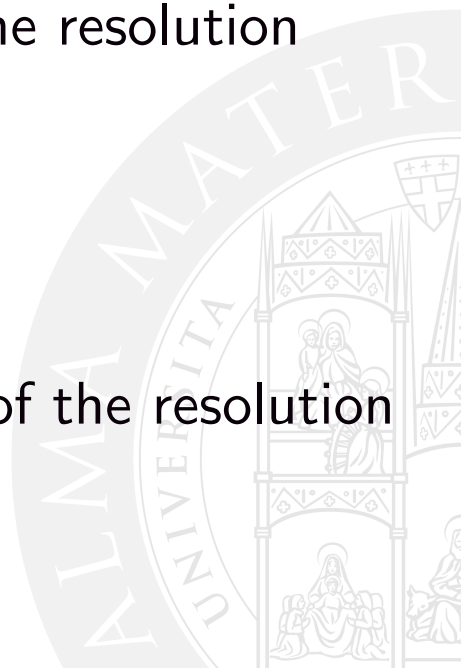
- when thrown within `Primitives`, they affect the ongoing resolution
- resolution is interrupted, and a `Solution.Halt` is produced
  - the solution keeps track of the exception
  - the exception keeps track of the execution context where the issue occurred

`HaltException` provokes the immediate interruption of the resolution process

- may carry an exit code (integer)
- analogous to `System.exit(`$\langle Int \rangle$`)` one the JVM
- produces a `Solution.Halt`

`TimeOutException` provokes the immediate interruption of the resolution process

- automatically thrown by `Solvers` upon timeouts



The TuPrologRuntimeException Type Hierarchy III

`PrologError` base types for LP-related errors

ie errors which can be caught in LP

eg via `catch/3`, in Prolog

`PrologWarning` base types for LP-related warnings

- these are not meant to be thrown
- instances of `PrologWarning` must be considered ordinary data structures to be outputted on the `warnings OutputChannel`

The PrologError Type Hierarchy I

About PrologErrors

- Each instance of `PrologError` corresponds to a `Term`
- The term can be thrown with predicate `throw/1`...
- ...and caught via `catch/3`

The PrologError Type Hierarchy II

Prolog ISO Standard Errors

Following the Standard:

- each subtype of `PrologError` has a corresponding error `Struct`
- the general pattern is as follows:

  `error(`*`Description`*`,` *`Custom`*`)`

  where:

  `Description` is a `Term` whose structure depends on the particular sub-type of `PrologError`
  - it aims at describing the error, logically

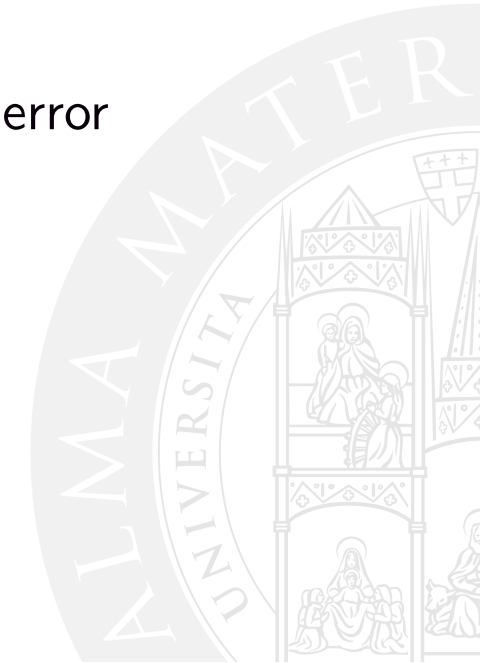
  `Custom` is a custom, implementation-specific `Term`
  - 2P-Kt's convention: it carries the message to be shown to the user
  - so, it is commonly an `Atom`

The `PrologError` Type Hierarchy III

`InstantiationError` to be thrown when a sub-goal/argument is being evaluated but it is not sufficiently instantiated

`Description` is always the atom `instantiation_error`

`Custom` describes what argument/sub-term provoked the error



The `PrologError` Type Hierarchy IV

`TypeError` to be thrown when a sub-goal/argument is being evaluated but has not the right logic type

`Description` is a `Struct` of the form `type_error(`*`Expected`*`, `*`Culprit`*`)`

`Expected` is an `Atom` describing the expected type

- admissible values are contained into the class `TypeError.Expected`

eg `atom`, `atomic`, `callable`, `character`, `compound`, etc.

`Culprit` is the `Term` having the wrong type

The PrologError Type Hierarchy V

`DomainError` to be thrown when a sub-goal/argument is being evaluated and it has the right type, but its value is not admissible

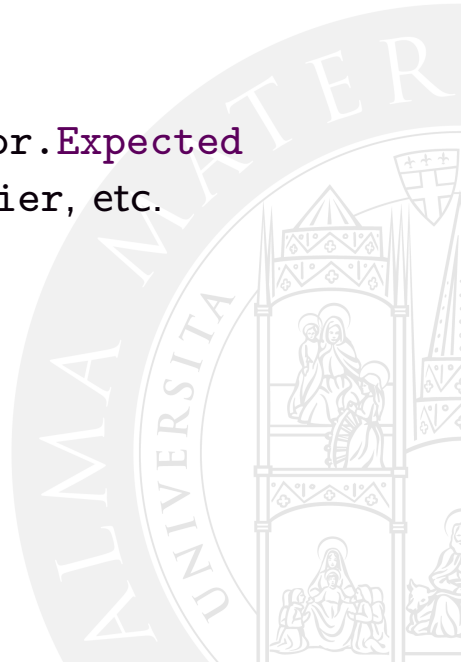
Description is a `Struct` of the form `domain_error(`*Expected*`,` *Culprit*`)`

Expected is an `Atom` describing the expected value

- admissible values are contained into the class `DomainError.Expected`

eg `not_less_than_zero`, `non_empty_list`, `operator_specifier`, etc.

Culprit is the `Term` having the wrong value



The PrologError Type Hierarchy VI

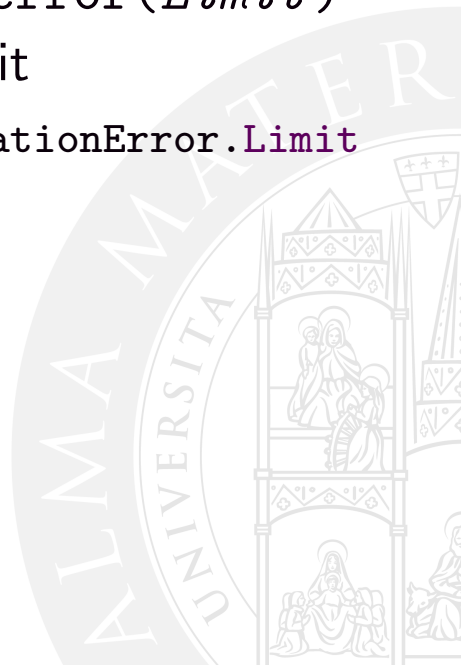
`RepresentationError` to be thrown when a value cannot be represented as a `Term`

Description is a `Struct` of the form `representation_error(`*Limit*`)`

Limit describes the representation limit which has been hit

- admissible values are contained into the class `RepresentationError.Limit`

eg `character`, `character_code`, `max_arity`, etc.



The `PrologError` Type Hierarchy VII

`SyntaxError` to be thrown when parsing some badly formed string during resolution

`Description` is always the atom `syntax_error`

`Custom` describes why the string could not be parsed, and localizes the issue into the string

The `PrologError` Type Hierarchy VIII

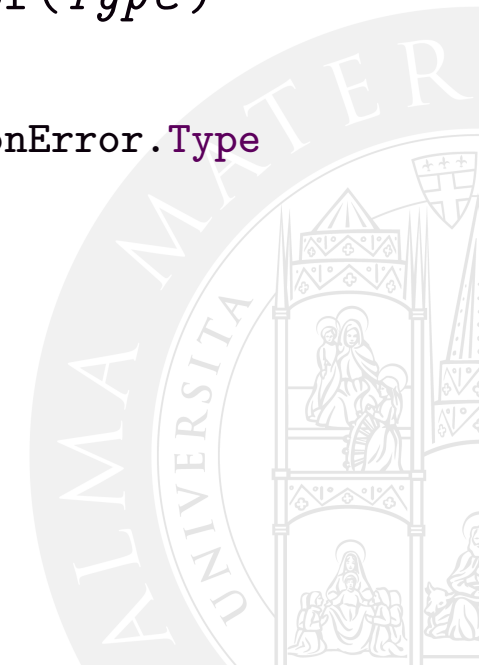
`EvaluationError` to be thrown when some expression evaluation meets an arithmetic issue

`Description` is a `Struct` of the form `evaluation_error(`*`Type`*`)`

`Type` describes the evaluation issue which has been met

- admissible values are contained into the class `EvaluationError.Type`

eg `zero_divisor`, `int_overflow`, `float_overflow`, etc.



The `PrologError` Type Hierarchy IX

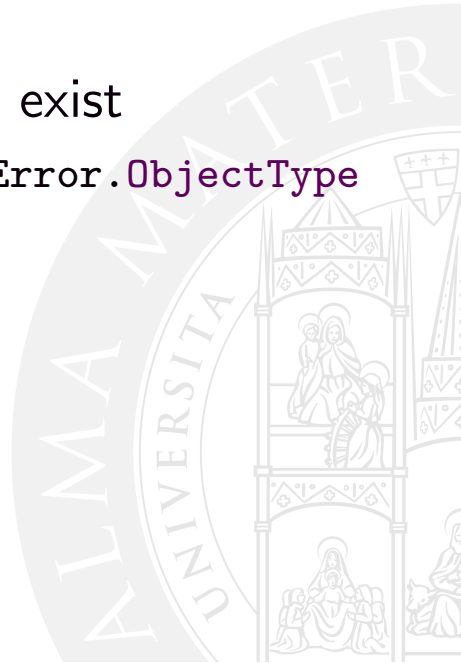
`ExistenceError` to be thrown accessing some resource by name/reference fails because the resource does not exist

`Description` is a `Struct` of the form `existence_error(`*`ObjectType`*`,` *`Culprit`*`)`

`ObjectType` describes the sort of resource which does not exist

- admissible values are contained into the class `ExistenceError.ObjectType`

eg `procedure`, `source_sink`, `stream`, etc.



The PrologError Type Hierarchy X

`PermissionError` to be thrown when performing an operation which is not permitted

Description is a `Struct` of the form `permission_error(`*Operation*`,` *Permission*`,` *Culprit*`)`

Operation describes the sort of operation which failed

- admissible values are contained into the class `PermissionError.Operation`

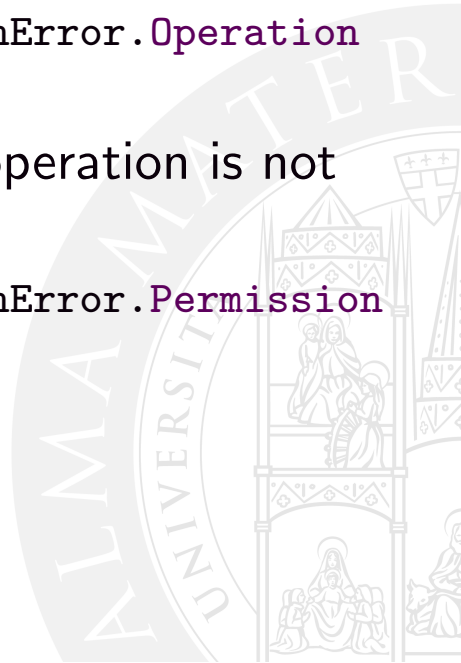
eg `access`, `invoke`, `modify`, `open`, etc.

Permission describes the sort of resource for which the operation is not permitted

- admissible values are contained into the class `PermissionError.Permission`

eg `flag`, `operator`, `source_sink`, `static_procedure`, etc.

Culprit is the (sub-)goal which provoked the error



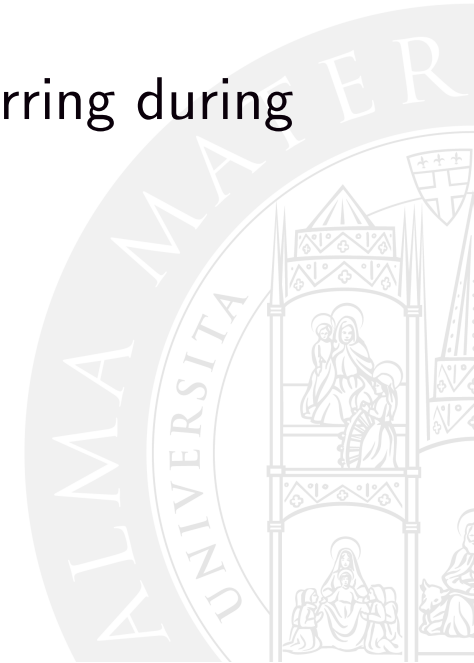
The `PrologError` Type Hierarchy XI

`SystemError` to be thrown for any other circumstance

`Description` is always the atom `system_error`

`Custom` provides the actual description of the problem

! this may be used to wrap Kotlin exceptions occurring during resolution



The `PrologWarning` Type Hierarchy

Main sub-types:

`MissingPredicate` used to signal that a `Solver` knows no predicate to solve a (sub-)goal

`InitializationIssue` used to signal a failure/error while executing some directive

Focus on...

Overview on Custom Libraries I

- Libraries are the basic means to provide built-in functionalities to solvers
- Default built-in functionalities are provided via libraries as well
- Creating a library essentially means:
  1. defining a number of primitives
  2. defining a number of functions
  3. defining a number of rules and facts
  4. deciding which of these should be usable as operators
  5. defining operators accordingly

! Primitives, in particular, are the most common object libraries implementers will manipulate

Overview on Custom Libraries II

Writing **primitives** involves many common aspects

- eg input validation
- eg permission checking
- eg arguments/results packing & unpacking
- ⋮

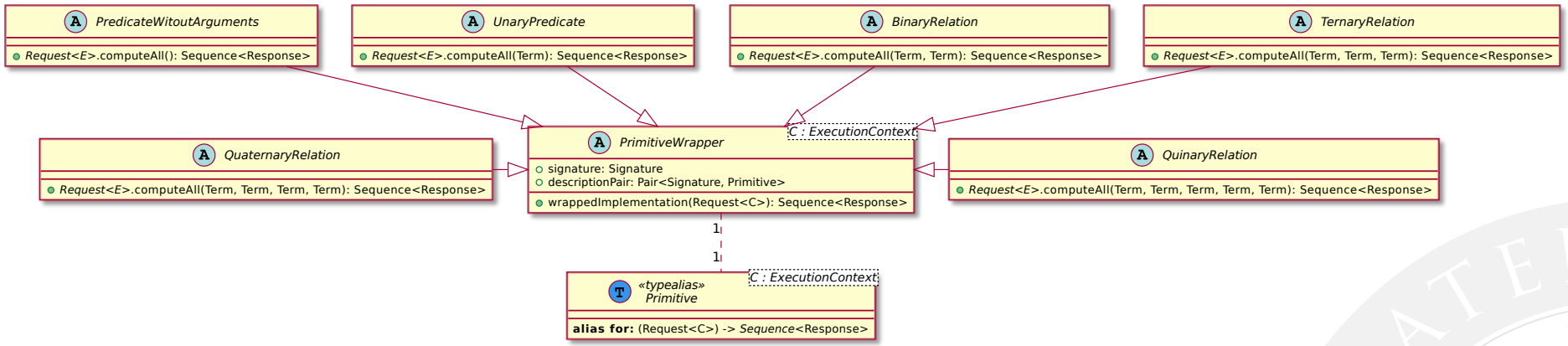
Overview on PrimitiveWrappers I

The PrimitiveWrapper type hierarchy

A pool of abstract classes factorising common aspects of primitives' implementation

- primitive wrapper $\approx$ signature + primitive + utility methods

Overview on PrimitiveWrappers II



Overview on PrimitiveWrappers III

Main wrapper classes:

`PredicateWithoutArguments` base class for 0-ary predicates

`UnaryPredicate` base class for 1-ary predicates

`BinaryRelation` base class for 2-ary predicates

`TernaryRelation` base class for 3-ary predicates

`QuaternaryRelation` base class for 4-ary predicates

`QuinaryRelation` base class for 5-ary predicates

Overview on `PrimitiveWrappers` IV

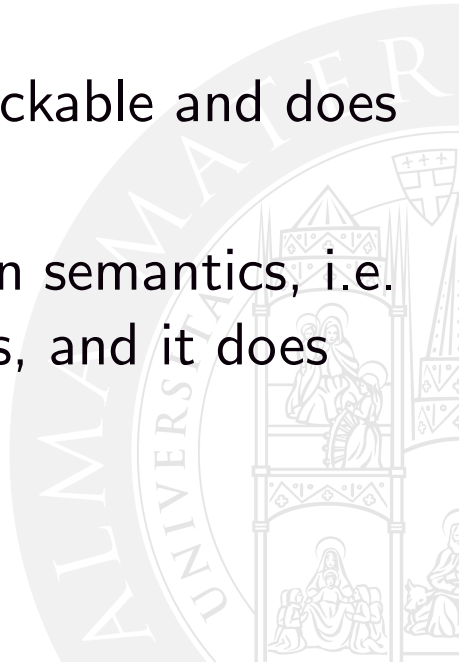
Each wrapper class, comes with 4 more abstract sub-classes:

`WithoutSideEffects` to be preferred if the predicate does not provoke side-effects

`NonBacktrackable` to be preferred if the predicate is not backtrackable, yet it may provoke side-effects

`Functional` to be preferred if the predicate is not backtrackable and does not provoke side-effects

`Predicative` to be preferred if the predicate has a boolean semantics, i.e. it is not backtrackable, it does not provoke side-effects, and it does not assign any variable



PrimitiveWrappers' Utility Methods I

Each sub-class of `PrimitiveWrapper` inherits a number of protected utility methods:

`ensuringAllArgumentsAreInstantiated()` throws an `InstantiationError` if some argument is a `Var`

`ensuringArgumentIsWellFormedIndicator(Int)` checks whether the argument at the provided index is a well formed indicator (i.e. $\langle Atom \rangle / \langle Integer \rangle$), throwing the most adequate error in case it is not

`ensuringArgumentIsWellFormedClause(Int)` checks whether the argument at the provided index is a well formed clause (i.e. no sub-goal in the body is a `Var`), throwing the most adequate error in case it is not

`ensuringArgumentIsInstantiated(Int)` throws an `InstantiationError` if the argument at the provided index is a `Var`

`ensuringArgumentIsNumeric(Int)` throws a `TypeError` if the argument at the provided index is not `Numeric`

PrimitiveWrappers' Utility Methods II

`ensuringArgumentIsStruct(Int)` throws a `TypeError` if the argument at the provided index is not a `Struct`

`ensuringArgumentIsCallable(Int)` alias for `ensuringArgumentIsStruct`

`ensuringArgumentIsVariable(Int)` throws a `TypeError` if the argument at the provided index is not a `Var`

`ensuringArgumentIsCompound(Int)` throws a `TypeError` if the argument at the provided index is not a `Struct` s.t. `arity` > 0

