functional scala 2024
Presents

Recognizing Regular Patterns in Heterogeneous Sequences

Jim E. Newton

EPITA Research Laboratory, Le Kremlin Bicêtre, FRANCE

December 5, 2024

Sponsored by

GOLEM

ZIVERGE

Overview

History

Publication History

- Regular Type Expressions (RTEs) introduced at ELS 2016 (European Lisp Symposium), implemented in Common Lisp.
  `https://www.lrde.epita.fr/wiki/Publications/newton.16.els`
- See PhD Thesis 2018 for theoretical, implementation, and performance details
  `https://www.lrde.epita.fr/wiki/Publications/newton.18.phd`
- RTE Available:

Scala	`https://github.com/jimka2001/scala-rte`
Clojure	`https://github.com/jimka2001/clojure-rte`
Python	`https://github.com/jimka2001/python-rte`
Common Lisp	`https://github.com/jimka2001/cl-rte`