`ensuringArgumentIsAtom(Int)` throws a `TypeError` if the argument at the provided index is not an `Atom`

`ensuringArgumentIsConstant(Int)` throws a `TypeError` if the argument at the provided index is not a `Constant`

`ensuringArgumentIsGround(Int)` throws an `InstantiationError` if the argument at the provided index is not *ground*

PrimitiveWrappers' Utility Methods III

`ensuringArgumentIsChar(Int)` throws a `TypeError` if the argument at the provided index is not an `Atom` s.t. `value.length` $==$ 1

`ensuringArgumentIsSpecifier(Int)` throws a `DomainError` if the argument at the provided index is not an `Atom` from the set {`fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, `xfy`}

`ensuringArgumentIsList(Int)` throws a `TypeError` if the argument at the provided index is not an `List`

`ensuringArgumentIsInteger(Int)` throws a `TypeError` if the argument at the provided index is not an `Integer`

`ensuringArgumentIsNonNegativeInteger(Int)` throws a `TypeError` if the argument at the provided index is not an `Integer`, or a `DomainError` if it is a negative integer

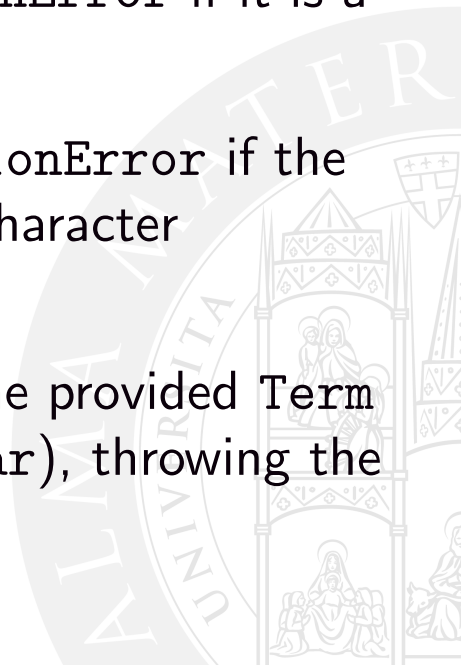
PrimitiveWrappers' Utility Methods IV

`ensuringArgumentIsArity(Int)` throws a `TypeError` if the argument at the provided index is not an `Integer`, a `DomainError` if it is a negative integer, or a `RepresentationError` is it is an integer greater than `MaxArity.defaultValue`

`ensuringArgumentIsWellFormedList(Int)` throws a `TypeError` if the argument at the provided index is not an `List`, or a `DomainError` if it is a `List` whose termination item is not `[]`

`ensuringArgumentIsCharCode(Int)` throws a `RepresentationError` if the argument at the provided index does not correspond to a character according to the UTF-16 encoding

`checkTermIsRecursivelyCallable(Term)` checks whether the provided `Term` is a well formed clause (i.e. no sub-goal in the body is a `Var`), throwing the most adequate error in case it is not



PrimitiveWrappers' Utility Methods V

`ensuringTermIsCharCode(Term)` throws a `RepresentationError` if the provided `Term` does not correspond to a character according to the UTF-16 encoding

`ensuringTermIsWellFormedList(Term)` throws a `TypeError` if the provided `Term` is not an `List`, or a `DomainError` if it is a `List` whose termination item is not `[]`

`ensuringProcedureHasPermission(Signature, PermissionError.Operation)` throws a `PermissionError` is the provided `Signature` matches some entry among the `Solver`'s libraries, or static KB

`ensuringClauseProcedureHasPermission(Clause, PermissionError.Operat` same as above, except that the `Signature` of the `Clause`'s `head` is considerer

Example of `PrimitiveWrappers` of all sorts I

Example 1: `atom_length(+Atom, -Length)`

- Purpose: computes the length of an atom
- Sort of primitive: binary relation, non-backtrackable, no side effect
- May throw:

  `InstantiationError` if the 1^{st} argument is a `Var`
  `TypeError` if the 1^{st} argument is not an `Atom`, or the 2^{nd} is not an `Integer`
  `DomainError` if the 2^{nd} argument is a *negative* `Integer`
- Usage example, in Prolog:

  ?- `atom_length(abc, X)` → `X=3`
  ?- `atom_length(abc, 3)` → `true`
  ?- `atom_length(abc, 4)` → `false`
  ?- `atom_length(X, 4)` → `instantiation_error`
  ?- `atom_length(a, b)` → `type_error`
  ?- `atom_length(a, -1)` → `domain_error`

Example of `PrimitiveWrappers` of all sorts II

```
object AtomLength : BinaryRelation.Functional<ExecutionContext>("atom_length") {
    override fun Solve.Request<ExecutionContext>.computeOneSubstitution(
        x: Term,
        y: Term
    ): Substitution = when {
        x is Var -> {
            ensuringAllArgumentsAreInstantiated()
            Substitution.failed()
        }
        y is Var -> {
            ensuringArgumentIsAtom(0)
            val atomLength = (x as Atom).value.length
            Substitution.of(y, Integer.of(atomLength))
        }
        else -> {
            ensuringArgumentIsAtom(0)
            ensuringArgumentIsNonNegativeInteger(1)
            val atomLength = Integer.of((x as Atom).value.length)
            atomLength mguWith y
        }
    }
}
```

Example of `PrimitiveWrappers` of all sorts III

Example 2: `assert(+Clause)`

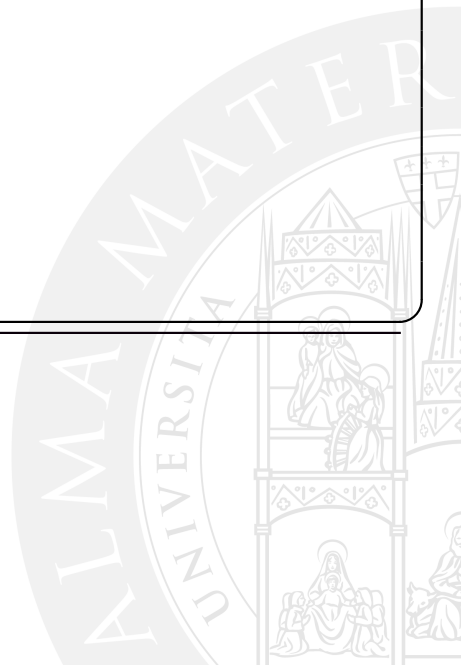
- Purpose: adds a clause to the dynamic KB
- Sort of primitive: unary predicate, non-backtrackable, side effects
- May throw:

  `InstantiationError` if the 1^{st} argument is a `Var`
  `TypeError` if the 1^{st} argument is not a `Struct`
  `DomainError` if the 1^{st} argument is a bad-formed `Clause`
  `PermissionError` if the signature of *`Clause`* clashes with some other procedure from the `Solver`'s library or static KB
- Usage example, in Prolog:

  `?- assert(fact)` → `true`
  `?- assert(goal :- subgoal)` → `true`
  `?- assert(goal :- (subgoal, G))` → `domain_error`
  `?- assert(1)` → `type_error`
  `?- assert(X)` → `instantiation_error`

Example of `PrimitiveWrappers` of all sorts IV

```
object Assert : UnaryPredicate.NonBacktrackable<ExecutionContext>("assert") {
    override fun Solve.Request<ExecutionContext>.computeOne(x: Term): Solve.Response {
        ensuringArgumentIsWellFormedClause(0)
        val clause: Clause = when (x) {
            is Clause -> x
            is Struct -> Fact.of(x)
            else -> return ensuringArgumentIsCallable(0).replyFail()
        }
        ensuringClauseProcedureHasPermission(clause, MODIFY)
        return replySuccess {
            addDynamicClauses(clause)
        }
    }
}
```



Example of `PrimitiveWrappers` of all sorts V

Example 3: `retract(?Pattern)`

- Purpose: removes each clause matching *Pattern* in the dynamic KB
  - removes clauses one by one, via backtracking
- Sort of primitive: unary predicate, backtrackable, side effects
- May throw:

  `InstantiationError` if the 1^{st} argument is a `Var`
  `TypeError` if the 1^{st} argument is not a `Struct`
  `DomainError` if the 1^{st} argument is a bad-formed `Clause`
  `PermissionError` if the signature of *Clause* matches some other procedure from the `Solver`'s library or static KB
- Usage example, in Prolog (assume KB contains `f(1).  f(2).`)

  `?- retract(f(X))` → `X=1, X=2`
  `?- retract(f(1))` → `true`
  `?- retract(f(3))` → `false`

Example of `PrimitiveWrappers` of all sorts VI

```
object Retract : UnaryPredicate<ExecutionContext>("retract") {
    override fun Solve.Request<ExecutionContext>.computeAll(
        x: Term
    ): Sequence<Solve.Response> {
        ensuringArgumentIsWellFormedClause(0)
        val clause: Clause = when (x) {
            is Clause -> x
            is Struct -> Rule.of(x, Var.anonymous())
            else -> return sequenceOf(ensuringArgumentIsCallable(0).replyFail())
        }
        ensuringClauseProcedureHasPermission(clause, PermissionError.Operation.MODIFY)
        return context.dynamicKb[clause].buffered().map {
            val substitution = when (x) {
                is Clause -> (x mguWith it) as Substitution.Unifier
                else -> (x mguWith it.head!!) as Substitution.Unifier
            }
            replySuccess(substitution) {
                removeDynamicClauses(it)
            }
        }
    }
}
```

Example of `PrimitiveWrappers` of all sorts VII

Example 4: `findall(+Template, :Goal, -Solutions)` (pt. 1)

- Assumptions: *Template* is a pattern reusing variables from *Goal*
- Purpose: computes all solutions to *Goal*, each solution is mapped into *Template*, all templates are aggregates into a list and the list is bound to *Solutions*
- Sort of primitive: ternary relation, non-backtrackable, no side-effects
- ! Peculiarities: involves an inner resolution process

Example of PrimitiveWrappers of all sorts VIII

Example 4: findall(*+Template*, *:Goal*, *-Solutions*) (pt. 2)

- May throw:

  InstantiationError if the 2^{nd} argument is a Var
  TypeError if the 2^{nd} argument is not a Struct
  any other error possibly thrown by *Goal*

- Usage example, in Prolog:
  - ?- findall(X+1, member(X, [a,b]), L) → L=[a+1, b+1]
  - ?- findall(_, false, []) → true
  - ?- findall(Z, (Z is X + 1), L) → instantiation_error
  - ?- findall(_, 4, _) → type_error
  - ?- findall(_, G, _) → instantiation_error

Example of `PrimitiveWrappers` of all sorts IX

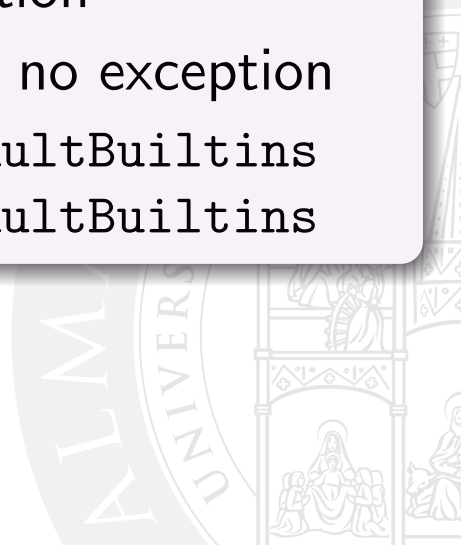
```
object FindAll : TernaryRelation.NonBacktrackable<ExecutionContext>("findall") {
    override fun Solve.Request<ExecutionContext>.computeOne(
        x: Term, y: Term, z: Term
    ): Solve.Response {
        ensuringArgumentIsInstantiated(1)
        ensuringArgumentIsCallable(1)
        val solver = subSolver()
        val solutions = solver.solve(y as Struct).toList()
        val error = solutions.asSequence()
            .filterIsInstance<Solution.Halt>()
            .firstOrNull()
        if (error != null) {
            return replyException(error.exception.pushContext(context))
        }
        val mapped = solutions.asSequence()
            .filterIsInstance<Solution.Yes>()
            .map { x[it.substitution].freshCopy() }
        return replyWith(z mguWith List.of(mapped))
    }
}
```

About DefaultBuiltins and CommonBuiltins I

Convention

Each implementer of `Solver` should provide a library object named `DefaultBuiltins`

- containing all default built-in functionalities for that sort of `Solver`
- to be used by the specific `SolveFactory` implementation
- modules *:solve-classic* and *:solve-streams* are no exception
  - eg `it.unibo.tuprolog.solve.classic.stdlib.DefaultBuiltins`
  - eg `it.unibo.tuprolog.solve.streams.stdlib.DefaultBuiltins`



About DefaultBuiltins and CommonBuiltins II

The CommonBuiltins library

- Most functionalities are agnostic w.r.t. how Solvers are implemented

→ Module *:solve* provides an implementation for most of them

  - in it.unibo.tuprolog.solve.stdlib.CommonBuiltins

- These functionalities are taken from the Prolog ISO Standard [1]
  - yet, they are resolution-agnostic
- Implementers may choose to reuse some or all functionalities from CommonBuiltins

About DefaultBuiltins and CommonBuiltins III

Module solve's CommonBuiltins content:

operators: all Prolog standard operators

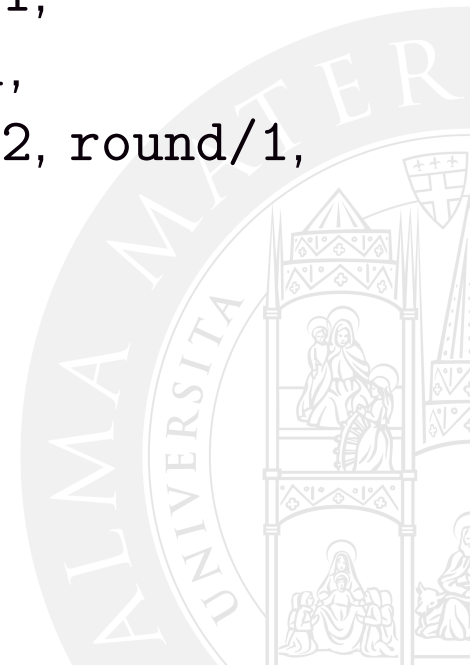
primitives: abolish/1, arg/3, ':='/2, '>'/2, '>='/2, '<'/2, '=<'/2, '=\='/2, assert/1, asserta/1, assertz/1, atom/1, atom_chars/2, atom_codes/2, atom_concat/3, atomic/1, atom_length/2, between/3, bagof/3, callable/1, char_code/2, clause/2, compound/1, copy_term/2, current_op/2, current_prolog_flag/2, ensure_executable/1, findall/3, float/1, functor/3, get_durable/2, get_ephemeral/2, get_persistent/2, ground/1, halt/0, halt/1, integer/1, is/2, natural/1, nl/0, nonvar/1, '\='/2, number/1, number_chars/2, number_codes/2, op/3, repeat/0, retract/1, retractall/1, reverse/2, set_durable/2, set_ephemeral/2, setof/2, set_persistent/2, set_prolog_plag/2, sleep/1, sub_atom/5,

About DefaultBuiltins and CommonBuiltins IV

'@>'/2, '@>='/2, '=='/2, '@<'/2, '@=<'/2, '\=='/2, '\=@='/2, '=@='/2, '='/2, '=..'/2, var/1, write/1

functions: abs/1, '+'/2, atan/1, '/\'/2, '\'/1, '<<'/2, '\/'/2, '>>'/2, ceiling/1, cos/1, exp/1, '**'/2, float/1, float_fractional_part/1, float_integer_part/1, '/'/2floor/1, '//'/2, mod/2, '*'/2, log/1, rem/2, round/1, sign/1, '-'/1, sin/1, sqrt/1, '-'/2, truncate/1

theory: not/1, '->'/2, ';'/2, member/2, append/3



About DefaultBuiltins and CommonBuiltins V

Module `solve-classic`'s `DefaultBuiltins` content:

operators: all operators in `CommonBuiltins.operators`

functions: all functions in `CommonBuiltins.functions`

theory: all clauses in `CommonBuiltins.theory`

+ `call/1`, `catch/3`, `','/2`, `'!'/0`, `'\+'/1`

primitives: all primitives in `CommonBuiltins.primitives`

+ `throw/1`

About DefaultBuiltins and CommonBuiltins VI

Module `solve-classic`'s `DefaultBuiltins` content:

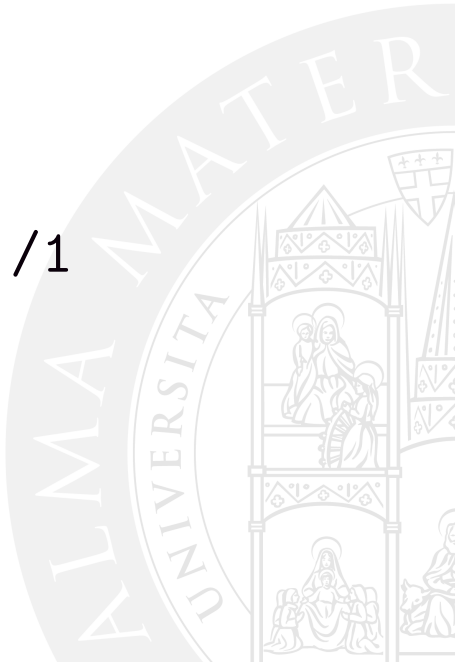
operators: all operators in `CommonBuiltins.operators`

functions: all functions in `CommonBuiltins.functions`

theory: all clauses in `CommonBuiltins.theory`

primitives: all primitives in `CommonBuiltins.primitives`

+ `call/1`, `catch/3`, `','/2`, `'!'/0`, `throw/1`, `'\+'/1`



Next in Line...

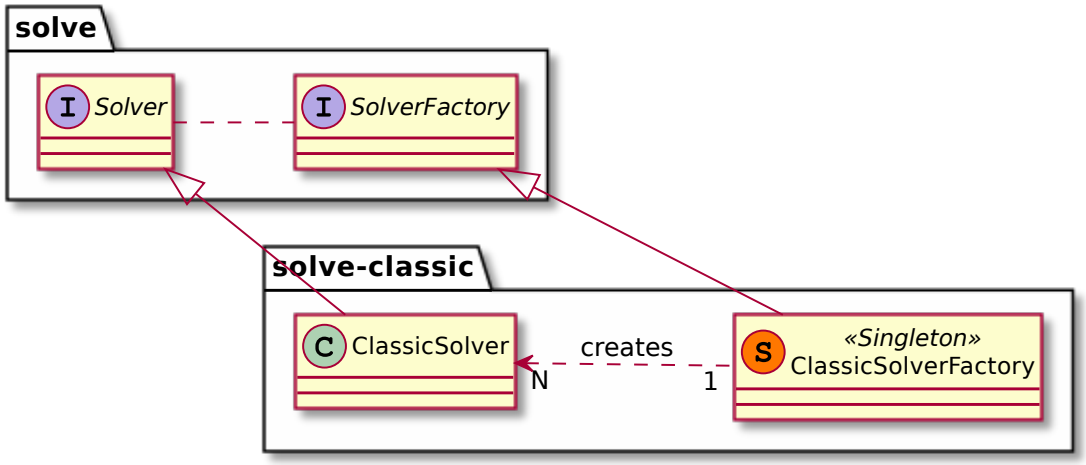
Overview on the *:solve-classic* Module I

Module *:solve-classic* in a nutshell

- Main implementation of `Solver`
- Provides Prolog ISO Standard resolution
- State-Machine-based design inspired by [15]

Overview on the *:solve-classic* Module II

Main abstractions from this module (client-code):

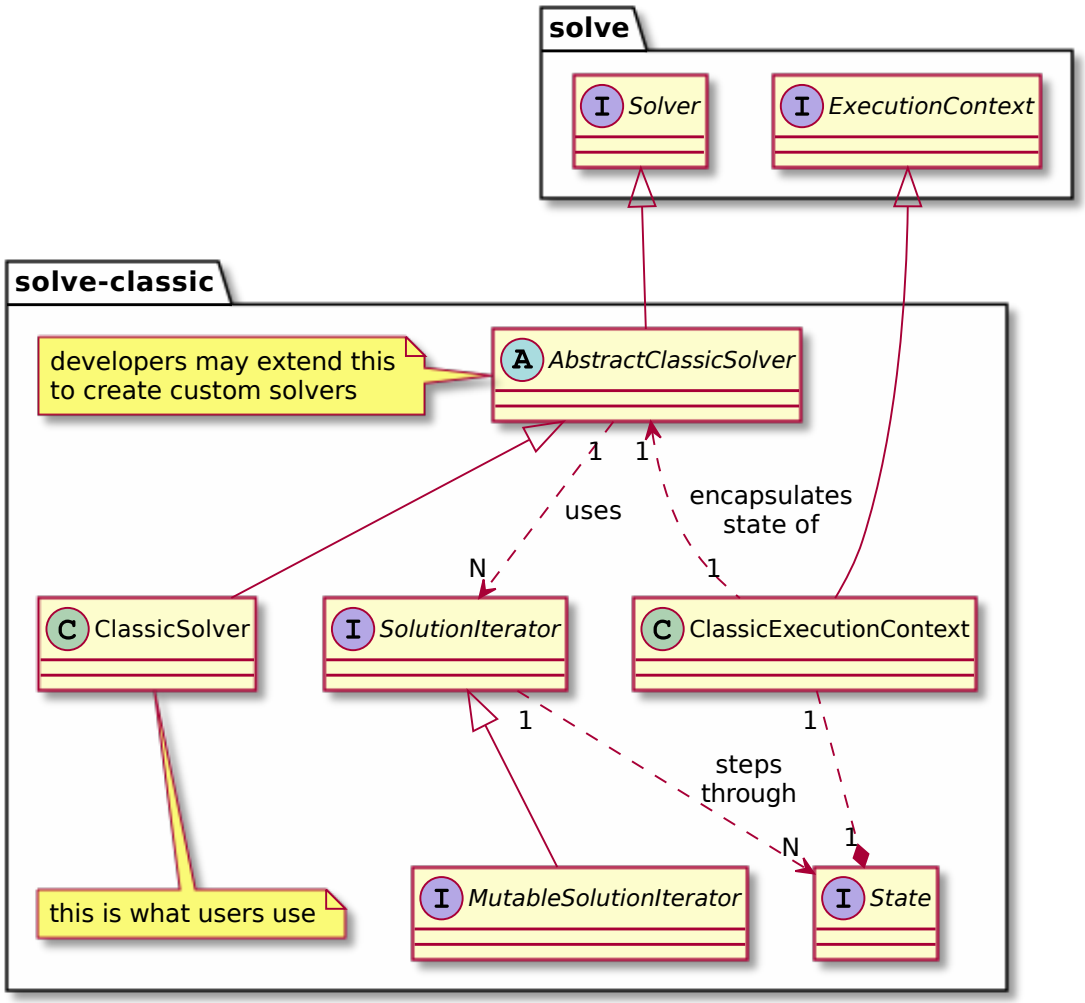


The `ClassicSolver`

- State-Machine-based implementation of `Solver`
  - whose instances are created via the `ClassicSolverFactory`
    - also accessible via `Solver.classic`

Overview on the *:solve-classic* Module III

Main abstractions from this module (details):



Overview on the *:solve-classic* Module IV

`State` base type for states of the Prolog State Machine (PSM)

`ClassicExecutionContext` data structure containing the `ExecutionContext` of (each state of) the PSM

`SolutionIterator` where the PSM is executed

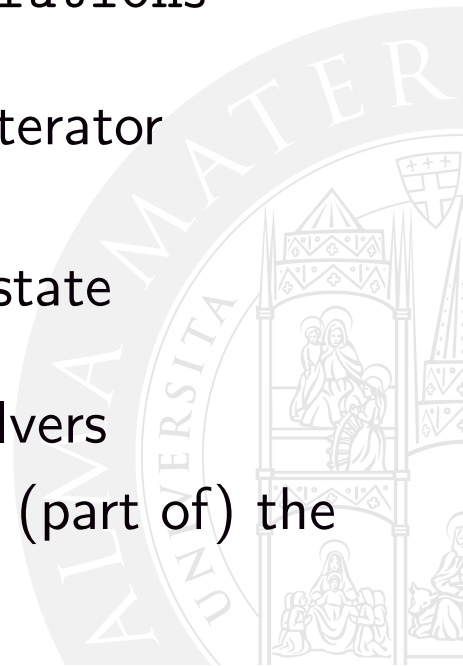
ie an object iterating over states and outputting `Solutions`

`MutableSolutionIterator` a particular sort of solution iterator supporting hijack of state transitions

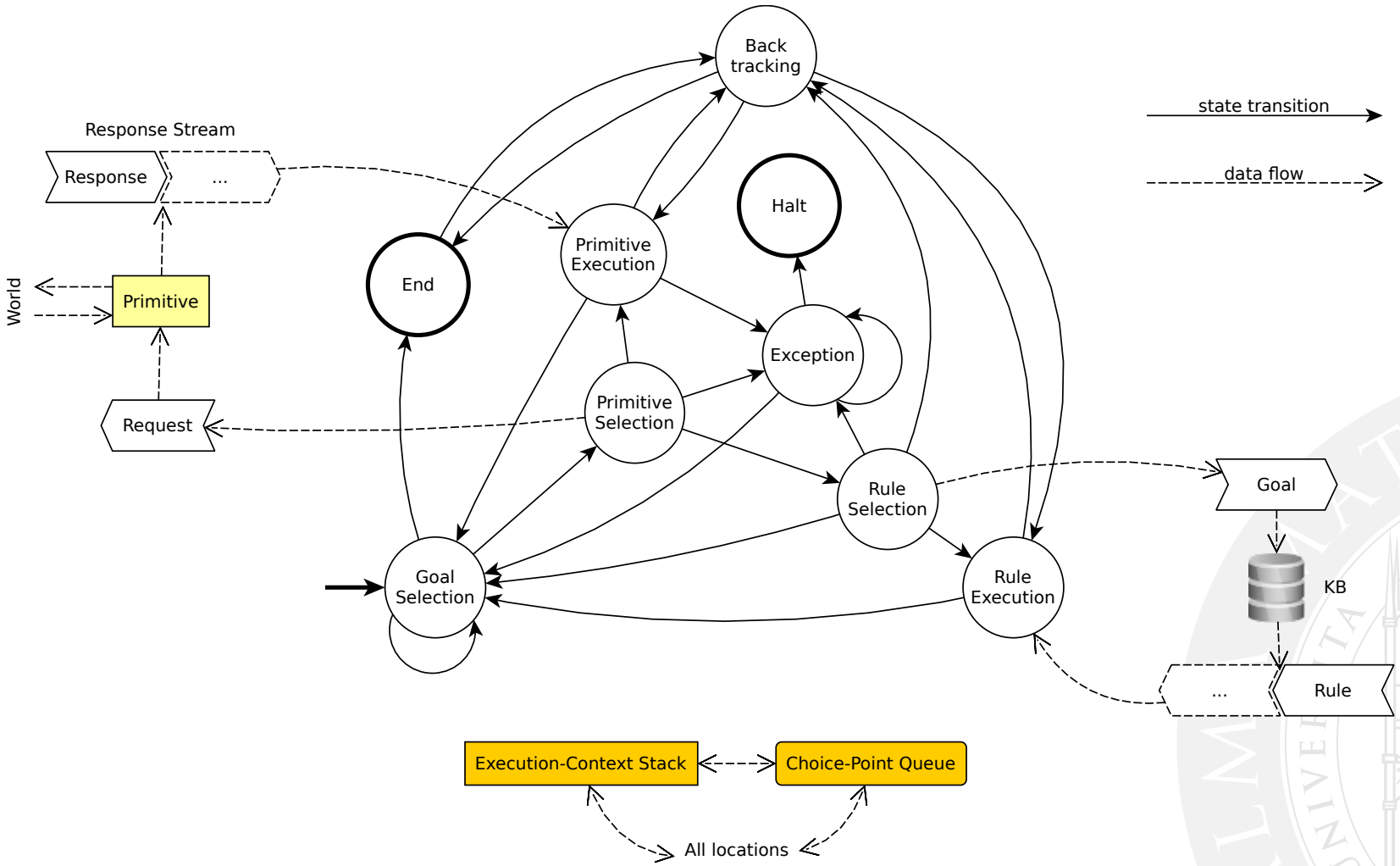
eg to alter the execution context or the destination state

`AbstractClassicSolver` abstract class for classic-like solvers

- supporting the creation of custom solvers reusing (part of) the PSM



Prolog State Machine in a Nutshell I



Prolog State Machine in a Nutshell II

Main Features

- 9 locations (1 initial, 2 ending)
- 2 ancillary data structures
- infinite possible states

Ancillary data structures

Execution Context **Stack** (**E**) tracking currently exploring items of the proof-tree

Choice Point **Queue** (**C**) tracking unexplored items of the proof-tree

Prolog State Machine in a Nutshell III

Locations

Goal Selection selects the next (sub-)goal to be proved, possibly popping from **E**

Primitive Selection looks for a primitive to solve the selected (sub-)goal;

Primitive Execution if some is found, the 1^{st} response is consumed, while a choice points are appended to **C** for the others

Rule Selection if no primitive is selected, some rule is looked for instead, for the same sub-goal;

Rule Execution if a rule is found, resolution proceeds by pushing a new execution context to **E**, to tackle the body;

Backtracking if no rule is found, the sub-goal is considered failed and resolution continues taking the next choice point in **C**.

End reached when there are no more goals or no more choice points.

Execption reached some primitive responses carries an exception. This is where catching occurs.

Halt reached when an exception is not caught.

Prolog State Machine, Formally

Formal specification here:

`https://github.com/tuProlog/2pkt-state-machine/releases/latest`

→ download the `2pkt-state-machine-X.Y.Z-DATE.pdf` file

Next in Line...

Overview on IO in LP I

Purpose / Requirements

Provide `Solvers` with I/O facilities:

1. in a platform-independent way
   - ie supporting both JVM and JS
2. in a solver-implentation-agnostic way
   - ie supporting both *classic*, and *streams* solvers
   - \+ supporting future sorts of solvers too
3. in a location-transparent way
   - ie supporting transparent access to both *local* and *remote* resources

Overview on IO in LP II

Main challenges

- Kotlin MP does not provide platform independent I/O API
- JVM supports accessing both local and remote resources via streams
  - blocking, synchronous API
  - async API available too
- JS server-side (i.e. NodeJS) supports accessing both local and remote resource via callbacks
  - non-blocking, asynchronous API
  - sync API available too
- JS browser-side supports accessing remote resources via callbacks
  - non-blocking, asynchronous API
  - NO sync API

→ requirements 1 and 3 are hard to attain

Overview on IO in LP III

Strategy

1. Define minimal multi-platform API for I/O
2. Provide platform-specific implementations for that API
3. Provide implementation of `Channels` exploiting such API
4. Provide logic predicated to manipulate `Channels` from LP

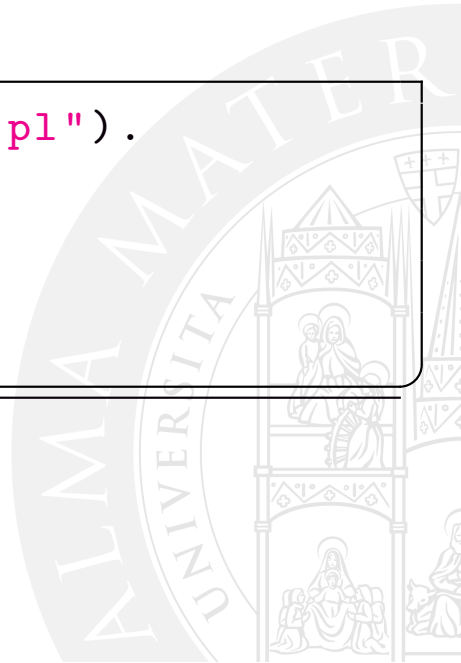
Overview on IO in LP IV

Goal

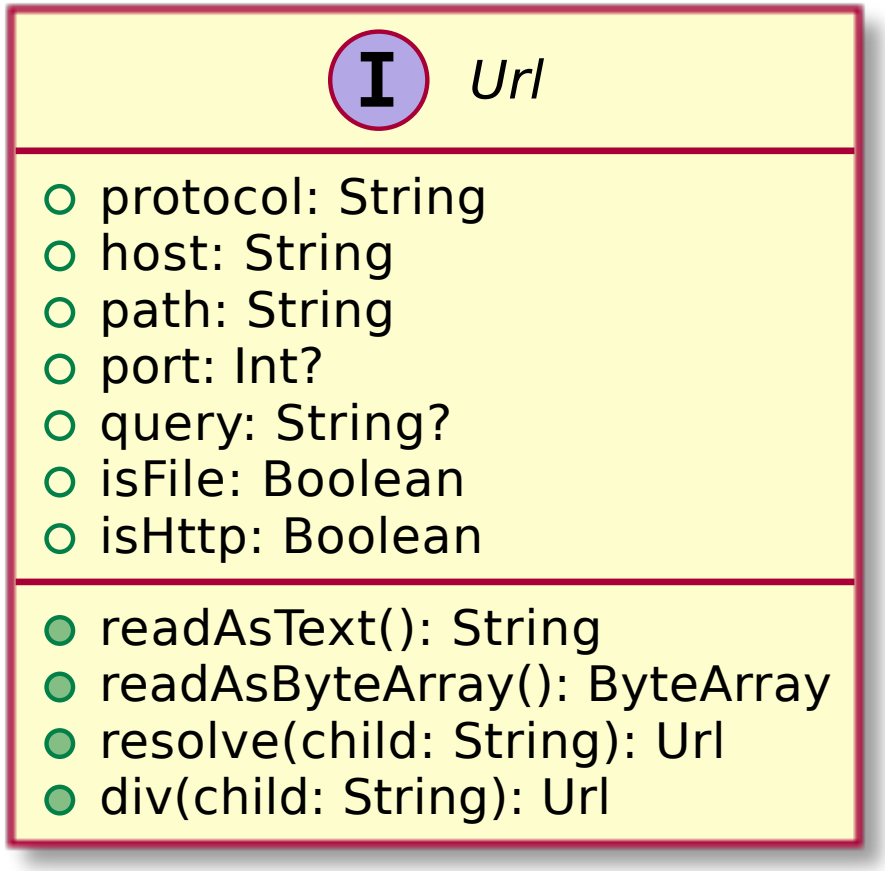
Support access to local/remote resources to any solver on all platforms

Cf. the `consult/1` predicate/directive, supporting loading of external theories into the current solver's static KB:

```
:- consult("http://example.com/path/to/remote/theory.pl").
    % importing remote theory from the Web

:- consult("file:///path/to/local/theory.pl").
    % importing theory from the local file system
```



The multi-platform notion of `Url` I



The multi-platform notion of `Url` II

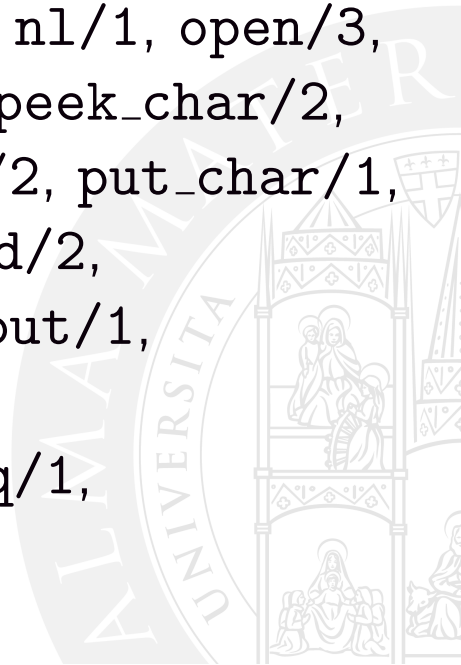
Purpose

- Mimicking Java's URL type
- Providing unified multi-platform API for I/O
- Supporting requirements 1 and 3

Content of the IOLib I

`it.unibo.tuprolog.solve.libs.io.IOLib`

primitives: `at_end_of_stream/0`, `at_end_of_stream/1`, `char_conversion/2`, `close/1`, `close/2`, `consult/1`, `current_char_conversion/2`, `current_input/1`, `current_output/1`, `flush_output/1`, `get_byte/1`, `get_byte/2`, `get_char/1`, `get_char/2`, `get_code/1`, `get_code/2`, `nl/1`, `open/3`, `open/4`, `peek_byte/1`, `peek_byte/2`, `peek_char/1`, `peek_char/2`, `peek_code/1`, `peek_code/2`, `put_byte/1`, `put_byte/2`, `put_char/1`, `put_char/2`, `put_code/1`, `put_code/2`, `read/1`, `read/2`, `read_term/2`, `read_term/3`, `set_input/1`, `set_output/1`, `set_theory/1`, `stream_property/2`, `write/2`, `write_canonical/1`, `write_canonical/2`, `write_eq/1`, `write_eq/2`, `write_term/2`, `write_term/3`



Content of the `IOLib` II

operators: none

theory: none

functions: none

Takeaway

`IOLib` = I/O predicates from the Prolog ISO Standard [1], realised as `Solver`-independent primitives

Next in Line...

Overview on the OOP Lib Design I

Perspective

`Solvers` → let OOP exploit LP

`Primitives` → let LP wrap/call OOP

:oop-lib → lets LP exploit OOP

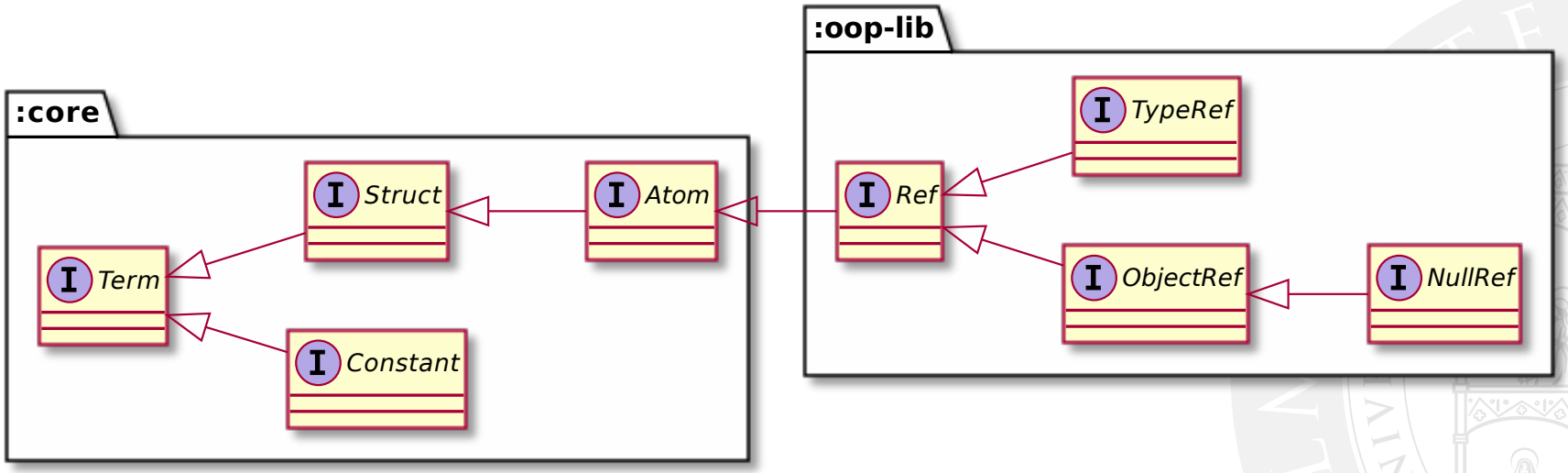
Overview on the OOP Lib Design II

Concept:

1. Two more sub-sorts of `Atom`, generically called `References`

   `ObjectReferences` — representing references to objects
   `TypeReferences` — representing references to types

   `eg` interfaces, classes, etc.



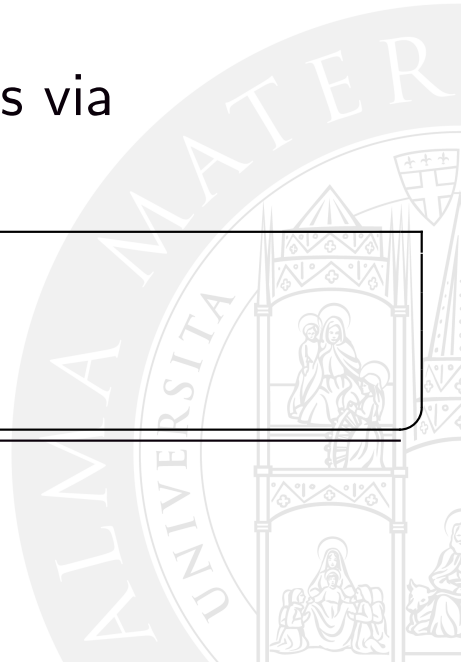
Overview on the OOP Lib Design III

2. TypeReferenes can be attained from type names via `type/2`:

```
?- type('java.lang.StringBuilder', Type).
        % yes.
        %   Type = <type:java.lang.StringBuilder>
```

3. `ObjectReferenes` can be attained from `TypeReferenes` via `new_object/2`:

```
?- type('java.lang.StringBuilder', Type), new_object(Type, Obj).
        % yes.
        %   Type = <type:java.lang.StringBuilder>
        %   Obj = <object:java.lang.StringBuilder#6223cf0b>
```



Overview on the OOP Lib Design IV

4. References' methods can be invoked via the following syntax:

$$\texttt{Ref.methodName(Arg}_1\texttt{, ..., Arg}_n\texttt{)}$$

5. References' properties, fields, 0-ary methods can be similarly accessed:

`Ref.propertyName` or `Ref.'PropertyName'`

6. Results of properties / fields / methods can be bound to variables via ':='/2:

```
?- type('java.lang.System', System), Out := System.out, Out.println("hello world").
    % hello world
    % yes.
    %     System = '<type:java.lang.System>'
    %     Out = '<object:java.io.PrintStream#6adbc9d>'
```

Overview on the OOP Lib Design V

7. Terms are automatically converted into object of the most adequate type, upon method invocation, and result are converted back into terms
   - we call this smart cast
   - smart overload selection is performed in case of ambiguity

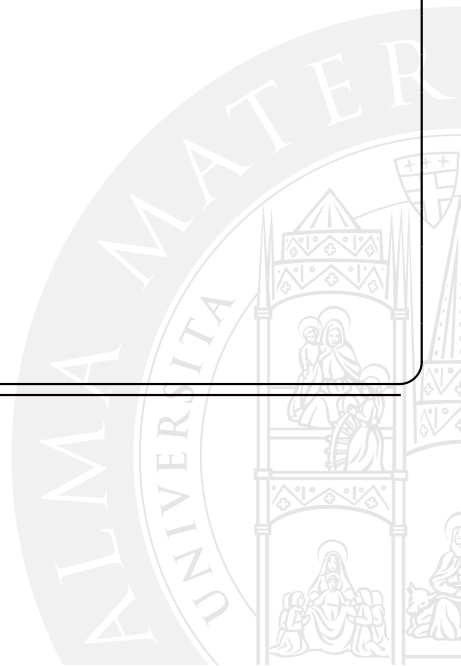
```
?- type('java.lang.StringBuilder', SB), new_object(SB, S),
   S.append('hello world'), % atom converted into string
   S.replace(0, 1, 'H'), % integers converted into int
   X := S.toString, % string converted back into atom
   atom(X).

% yes.
%    SB = '<type:java.lang.StringBuilder>'
%    S = '<object:java.lang.StringBuilder#4bbb49b0>'
%    X = 'Hello world'
```

   - recall that Java's `StringBuilder.`*`append`* has several overloads

Overview on the OOP Lib Design VI

8. Aliases are available to quick-access common types or objects

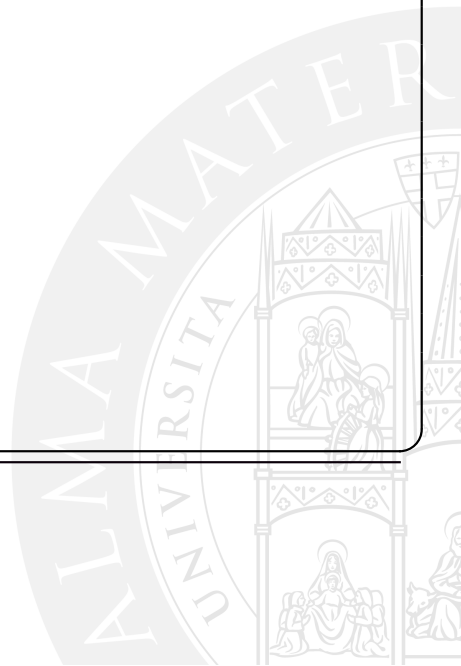
```
?- alias(Alias, Ref).
% yes.
%     Alias = string
%     Ref = '<type:kotlin.String>'
% yes.
%     Alias = array
%     Ref = '<type:kotlin.Array>'
% ...
% yes.
%     Alias = stdin
%     Ref = '<object:java.io.BufferedInputStream#4ec4f3a0>'
% yes.
%     Alias = stdout
%     Ref = '<object:java.io.PrintStream#6adbc9d>'
% ...
```



Overview on the OOP Lib Design VII

9. Aliases can be
   - inspected via `alias/2`
   - added (resp. removed) via `register/2` (resp. `unregister/1`)
   - used in place of actual References via the de-aliasing operator `$/1`

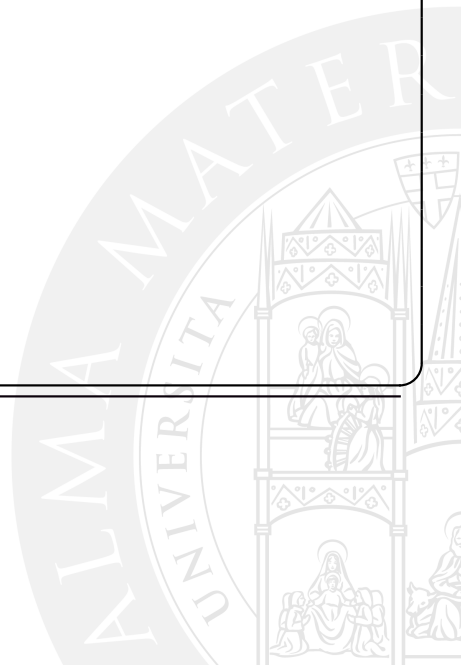
```
?- type('java.lang.StringBuilder', Type),
   register(Type, string_builder),
   new_object($string_builder, SB),
   SB.append(hello),
   X := SB.toString,
   $stdout.println(X),
   unregister(string_builder).

% yes.
%    Type = '<type:java.lang.StringBuilder>'
%    SB = '<object:java.lang.StringBuilder#624a24f6>'
%    X = hello
```



Overview on the OOP Lib Design VIII

10. Specific casts can be performed via the cast operator as/2
    - which enables *explicit* overload selection

```
?- type('java.lang.StringBuilder', StringBuilder),
   new_object(StringBuilder, SB),
   X is 1 / 3,
   SB.append(X as $double),
   SB.append('|'),
   SB.append(X as $float),
   S := SB.toString.

% yes.
%    StringBuilder = '<type:java.lang.StringBuilder>'
%    SB = '<object:java.lang.StringBuilder#4aedaf61>'
%    X = 0.333333333
%    S = '0.333333333|0.33333334'
```



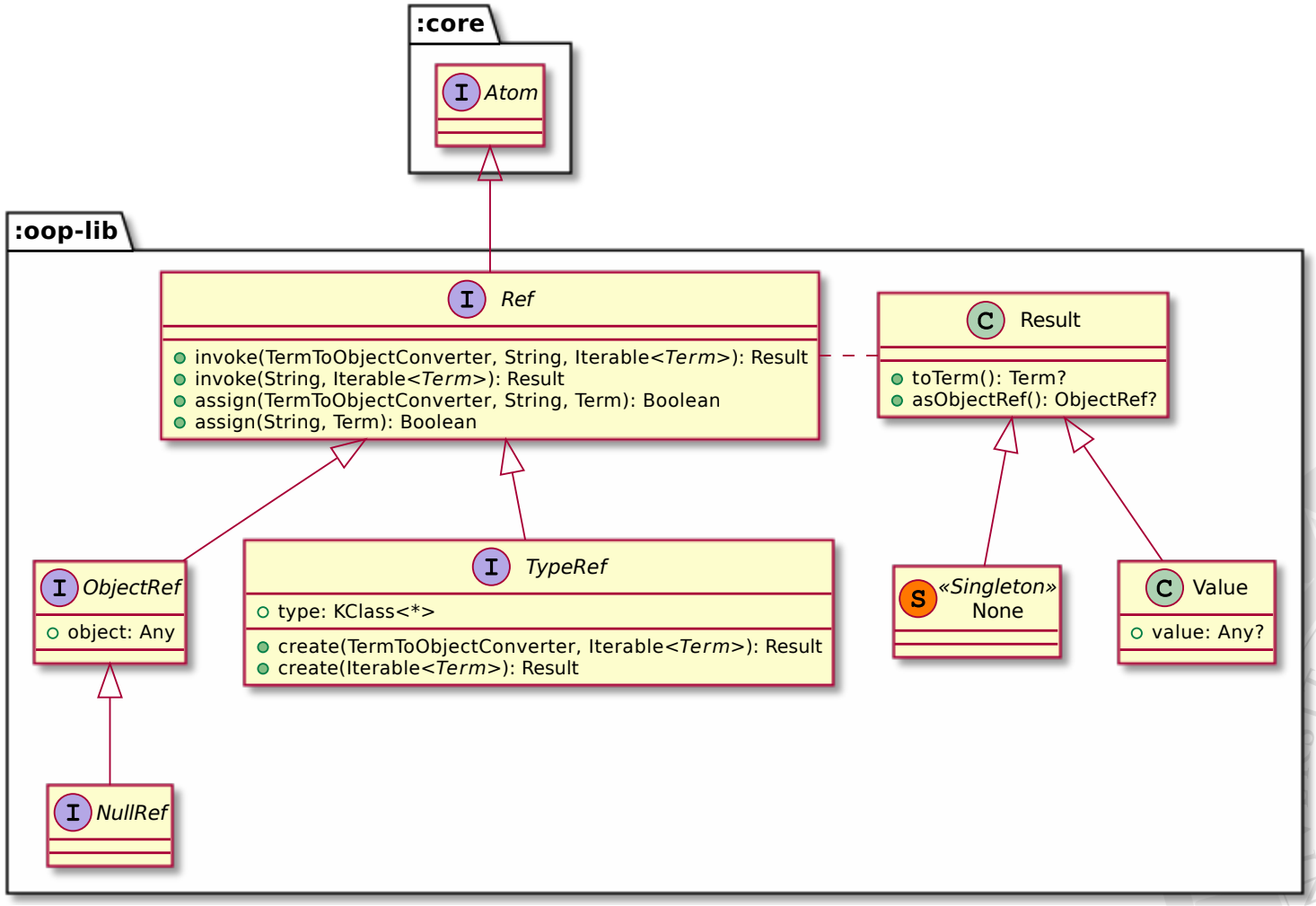
Overview on the OOP Lib Design IX

Notable remarks about *:oop-lib*

- API is currently considered unstable
- Design is tailored on Kotlin's OOP model:
  - fields → properties
  - static members → companion objects
  - strong reliance on Kotlin's reflection
  - pervasive exploitation of `KClasses`
- Design is multi-platform, yet only JVM implementation works
  - Kotlin's JS reflection is poorly supported ATM
  - Kotlin's JVM reflection is quite buggy ATM
  - Kotlin's reflection has a very clean design
- On JVM, fields and static are supported too!

Focus on...

The Ref Type Hierarchy I



The Ref Type Hierarchy II

The Ref type is a sub-type of Atom

Base type for all sorts of references

`invoke(TermToObjectConverter, String, ⟨Terms⟩): Result`
: invokes a method on the current Reference by name, passing the provided Terms as arguments, which are converted into objects using the provided TermToObjectConverter

`invoke(String, ⟨Terms⟩): Result` invokes a method on the current Reference by name, passing the provided Terms as arguments, which are converted into objects using the *default* TermToObjectConverter

`assign(TermToObjectConverter, String, Term): Boolean` assign the provided Term as the new value of some property/field of the current Reference, selected by name. The term is converted into an object using the provided TermToObjectConverter

The Ref Type Hierarchy III

`assign(String, Term): Boolean` assign the provided `Term` as the new value of some property/field of the current `Reference`, selected by name. The term is converted into an object using the *default* `TermToObjectConverter`

The Ref Type Hierarchy IV

The TypeRef type is a sub-type of Ref

Base type for all sorts of references to types—there including interfaces, classes, Kotlin objects, etc.

`type: KClass<*>` the type being referenced

`invoke(...)` methods inherited from `Ref` can be used to call both static methods and `companion objects'` methods

`assign(...)` methods inherited from `Ref` can be used to set both static properties/methods or companion objects' ones

`create(TermToObjectConverter, ⟨Terms⟩): Result` calls the constructor of the referenced type to create a new instance. The provided terms are passed to the constructor after being converted into objects using the provided `TermToObjectConverter`

The Ref Type Hierarchy V

`create(`⟨*Terms*⟩`): Result` calls the constructor of the referenced type to create a new instance. The provided terms are passed to the constructor after being converted into objects using the *default* `TermToObjectConverter`

The Ref Type Hierarchy VI

The ObjectRef type is a sub-type of Ref

Base type for all sorts of references to objects—i.e. instances of types

`object: Any` the object being referenced

`invoke(...)` methods inherited from `Ref` can be used to call both instance methods on the referenced objects

`assign(...)` methods inherited from `Ref` can be used to set instance properties/fields on the referenced objects

The Ref Type Hierarchy VII

The Result type

Instances of `Result` aim to wrap objects:

- returned by invoking methods on `References`
- created from `TypeReferences`

Two sub-types:

`None` representing the lack of result

`Value` representing the return of an object

`value: Any?` is the returned object

`toTerm(): Term?` converts the returned value (if any) into a `Term`, using the default `ObjectToTermConverter`

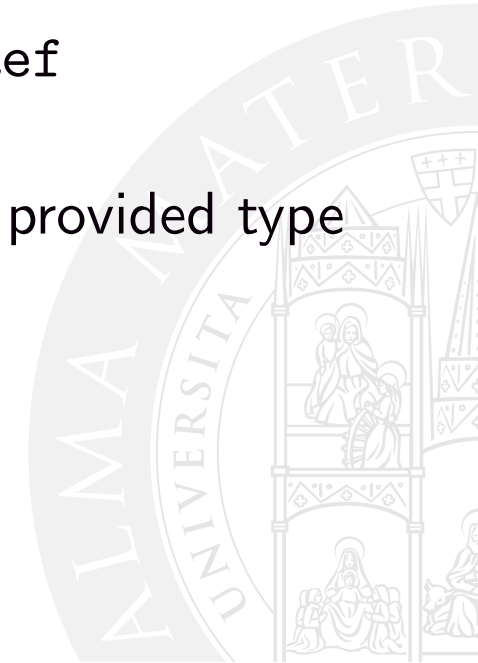
`asObjectRef(): ObjectRef?` converts the returned value (if any) into an `ObjectReference`

Creation of References

`ObjectRef.of(Any?)` creates a new `ObjectRef` to the provided object.
May return a `NullRef` is the provided object is `null`

`ObjectRef.NULL` returns the singleton instance of `NullRef`

`TypeRef.of(KClass<*>)` creates a new `TypeRef` to the provided type



Focus on...

The `TypeFactory` Type I

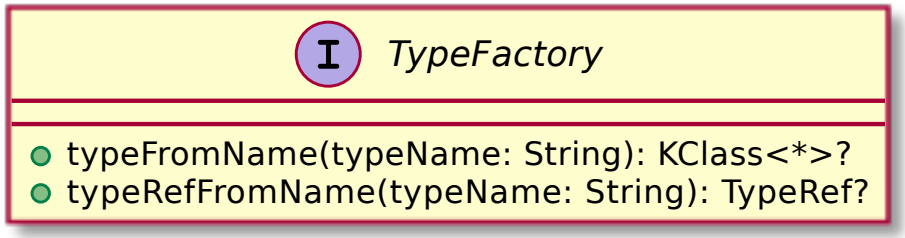
Purpose

- Loading OOP types from names
- Instantiating `TypeReferences` from type names

Concept

- Type discovery/loading is platform specific...
- ...a platform-agnostic type is necessary in a multi-platform module

The TypeFactory Type II



The TypeFactory Type III

Platform-specific names for types

JVM: `name.of.package.TypeName`

JS: `module-name:name.of.package.TypeName`

The `TermToObjectConverter` Type I

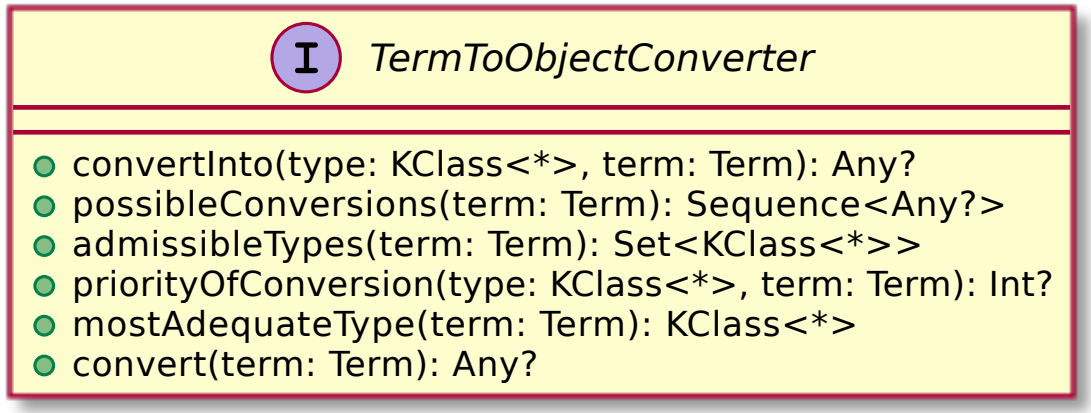
Purpose

- Converting arguments of methods into objects upon invocation
- Converting arguments of constructors into objects upon creation
- Converting values of properties into objects upon assignment

Concept

- Each LP type may correspond to several OOP types
- Preferences may exists among target OOP types
- Each LP type has 1 preferred corresponding OOP type

The TermToObjectConverter Type II



The TermToObjectConverter Type III

LP Type	OOP Types[a]
NullRef	Nothing
Var	Nothing
ObjectRef	type of refereced object
TypeRef	KClass<*>
Truth	Boolean, String
Atom	String, Char[b]
Real	BigDecimal, Double, Float
Integer	BigInteger, Long[c], Int[c], Short[c], Byte[c], Double, Float
Struct	as(_, ⟨*Type*⟩) → ⟨*Type*⟩ $⟨*ObjectRef*⟩ → type of refereced object $⟨*TypeRef*⟩ → KClass<*> otherwise → TermToObjectConversionException
otherwise	TermToObjectConversionException

[a] written in decreasing priority order

[b] only if the atom's length is 1

[c] only if the integer's value is within the OOP type's range

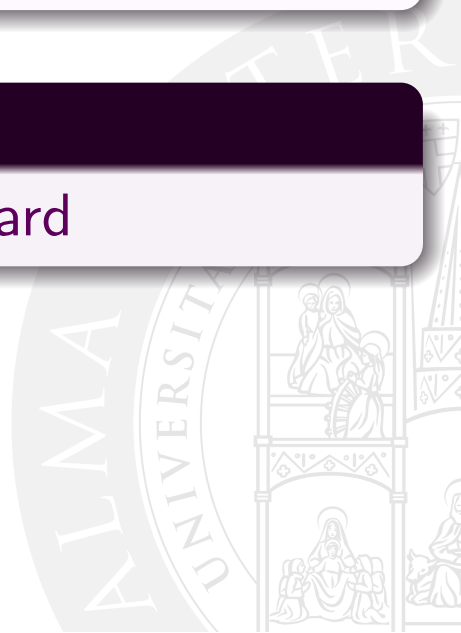
The `ObjectToTermConverter` Type I

Purpose

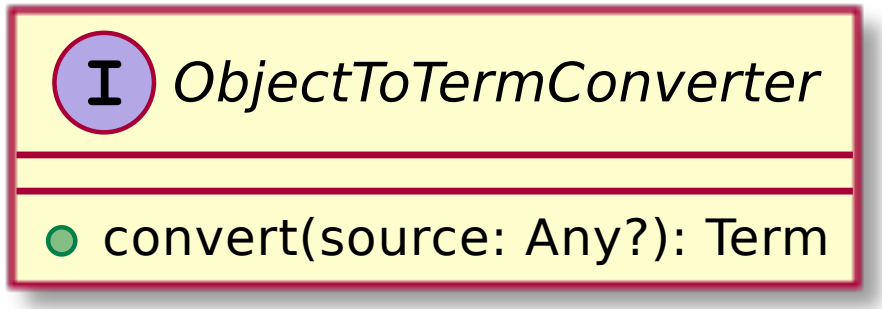
- Converting results of methods invocations into `Terms`
- Converting results of constructors into `ObjectReferences`

Concept

- Conversion of OOP types into LP ones is straightforward



The `ObjectToTermConverter` Type II



The ObjectToTermConverter Type III

OOP Type	LP Type
Nothing	NullRef
BigInteger	Integer
Long	Integer
Int	Integer
Short	Integer
Byte	Integer
BigDecimal	Real
Double	Real
Float	Real
String	Atom
Char	Atom
Boolean	Truth
KClass	TypeRef
otherwise	ObjectRef

Focus on...

The `OverloadSelector` Type I

Purpose

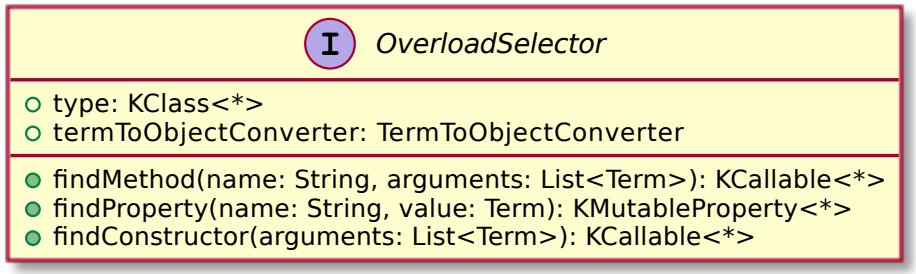
Selecting the most adequate overload

- of a method / property / constructor
- w.r.t. the admissible LP→OOP conversions for actual arguments
- when invoking / assigning / creating

Concept

- Many strategies may be exploited to select an overload
- `OverloadSelector` is the general API to do so
- They essentially select target OOP types for actual arguments

The `OverloadSelector` Type II

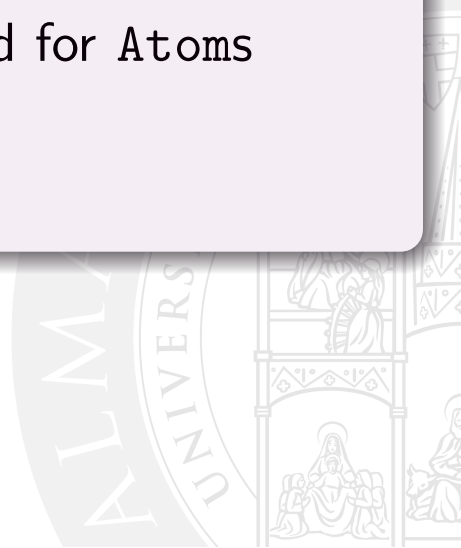


The `OverloadSelector` Type III

Current strategy (JVM)

An attempt is performed to select the available overload which

- maximizes the sum of the proprities of all target OOP types
- using explicit casts (i.e. `as/2`) as strong constraints
- taking the most specific formal type into account
  - eg among `Any` and `String`, the latter should be preferred for `Atoms`
- taking actual values into account
  - eg 129 cannot be converted into `Byte`



Focus on...

Content of the OOPLib

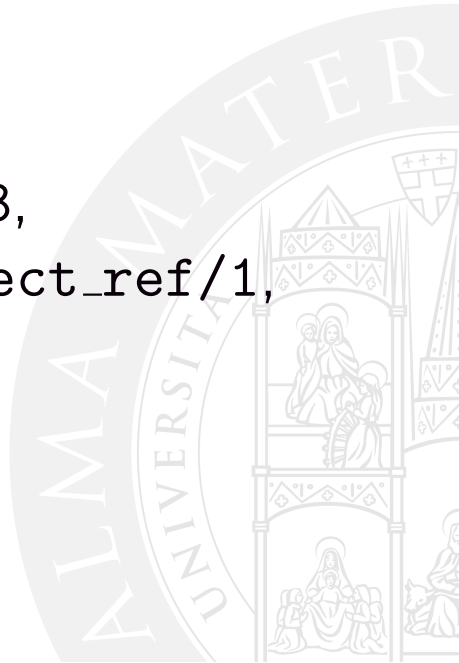
`it.unibo.tuprolog.solve.libs.oop.OOPLib`

operators: `'.'/2`, `':='/2`, `'as/2'`, `'$'/1`

theory: `':='/2`, `'.'/2`, `fluent_reduce/2`, `new_object/2`, `property_reduce/2`

primitives: `assign/3`, `cast/3`, `type/2`, `invoke_method/3`, `invoke_strict/3`, `new_object/3`, `null_ref/1`, `object_ref/1`, `ref/1`, `register/2`, `type_ref/1`, `unregister/1`

functions: none



Next in Line...

Overview on the DSL I

Perspective

`Solvers` → let OOP exploit LP

`Primitives` → let LP wrap/call OOP

:oop-lib → lets LP exploit OOP

*:dsl-** → let Kotlin expoit LP with Prolog-like syntax

Overview on the DSL II

Purpose

- integrating OOP, FP and LP
- bringing OOP software engineering techniques to LP and FP
- bringing declarativity from FP and LP to OOP
- enriching the Kotlin language in a fruitful and natural way

Overview on the DSL III

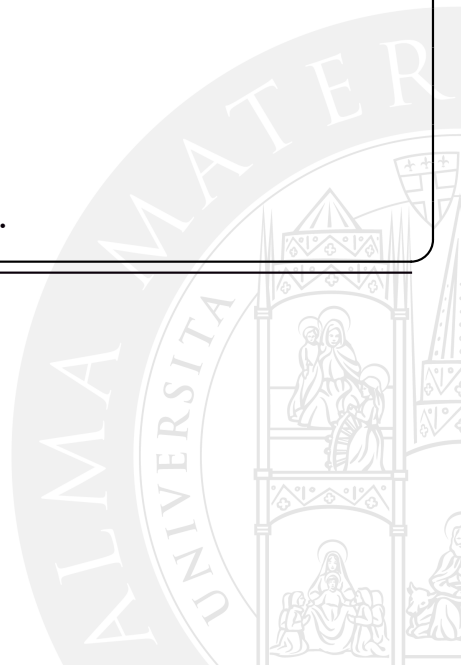
DSL in practice

- writing tests using LP-like syntax
- ! without relying a parser
  - to keep the dependency graph small/sparse
  - to prevent developers from using the parser unless strictly needed

Overview on the DSL IV

```
prolog {
  staticKb(
    rule {
      "ancestor"(X, Y) `if` "parent"(X, Y)
    },
    rule {
      "ancestor"(X, Y) `if` (
          "parent"(X, Z) and
          "ancestor"(Z, Y))
    },
    fact { "parent"("abraham", "isaac") },
    fact { "parent"("isaac", "jacob") },
    fact { "parent"("jacob", "joseph") }
  )

  for (sol in solve("ancestor"("abraham",
      X)))
    if (sol is Solution.Yes)
      println(sol.substitution[X])
}
```

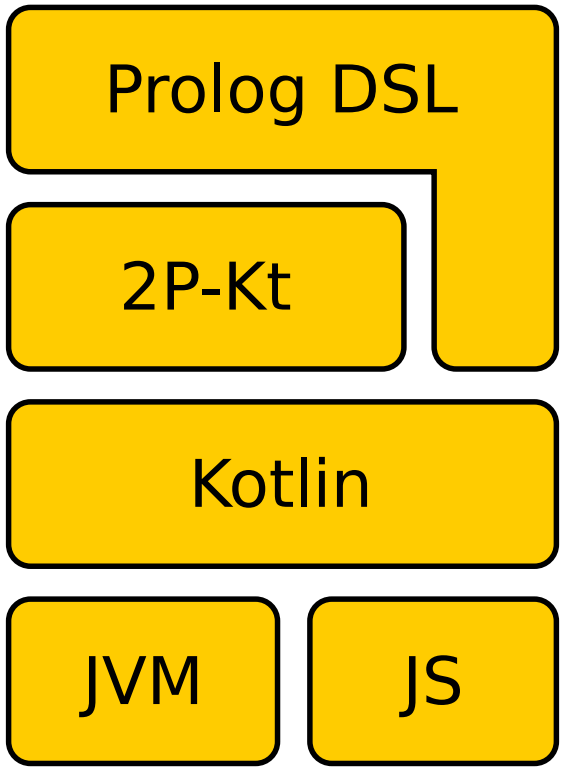
```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z),
    ancestor(Z, Y).

parent(abraham, isaac).
parent(isaac, jacob).
parent(jacob, joseph).


?- ancestor(abraham, X).
```



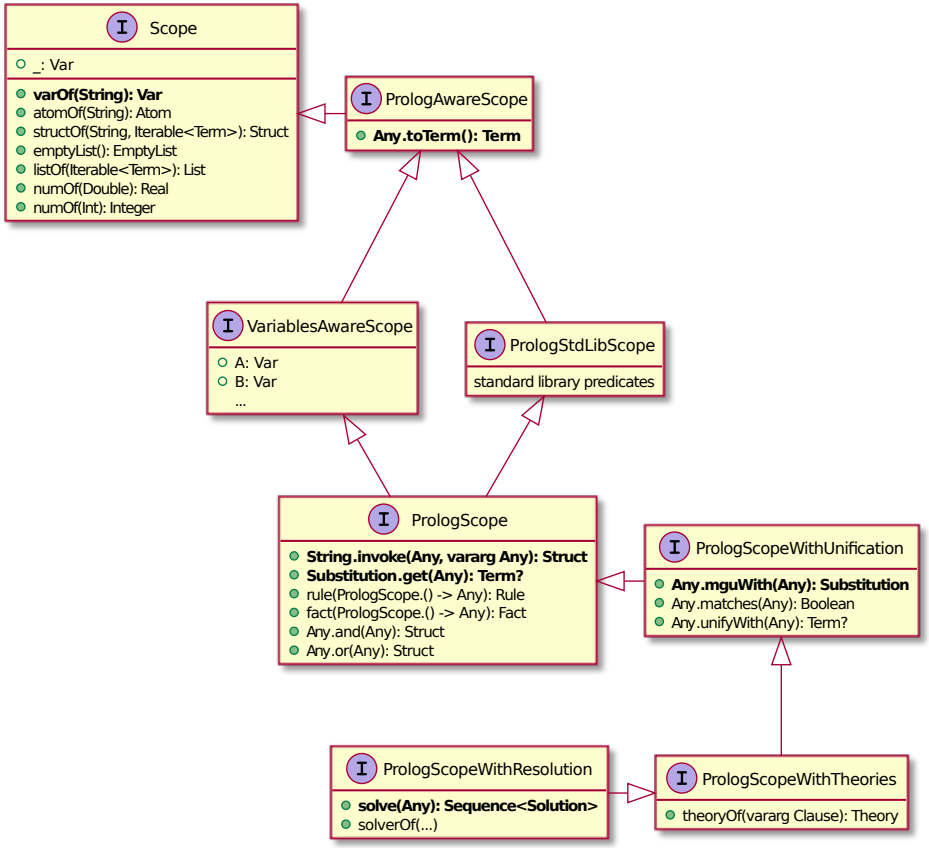
Design Principles of the DSL (cf. [4]) I

1. the DSL has to be a *strict* extension of Kotlin
   → no Kotlin feature is hindered by using the DSL
2. the DSL has to be fully *interoperable* with the hosting language
   → all Kotlin features may be exploited by the DSL
3. the DSL has to be fully *encapsulated* and *identifiable*
   → preventing unintended usage
4. the DSL should be as close as possible to LP, syntactically and semantically
   → easing its exploitation by logic programmers

Architecture I



Architecture II



Architecture III

The `PrologAwareScope` interface

- Enables the conversion of Kotlin objects to Prolog `Terms` via the `Any.toTerm()` extension method
- using the following type mapping:
  - a Kotlin number is converted either to a Prolog real or integer number
  - a Kotlin string is converted either to a Prolog variable or atom
  - a Kotlin boolean is always converted to an atom
  - a Kotlin iterable is always converted into a Prolog list
    - ! each item has to be convertible to a `Term`
  - Kotlin objects that are sub-types of `Term` remain unaffected
  - if the object cannot be converted to a `Term`, an error is raised

Architecture IV

The `VariablesAwareScope` interface

- provides `A-Z` shortcuts to instantiate Prolog variables

The `PrologStdLibScope` interface

- provides a number of methods that allow the invocation of standard library predicates

  e.g. `assert`, `retract`, `bagof`, `consult`, etc. . .

Architecture V

The PrologScope interface

- enables the construction of compound terms
  - → via the `String.invoke()` extension method

  e.g. `"human"("socrates")`
- allows to retrieve variable substitutions in a leaner fashion, without having to create explicit variables through `varOf()`
  - → via the `Substitution.get(Any)` extension method

  e.g. `substitution.get("X")`
- adds the capability of converting Kotlin objects into logic clauses
  - → via the `rule` and `fact` factory methods
  - → via the `Any.and(Any)` and `Any.or(Any)` extension methods

Architecture VI

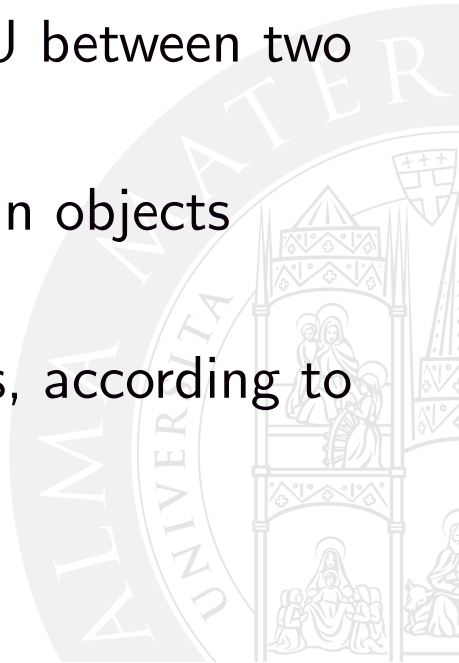
The `PrologScopeWithUnification` interface

- exposes a number of extension methods providing unification-related support to Kotlin objects
  - ! provided that they can be converted into `Terms`

`Any.mguWith(Any): Substitution` computes the MGU between two Kotlin objects

`Any.matches(Any): Boolean` checks whether two Kotlin objects match or not, according to logic unification

`Any.unifyWith(Any): Term?` unifies two Kotlin objects, according to logic unification



Architecture VII

The `PrologScopeWithTheories` interface

- allows the combination of clauses into logic theories
  - → via the `theoryOf` factory method

The `PrologScopeWithResolution` interface

- adds the capability of performing logic queries and consuming their solutions
- provides methods that allow
  - instancing Prolog solvers
    - → via `solverOf(...)`
  - loading static and dynamic Prolog theories
    - → via `staticKb(...)`, `dynamicKb(...)`
  - invoking Prolog queries on the loaded theories
    - → via `solve(...)`

Usage Example

```
fun nQueens(n: Int) = prolog {
  staticKb(
    rule {
      "no_attack"(("X1" and "Y1"), ("X2" and "Y2")) `if` (
          ("X1" `=!=` "X2") and // infix operator
          ("Y1" `=!=` "Y2") and
          (("Y2" - "Y1") `=!=` ("X2" - "X1")) and
          (("Y2" - "Y1") `=!=` ("X1" - "X2")) // arithmetic expr
        )
    },
    fact { "no_attack_all"(`_`, emptyList) },
    rule {
      "no_attack_all"("C", consOf("H", "Hs")) `if` (
          "no_attack"("C", "H") and
          "no_attack_all"("C", "Hs")
        )
    },
    fact { "solution"(`_`, emptyList) },
    rule {
      "solution"("N", consOf(("X" and "Y"), "Cs")) `if` (
          "solution"("N", "Cs") and
          between(1, "N", "Y") and // built-in predicate
          "no_attack_all"(("X" and "Y"), "Cs")
        )
    }
  )
  return solve("solution"(n, (1 .. n).map { it and "Y$it" }))
}
```

Next in Line...

The TermParser Type I

The TermParser interface

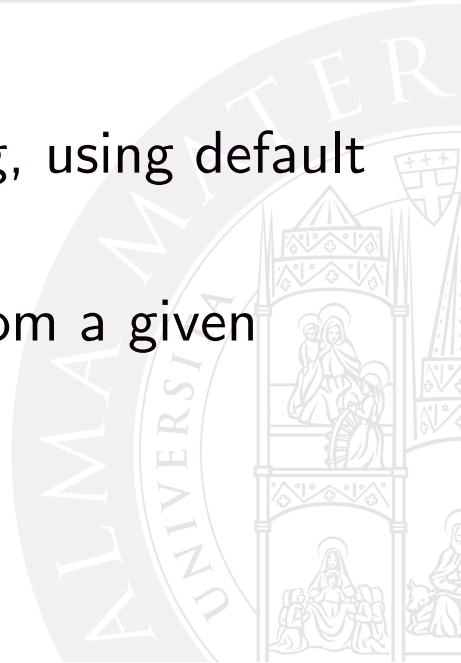
- provides methods to parse Terms from strings
  - ! throwing a ParseException upon failure
- allows to select the operators used for parsing

All methods come in two forms:

`parse⟨T⟩(String)`: ⟨T⟩ parses a ⟨T⟩ from a given string, using default operators

`parse⟨T⟩(String, OperatorSet)`: ⟨T⟩ parses a ⟨T⟩ from a given string, using the given operators

with ⟨T⟩ a sub-type of Term



The TermParser Type II

Static factories for `TermParser`:

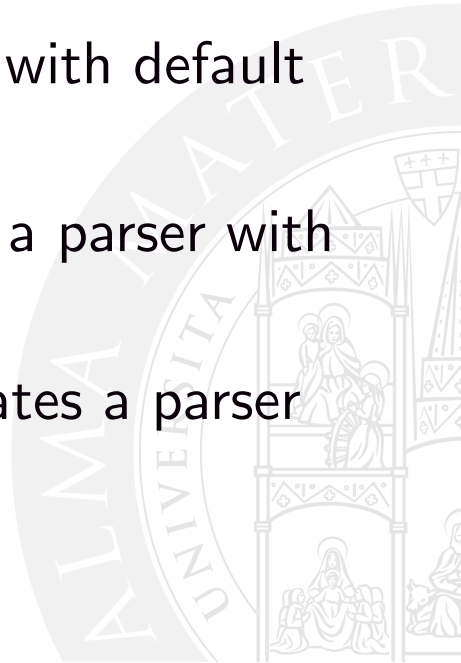
`withNoOperator: TermParser` creates a parser with no operators loaded

`withStandardOperators: TermParser` creates a parser with standard operators loaded

`withDefaultOperators: TermParser` creates a parser with default operators loaded (`standard` operators)

`withOperators(OperatorsSet): TermParser` creates a parser with the given `OperatorSet` loaded

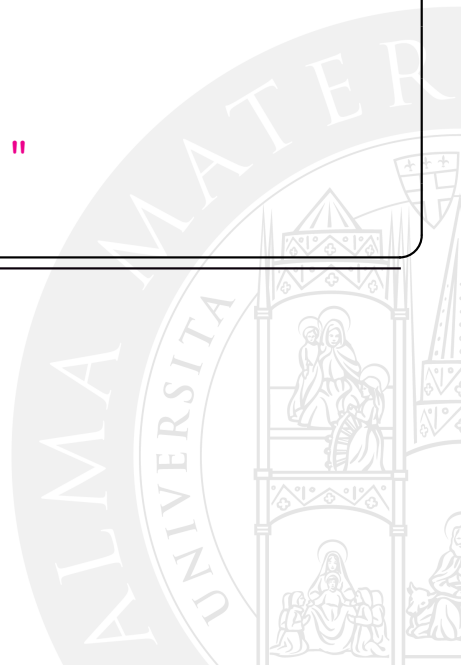
`withOperators(vararg Operator): TermParser` creates a parser with the given operators loaded



The TermParser Type III

Successful Term parsing:

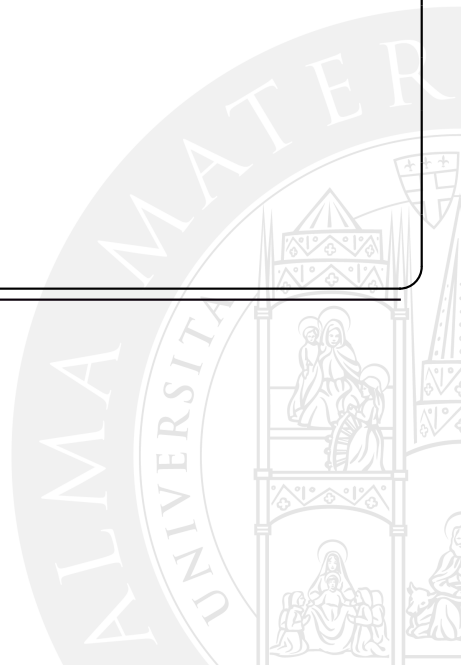
```
val parser = TermParser.withDefaultOperators

val atom: Atom = parser.parseAtom("a")
val list: Struct = parser.parseStruct("[1, 2, 3]")
val clause: Clause = parser.parseClause(
    "grandfather(X, Y) :- father(X, Z), father(Z, Y)."
)
```



The TermParser Type IV

Term parsing failure:

```
val parser = TermParser.withDefaultOperators
val input = "[1, 2, ]"

try {
    val list = parser.parseStruct(input) // will fail
} catch (e: ParseException) {
    println(e.message)
}
```



The TermReader Type (JVM-only) I

The TermReader interface

- provides methods to parse Terms
- input is loaded **lazily** from an InputStream/Reader/String
- only available on the JVM as part of the jvm implementations of the parser-core module

`readTerm(Reader): Term?` tries to read a term from the given Reader, with default operators

`readTerm(Reader, OperatorSet): Term?` tries to read a term from the given Reader, with the given operator set

`readTerm(InputStream): Term?` tries to read a term from the given InputStream, with default operators

`readTerm(InputStream, OperatorSet): Term?` tries to read a term from the given InputStream, with the given operator set

The TermReader Type (JVM-only) II

Multiple-Term reading methods:

`readTerms(Reader): Sequence<Term>` reads a sequence of terms from the given `Reader`, with default operators

`readTerms(Reader, OperatorSet): Sequence<Term>` reads a sequence of terms from the given `Reader`, with the given operator set

`readTerms(InputStream): Sequence<Term>` reads a sequence of terms from the given `InputStream`, with default operators

`readTerms(InputStream, OperatorSet): Sequence<Term>` reads a sequence of terms from the given `InputStream`, with the given operator set

`readTerms(String): Sequence<Term>` reads a sequence of terms from the given `String`, with default operators

`readTerms(String, OperatorSet): Sequence<Term>` reads a sequence of terms from the given `String`, with the given operator set

The TermReader Type (JVM-only) III

Static factories for TermReader:

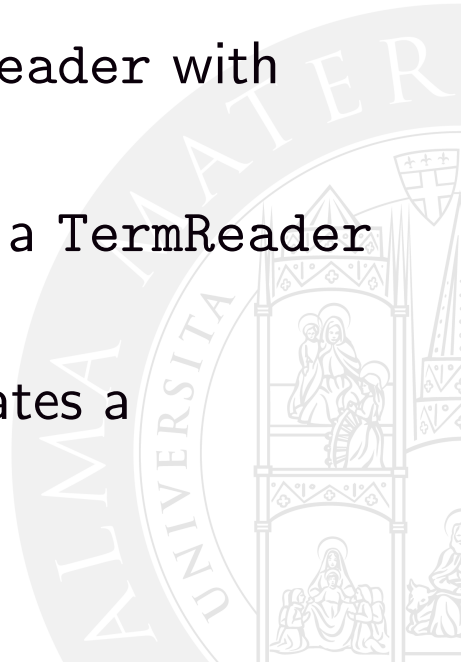
`withNoOperator: TermReader` creates a TermReader with no operators loaded

`withStandardOperators: TermReader` creates a TermReader with standard operators loaded

`withDefaultOperators: TermReader` creates a TermReader with default operators loaded (standard operators)

`withOperators(OperatorsSet): TermReader` creates a TermReader with the given OperatorSet loaded

`withOperators(vararg Operator): TermReader` creates a TermReader with the given operators loaded



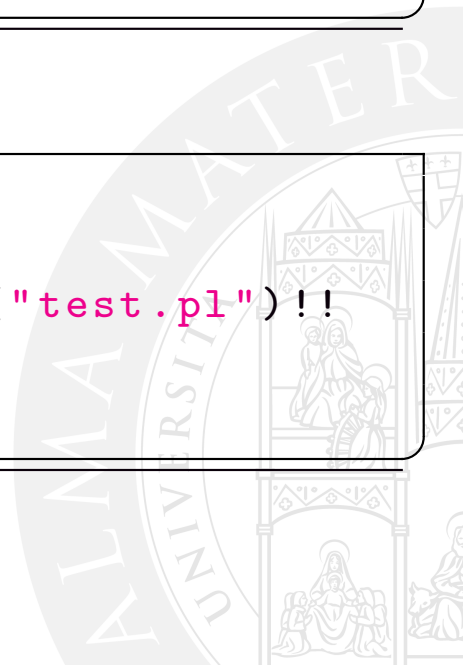
The TermReader Type (JVM-only) IV

Reading Terms from a String:

```
val reader = TermReader.withDefaultOperators

val terms: Sequence<Term> = reader.readTerms(
    "f(1) f(2) f(3) f(4) f(5)"
)
```

Reading Terms from a File:

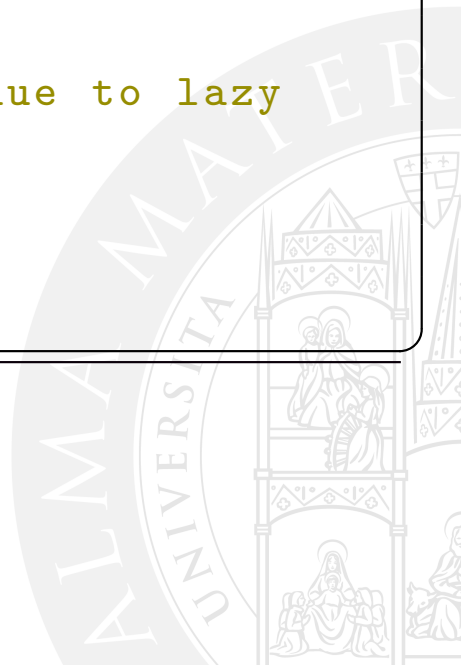
```
val reader = TermReader.withDefaultOperators
val inputStream =
    this::class.java.classLoader.getResourceAsStream("test.pl")!!

val terms = reader.readTerms(inputStream)
```



The TermReader Type (JVM-only) V

Reading failure:

```
val reader = TermReader.withDefaultOperators
val input = "f(1) f(2) f(3 f(4) f(5)" // malformed input

try {
    val terms = reader.readTerms(input)
    val count = terms.count() // parsing fails here due to lazy
    evaluation
} catch (e: ParseException) {
    println(e.message)
}
```



The ClausesParser Type I

The ClausesParser interface

- provides methods to parse multiple Clauses from strings
  - ! throwing a ParseException upon failure
- allows to select the operators used for parsing
- very similar to TermParser, but is capable of parsing entire theories instead of single Terms

`parseTheory(String, OperatorSet): Theory` parses a Theory from the given string, using the given operator set

`parseTheory(String): Theory` parses a Theory from the given string, using default operators

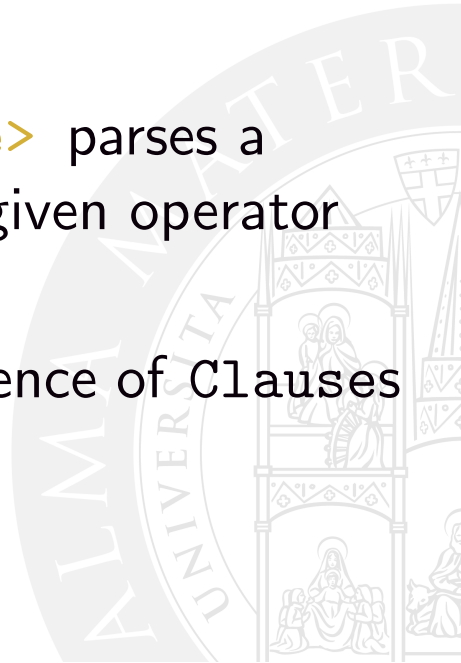
The ClausesParser Type II

`parseClausesLazily(String, OperatorSet): Sequence<Clause>`
: parses a *lazily evaluated* sequence of `Clauses` from the given string, with the given operator set

`parseClausesLazily(String): Sequence<Clause>` parses a *lazily evaluated* sequence of `Clauses` from the given string, with default operators

`parseClauses(String, OperatorSet): List<Clause>` parses a sequence of `Clauses` from the given string, with the given operator set

`parseClauses(String): List<Clause>` parses a sequence of `Clauses` from the given string, with default operators



The ClausesParser Type III

Static factories for `ClausesParser`:

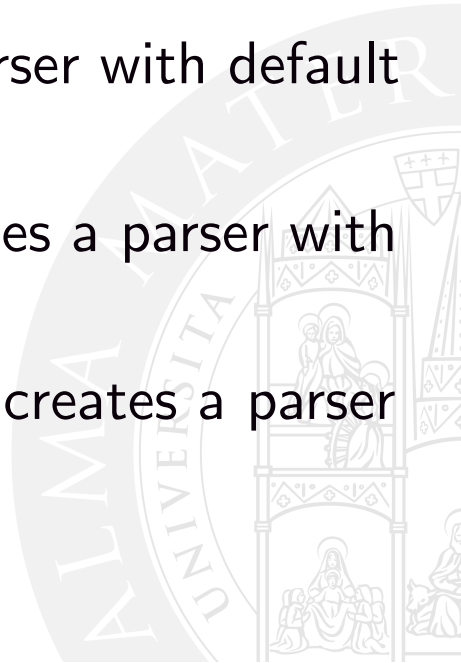
`withNoOperator: ClausesParser` creates a parser with no operators loaded

`withStandardOperators: ClausesParser` creates a parser with standard operators loaded

`withDefaultOperators: ClausesParser` creates a parser with default operators loaded (`standard` operators)

`withOperators(OperatorSet): ClausesParser` creates a parser with the given `OperatorSet` loaded

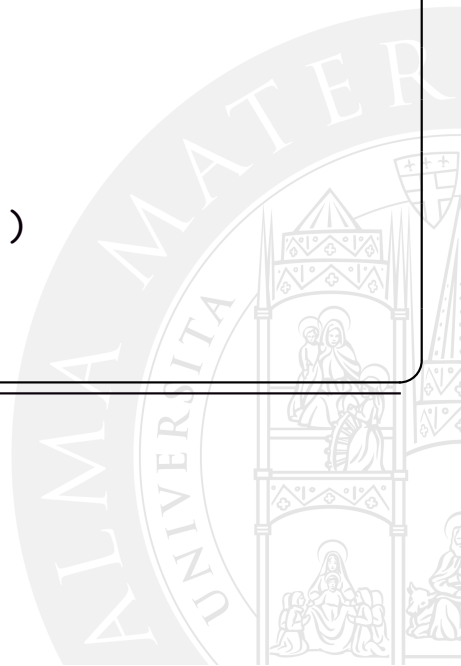
`withOperators(vararg Operator): ClausesParser` creates a parser with the given operators loaded



The ClausesParser Type IV

Successful clauses parsing (as Theory, Sequence and List of Clauses):

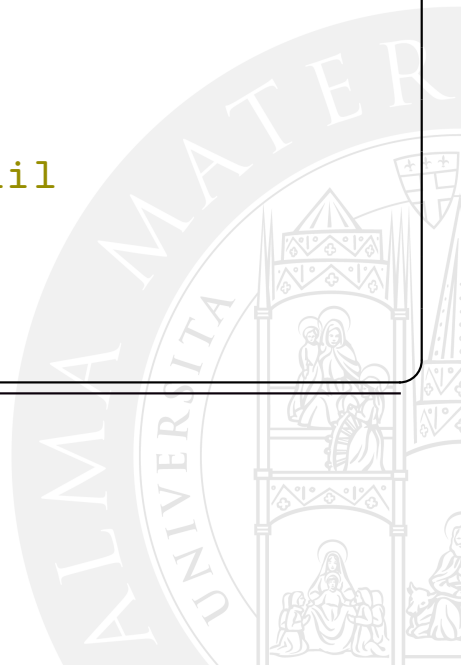
```
val parser = ClausesParser.withDefaultOperators
val input = """
    grandfather(X, Y) :- father(X, Z), father(Z, Y).
    father(abraham, isaac).
    father(isaac, jacob).
""".trimIndent()

val theory: Theory = parser.parseTheory(input)
val clauses: List<Clause> = parser.parseClauses(input)
val lazyClauses: Sequence<Clause> =
    parser.parseClausesLazily(input)
```



The ClausesParser Type V

Clauses parsing failure:

```
val parser = ClausesParser.withDefaultOperators
val input = """
    father(abraham, isaac)
    father(isaac, jacob)
""".trimIndent() // no full stops after each clause

try {
    val theory = parser.parseTheory(input) // will fail
} catch (e: ParseException) {
    println(e.message)
}
```



Next in Line...

(De)Serialisation Overview I

Serialisation

- performed via `Serializers` and `Objectifiers`
- `Serializers` convert `Terms`/`Theorys` into strings
  - ! formatted according to some `MIME` type
- `Objectifiers` are used by `Serializers` to convert `Terms`/`Theorys` into translatable objects first
  - ! acting as a middleware layer

De-serialisation

- serialisation's dual process
- converting strings into `Terms` and `Theorys`
- employing `Deserializers` and `Deobjectifiers`

(De)Serialisation Overview II

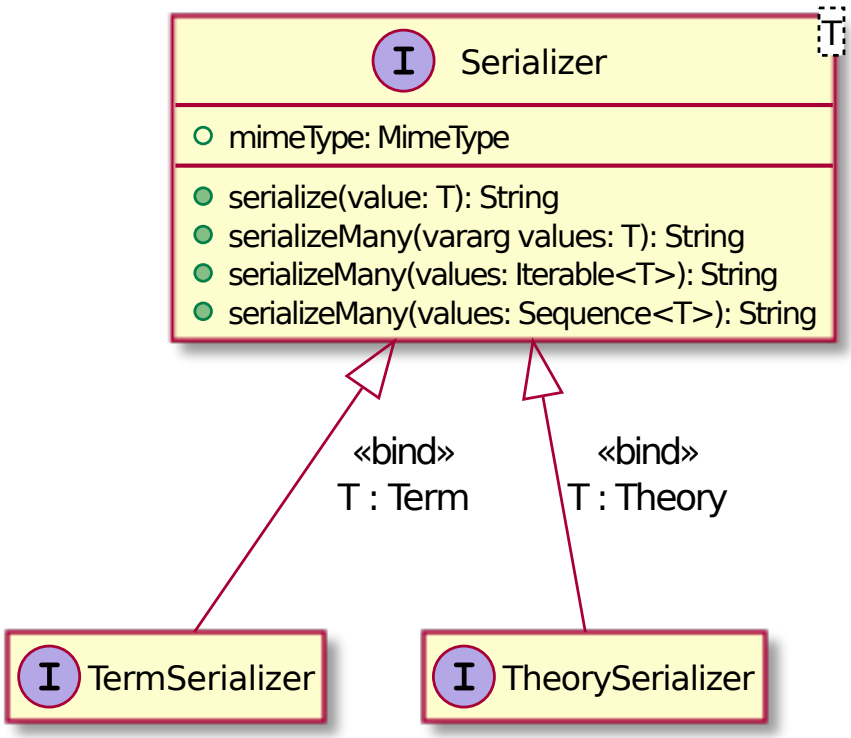
`Term` vs `Theory` (de)serialisation

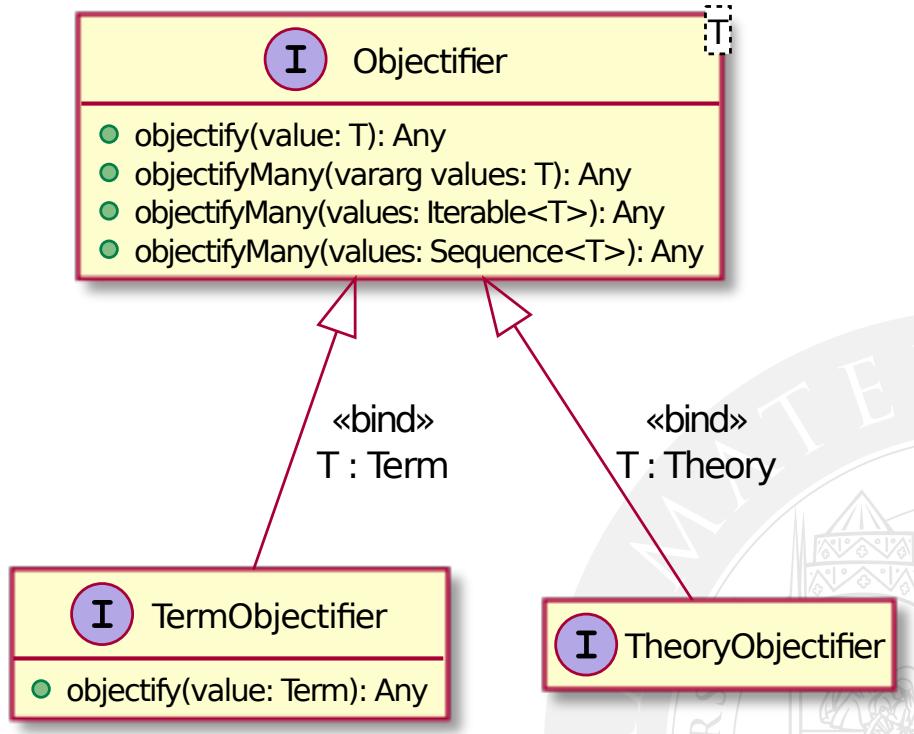
- 2P-Kt's design tackles both single and multiple serialisation strategies
  - `:serialize-core` concerns single-`Term` (de)serialisation
  - `:serialize-theory` concerns `Theory` (de)serialisation

Supported MIME types

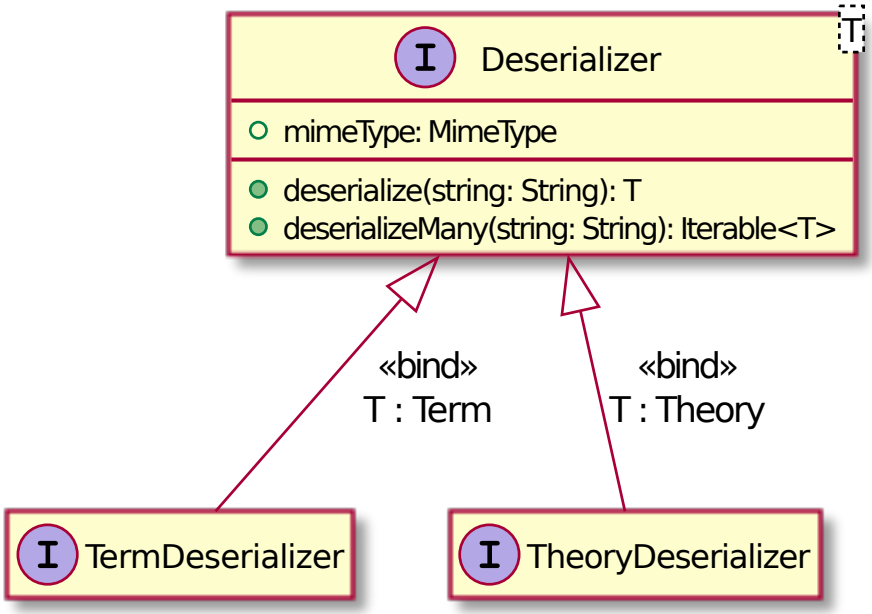
- currently, (de)serialisation supports `JSON` and `YAML`

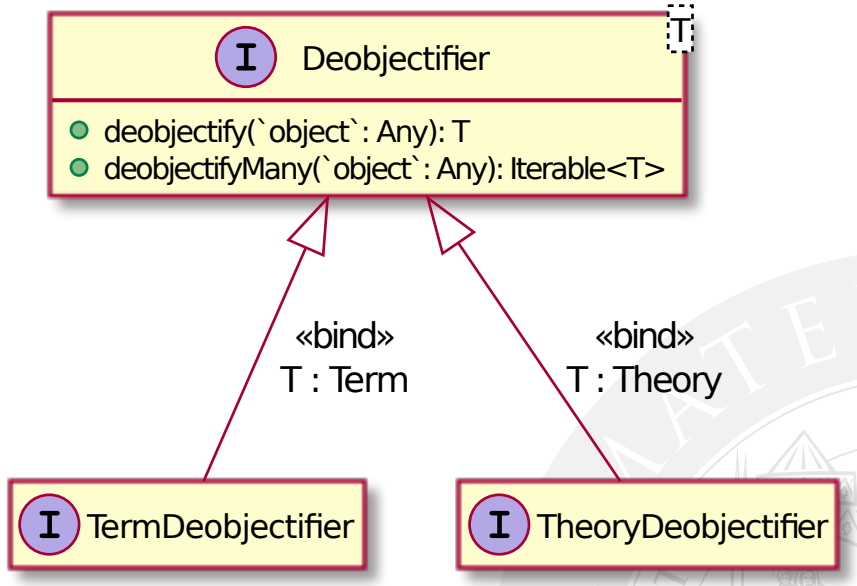
The Serializer/Objectifier hierarchy I





The Deserializer/Deobjectifier hierarchy I





The TermSerializer Type I

The TermSerializer type

- exposes methods that perform Term serialisation
- targeting a specific MIME type

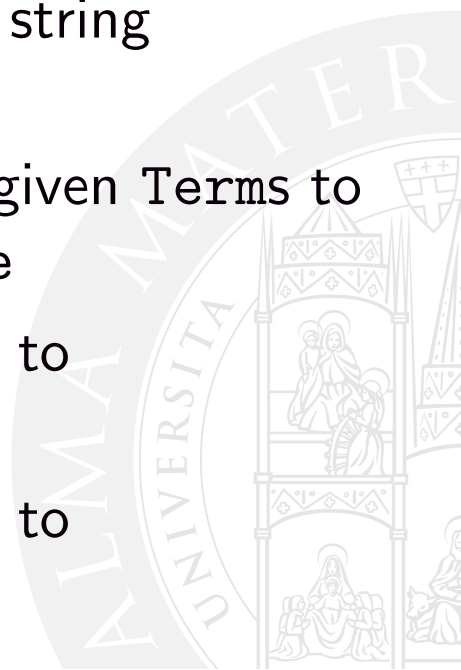
`mimeType: MimeType` the format to which input Terms are serialized

`serialize(Term): String` serialises a given Term to a string formatted according to the selected mimeType

`serializeMany(vararg Term): String` serialises the given Terms to a string formatted according to the selected mimeType

`serializeMany(Iterable<Term>): String` analogous to `serializeMany(varargs Term): String`

`serializeMany(Sequence<Term>): String` analogous to `serializeMany(varargs Term): String`



The TermSerializer Type II

Static factories for `TermSerializer`:

`of(MimeType): TermSerializer` creates a new `TermSerializer` employing the given format

The TermDeserializer Type I

The TermDeserializer Type

- provides methods that convert strings back to Terms
- according to a specific MIME type

`mimeType: MimeType` the format from which the input string is de-serialised

`deserialize(String): Term` de-serialises the given string according to the selected mimeType, yielding a single Term

`deserializeMany(String): Iterable<Term>` de-serialises the given string according to the selected mimeType, yielding multiple Terms

The TermDeserializer Type II

Static factories for `TermDeserializer`:

`of(MimeType): TermDeserializer` creates a new `TermDeserializer` according to the given format

The TheorySerializer Type I

The TheorySerializer Type

- exposes methods that perform Theory serialisation
- targeting a specific MIME type

`mimeType: MimeType` the format to which input Theorys are serialized

`serialize(Theory): String` serialises a given Theory to a string formatted according to the selected `mimeType`

`serializeMany(vararg Theory): String` serialises the given Theorys to a string formatted according to the selected `mimeType`

`serializeMany(Iterable<Theory>): String` analogous to `serializeMany(varargs Theory): String`

`serializeMany(Sequence<Theory>): String` analogous to `serializeMany(varargs Theory): String`

The TheorySerializer Type II

Static factories for `TheorySerializer`:

`of(MimeType): TheorySerializer` creates a new `TheorySerializer` employing the given format

The TheoryDeserializer Type I

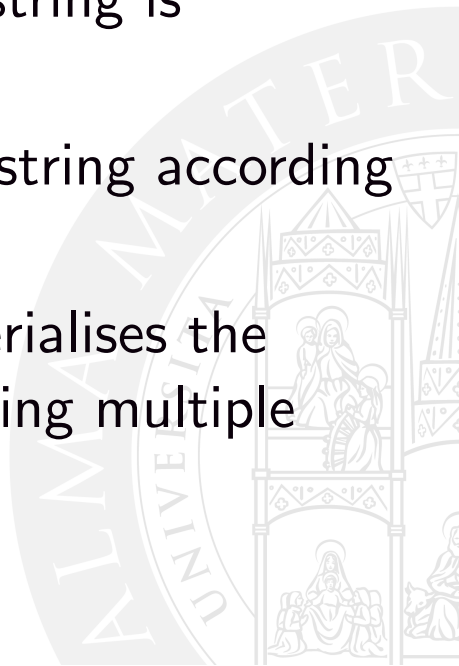
The TheoryDeserializer Type

- provides methods that convert strings back to Theorys
- according to a specific MIME type

`mimeType: MimeType` the format from which the input string is de-serialised

`deserialize(String): Theory` de-serialises the given string according to the selected `mimeType`, yielding a single `Theory`

`deserializeMany(String): Iterable<Theory>` de-serialises the given string according to the selected `mimeType`, yielding multiple `Theorys`



The TheoryDeserializer Type II

Static factories for `TheoryDeserializer`:

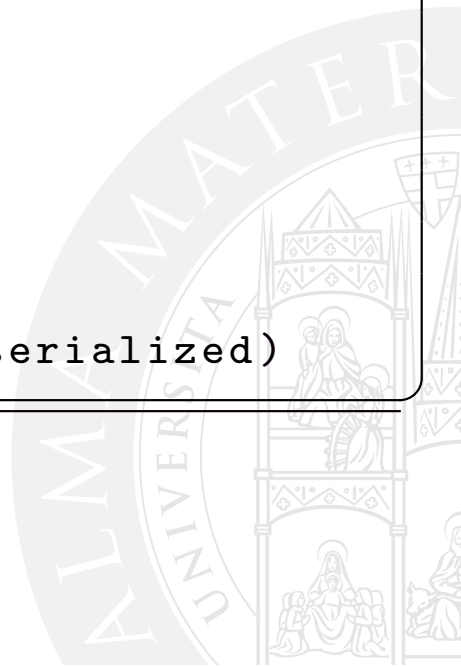
`of(MimeType): TheoryDeserializer` creates a new `TheoryDeserializer` according to the given format

Terms/Theory to JSON Mapping I

Type	JSON Type	Logic	JSON	YAML
Atom	string	a	"a"	a
		'a b'	"a b"	a b
Numeric	number	42	42	42
Integer	object	42	{"integer":"42"}	integer: "42"
Real	object	4.2	{"real":"4.2"}	real: "4.2"
List	object	[⟨t_1⟩, ..., ⟨t_n⟩]	{"list":[⟨j_1⟩,...,⟨j_n⟩]}	list: [⟨y_1⟩,...,⟨y_n⟩]
		[⟨t_1⟩, ..., ⟨t_n⟩ | ⟨t⟩]	{"list":[⟨j_1⟩,...,⟨j_n⟩],"last":⟨j⟩}	list: [⟨y_1⟩,...,⟨y_n⟩] last: ⟨y⟩
Set	object	{⟨t_1⟩, ..., ⟨t_n⟩}	{"set":[⟨j_1⟩,...,⟨j_n⟩]}	set: [⟨y_1⟩,...,⟨y_n⟩]
Tuple	object	(⟨t_1⟩, ..., ⟨t_n⟩)	{"tuple":[⟨j_1⟩,...,⟨j_n⟩]}	tuple:[⟨y_1⟩,...,⟨y_n⟩]
Struct	object	f(⟨t_1⟩, ..., ⟨t_n⟩)	{"fun":"f","args":[⟨j_1⟩,...,⟨j_n⟩]}	fun: f args: [⟨y_1⟩, ..., ⟨y_n⟩]
Truth	boolean	true	true	true
		false	false	false
	string	fail	"fail"	fail
Var	object	X	{"var":"X"}	var: X
Indicator	object	⟨t_1⟩ / ⟨t_2⟩	{"name":⟨j_1⟩, "arity":⟨j_2⟩}	name: ⟨y_1⟩ arity: ⟨y_2⟩
Rule	object	⟨t_1⟩ :- ⟨t_2⟩	{"head":⟨j_1⟩, "body":⟨j_2⟩}	head: ⟨y_1⟩ body: ⟨y_2⟩
Fact	object	⟨t⟩ :- true	{"head":⟨j⟩}	head: ⟨y⟩
Directive	object	:- ⟨t⟩	{"body":⟨j⟩}	body: ⟨y⟩

Complete Usage Example I

```
val parser = ClausesParser.withDefaultOperators
val serializer = TheorySerializer.of(MimeType.Json)
val deserializer = TheoryDeserializer.of(MimeType.Json)

val theory = parser.parseTheory("""
    grandfather(X, Y) :- father(X, Z), father(Z, Y).
    father(abraham, isaac).
    father(isaac, jacob).
""".trimIndent())

val serialized: String = serializer.serialize(theory)

val deserialized: Theory = deserializer.deserialize(serialized)
```



Next in Line...

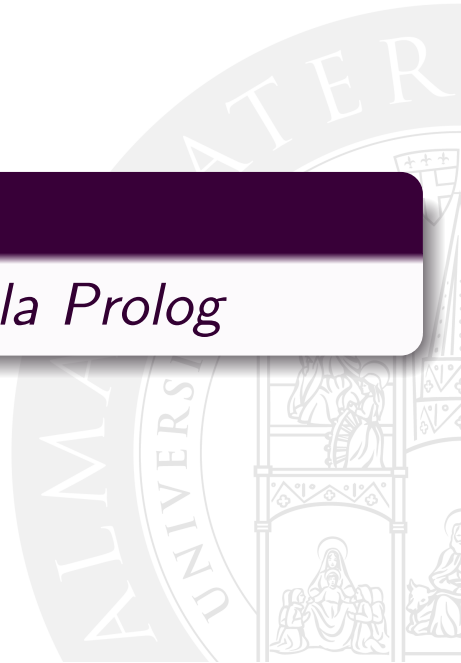
Intended Usages of 2P-Kt

Usage as a JVM or JS **library**

For developers willing to re-use one or more functionalities

Usage as a JVM or Web **application**

For users willing to use logic solvers interactively, *à la Prolog*



Focus on...

Using 2P-Kt as a library – Overview

- 2P-Kt modules are compiled, packed, and deployed on both:
  Maven Central Repository — Multiplatform + JVM packages only
  GitHub Packages — Multiplatform + JVM packages only
  GitHub Releases —JVM executables only + Dokka documentation
  NPM — JS packages only
- Developers willing to use 2P-Kt facilities can
  - declare a Maven dependency (in JVM or MPP projects)
  - declare a NPM dependency (in JS projects)
- JS and JVM artefacts are versioned using `SemVer` and versions are kept aligned
- Documentation is produced for the Kotlin API alone
  - the Java or JS API are very close to the Kotlin one
  - yet, knowing how Kotlin code is compiled is very useful

Using 2P-Kt as a JVM library – Overview

- 2P-Kt modules are compiled into ordinary JVM bytecode
- They can be exploited as dependencies in most JVM-targetting languages

  eg Java, Scala, Groovy, etc.

- They can be imported using the Maven naming convention:

  `groupId:` `it.unibo.tuprolog`
  `artifactId:` *`MODULE_NAME`*`-jvm`
  `version:` a SemVer-compliant string

  which are kept coherent on all Maven Repositories[1]
  - provided that some dependency management system is in place

    eg Gradle, Maven, etc.

[1] https://search.maven.org/search?q=g:it.unibo.tuprolog

Kotlin–Java interoperability I

Required readings

- https://kotlinlang.org/docs/java-to-kotlin-interop.html
- https://kotlinlang.org/docs/java-interop.html

Kotlin–Java interoperability II

Kotlin to Java in a nutshell

- `Unit` → `void`
  - the remainder of the type system is almost identical
- `varargs` → `...` (Java's varargs)
- public instance properties → getters (+ setters, if `var`)
- private/protected instance properties → fields
- objects → singleton classes with `INSTANCE` constant
- optional parameters → method overloads
  - if `@JvmOverloads` are provided
- companion objects → `Companion` constants
- companion objects members → accessible via `Companion` constants
  - unless `@JvmStatic` annotations are provided

Kotlin–Java interoperability III

2P-Kt Conventions

- `@JvmStatic` annotation on all public members of companion objects
- overload provided where possible in presence of optional parameters

Using 2P-Kt as a JS library – Overview

- 2P-Kt modules are transpiled into ordinary JS code
  - targetting NodeJS
    - transpiled NodeJS code can then be packed via WebPack
- They can be exploited as dependencies in most JS-targetting languages
  - there including TypeScript, etc
- They can be imported as NPM dependencies[2]

  `organization:` `@tuprolog`
  `name:` 2p-*MODULE_NAME*
  `version:` a SemVer-compliant string

[2] https://www.npmjs.com/org/tuprolog

Kotlin–JavaScript interoperability I

Required readings

- https://kotlinlang.org/docs/js-to-kotlin-interop.html
- https://kotlinlang.org/docs/js-interop.html

Kotlin–JavaScript interoperability II

Kotlin to JavaScript in a nutshell

- no overloading → members name are mangled[3] by default
  - unless `@JsName` annotation is used
- type-system & Kotlin's common library are re-implemented in JS
- primitive types are cumbersome
  - integer & float types are collapsed
  - longs are re-implemented
  - Kotlin's array → JS arrays
  - Kotlin's function → JS functions (there including lambdas)
- packages, interfaces, and classes are emulated via JS objects
  - but they correctly reflect the Kotlin codebase
- companion objects → `Companion` constants
- `varargs` → JS arrays

Kotlin–JavaScript interoperability III

2P-Kt Conventions

- `@JsName` annotation on all public methods/properties
  - → the JavaScript API is slightly different

[3] https://en.wikipedia.org/wiki/Name_mangling

Focus on...

Using 2P-Kt as an application – Overview

Graphical User Interface (GUI)

:ide JavaFX-based, JVM-specific GUI for Prolog

- pluggable `Solvers`

Playground Web-based, JS-specific GUI for Prolog

Command-Line Interface (CLI)

:cli Kotlin-based, multi-platform CLI for Prolog

Playground Web-based, JS-specific GUI

The *:ide* I

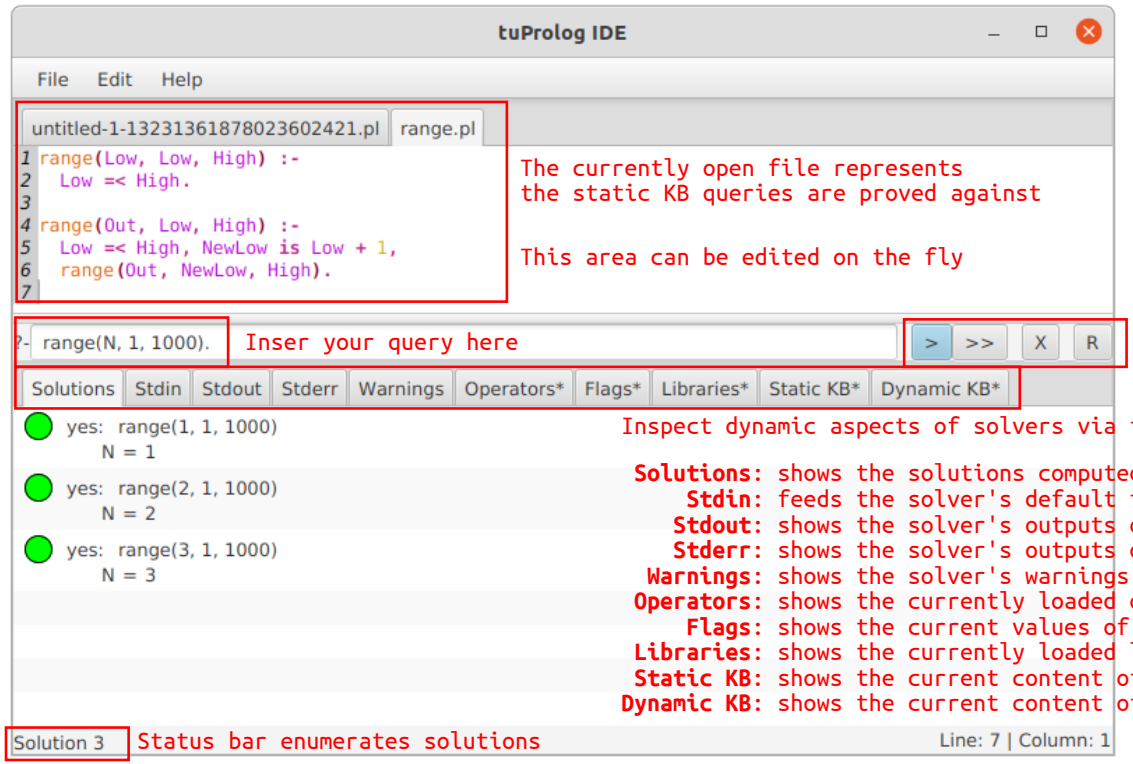
How to obtain & run the *:ide*

1. From `https://github.com/tuProlog/2p-kt/releases/latest`
   - download the `2p-ide-X.Y.Z-redist.jar` file
2. Start the `.jar` by either
   - double click on the file (depends on your system configuration)
   - run `java -jar /path/to/2p-ide-X.Y.Z-redist.jar` on your shell

! Requires Java 11+ to be properly installed and configured
   - should work on Win, Linux, and MacOS

The *:ide* II

How to use the IDE:



>: Start resolution and/or compute next solution
>>: Start resolution and/or compute ALL next solutions
X: Stop solutions enumeration
R: Resets the solver (erases outputs & warnings)

Inspect dynamic aspects of solvers via these tabs:

Solutions: shows the solutions computed so far
Stdin: feeds the solver's default input channel
Stdout: shows the solver's outputs on the default output channel
Stderr: shows the solver's outputs on the default error channel
Warnings: shows the solver's warnings
Operators: shows the currently loaded operators
Flags: shows the current values of flags
Libraries: shows the currently loaded libraries and their content
Static KB: shows the current content of the static KB
Dynamic KB: shows the current content of the dynamic KB

The Playground I

How to obtain & run the Playground

1. Browse to `https://pika-lab.gitlab.io/tuprolog/2p-kt-web/`
   - that's it

! **The Playground is currently unstable**
   - it is just a proof of concept

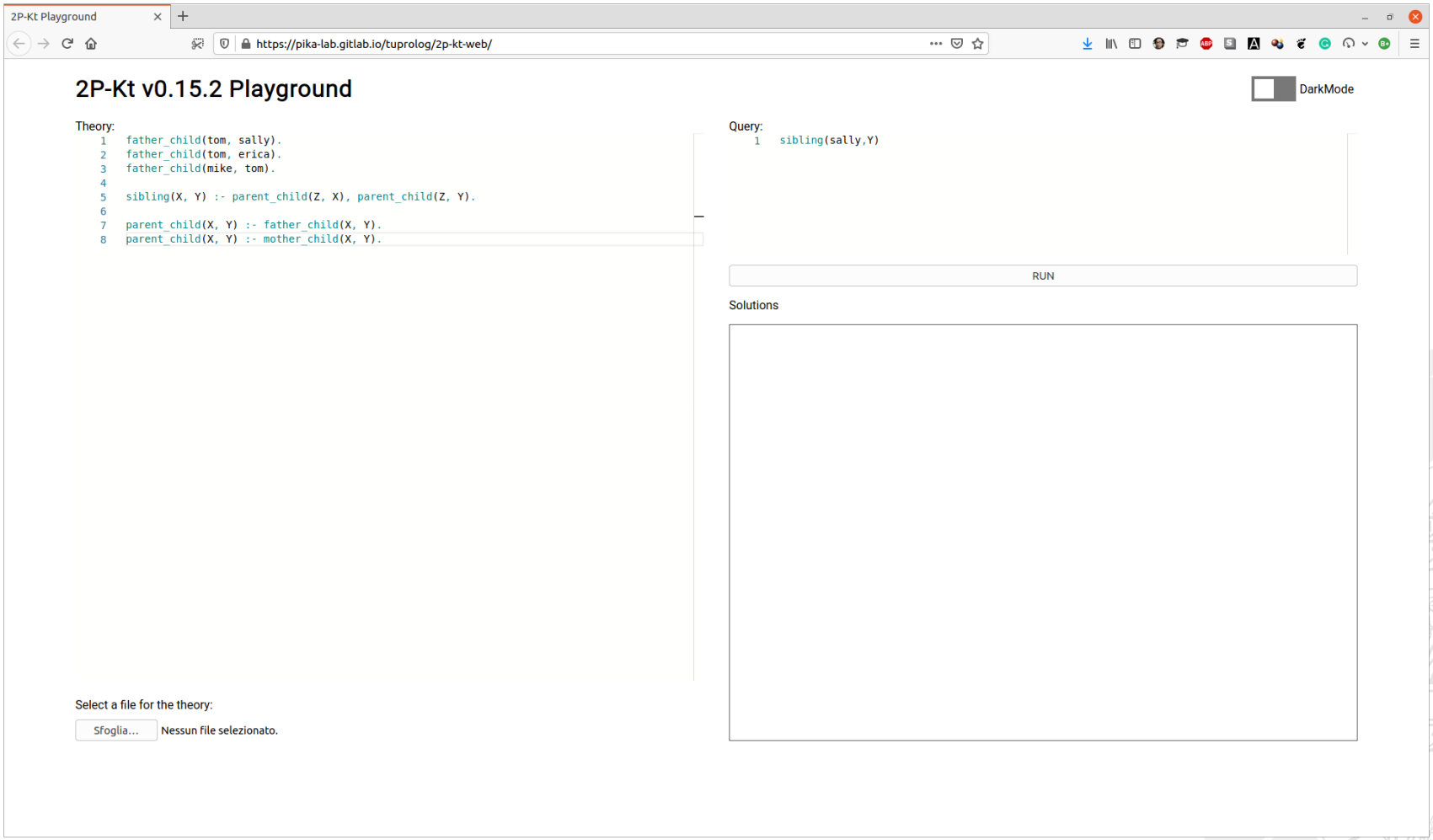
Potential

Even if it is currently unstable, our Playground:

- aims at demonstrating the feasibility of using 2P-Kt in-browser
- aims to become a valuable alternative to SWISH Prolog
  - where logic computation occurs on the browser-side
    - no sandbox required
    - no Internet connection required (after page load)

The Playground II

Preview:



Usage of the CLI I

How to obtain & run the *:repl*

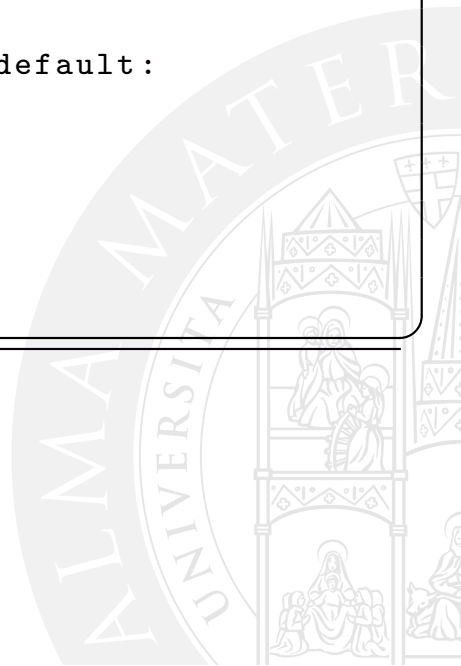
1. From `https://github.com/tuProlog/2p-kt/releases/latest`
   - download the `2p-repl-X.Y.Z-redist.jar` file
2. Start the `.jar` by either
   - double click on the file (depends on your system configuration)
   - run `java -jar /path/to/2p-repl-X.Y.Z-redist.jar` on your shell

! Requires Java 11+ to be properly installed and configured
   - should work on Win, Linux, and MacOS

Usage of the CLI II

Run `java -jar 2p-repl-X.Y.Z-redist.jar --help` to get:

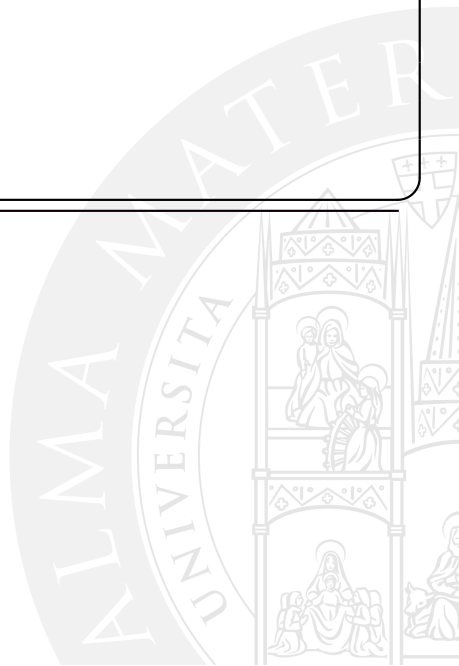
```
Usage: java -jar 2p-repl.jar [OPTIONS] COMMAND [ARGS]...

  Start a Prolog Read-Eval-Print loop

Options:
  -T, --theory TEXT  Path of theory file to be loaded
  -t, --timeout INT  Maximum amount of time for computing a solution (default:
                     1000 ms)
  --oop              Loads the OOP library
  -h, --help         Show this message and exit

Commands:
  solve  Compute a particular query and then terminate
```



Usage of the CLI III

Run `java -jar 2p-repl-X.Y.Z-redist.jar solve --help` to get:

```
Usage: java -jar 2p-repl.jar solve [OPTIONS] QUERY

  Compute a particular query and then terminate

Options:
  -n, --numberOfSolutions INT  Number of solution to calculate
  -h, --help                   Show this message and exit
```



Usage of the CLI IV

Usage example:

2P-Kt: A Kotlin Multi-Platform ecosystem for Symbolic AI

Giovanni Ciatto[1] Lorenzo Rizzato[2]

[1] Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna
giovanni.ciatto@unibo.it

[2] lorenzo.rizzato@studio.unibo.it

May 2021

References I

[1] ISO/IEC JTC 1/SC 22 Technical Committee.
ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core.
International Standard ISO/IEC 13211-1, ISO/IEC, 1995.

[2] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini.
Logic programming as a service.
Theory and Practice of Logic Programming, 18(5-6):846–873, September 2018.
Special Issue "Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming".

[3] Mats Carlsson.
On implementing prolog in functional programming.
New Generation Computing, 2(4):347–359, 1984.

References II

[4] Giovanni Ciatto, Roberta Calegari, Enrico Siboni, Enrico Denti, and Andrea Omicini.

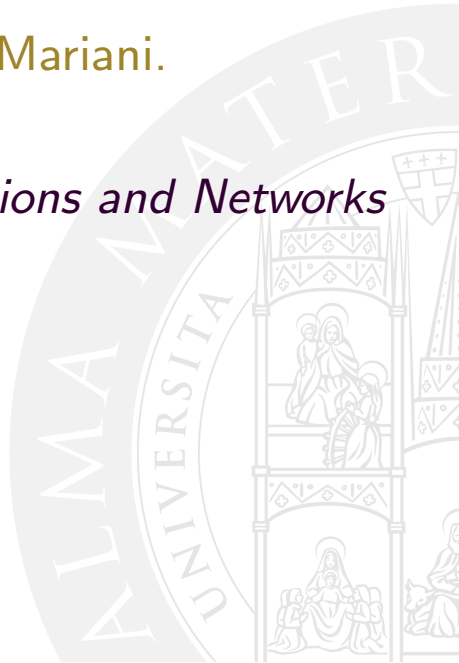
2P-Kt: logic programming with objects & functions in kotlin.

In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *WOA 2020 – 21th Workshop "From Objects to Agents"*, volume 2706 of *CEUR Workshop Proceedings*, pages 219–236, Aachen, Germany, October 2020. Sun SITE Central Europe, RWTH Aachen University.

[5] Giovanni Ciatto, Lorenzo Rizzato, Andrea Omicini, and Stefano Mariani.

TuSoW: Tuple spaces for edge computing.

In *The 28th International Conference on Computer Communications and Networks (ICCCN 2019)*, Valencia, Spain, 29 July–1 August 2019. IEEE.



References III

[6] Keith L. Clark.

Negation as failure.

In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[7] Enrico Denti, Andrea Omicini, and Alessandro Ricci.

tuProlog: A light-weight Prolog for Internet applications and infrastructures.

In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001.

3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[8] Enrico Denti, Andrea Omicini, and Alessandro Ricci.

Multi-paradigm Java-Prolog integration in tuProlog.

Science of Computer Programming, 57(2):217–250, August 2005.

References IV

[9] Philipp Körner, Michael Beuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto.

A multi-walk through the past, present and future of prolog.

(Submitted to) Theory and Practice of Logic Programming (TPLP), 2021.

[10] Stefano Mariani and Andrea Omicini.

Molecules of Knowledge: Self-organisation in knowledge-intensive environments.

In Giancarlo Fortino, Costin Bădică, Michele Malgeri, and Rainer Unland, editors, *Intelligent Distributed Computing VI*, volume 446 of *Studies in Computational Intelligence*, pages 17–22. Springer, 2013.

6th International Symposium on Intelligent Distributed Computing (IDC 2012), Calabria, Italy, 24-26 September 2012. Proceedings.

[11] Alberto Martelli and Ugo Montanari.

An efficient unification algorithm.

ACM Trans. Program. Lang. Syst., 4(2):258–282, April 1982.

References V

[12] Andrea Omicini and Enrico Denti.
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294, November 2001.

[13] Andrea Omicini and Franco Zambonelli.
Coordination for Internet application development.
Autonomous Agents and Multi-Agent Systems, 2(3):251–269, September 1999.
Special Issue: Coordination Mechanisms for Web Agents.

[14] Terence Parr.
The Definitive ANTLR 4 Reference.
Pragmatic Bookshelf, 2nd edition, 2013.

References VI

[15] Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci.
The architecture and design of a malleable object-oriented Prolog engine.
In Roger L. Wainwright, Hisham M. Haddad, Ronaldo Menezes, and Mirko Viroli, editors, *23rd ACM Symposium on Applied Computing (SAC 2008)*, volume 1, pages 191–197, Fortaleza, Ceará, Brazil, 16–20 March 2008. ACM.
Special Track on Programming Languages.

[16] Giulio Piancastelli, Andrea Omicini, and Enrico Denti.
Towards a logic framework for Web programming.
Intelligenza Artificiale, 5(1):151–155, 2011.

References VII

[17] Giuseppe Pisano, Roberta Calegari, Andrea Omicini, and Giovanni Sartor.
Arg-tuProlog: A tuProlog-based argumentation framework.
In Francesco Calimeri, Simona Perri, and Ester Zumpano, editors, *CILC 2020 – Italian Conference on Computational Logic. Proceedings of the 35th Italian Conference on Computational Logic*, volume 2719 of *CEUR Workshop Proceedings*, pages 51–66, Aachen, Germany, 13-15 October 2020. Sun SITE Central Europe, RWTH Aachen University, CEUR-WS.

[18] J. A. Robinson.
A machine-oriented logic based on the resolution principle.
J. ACM, 12(1):23–41, January 1965.

[19] David H. D. Warren.
An abstract prolog instruction set.
Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.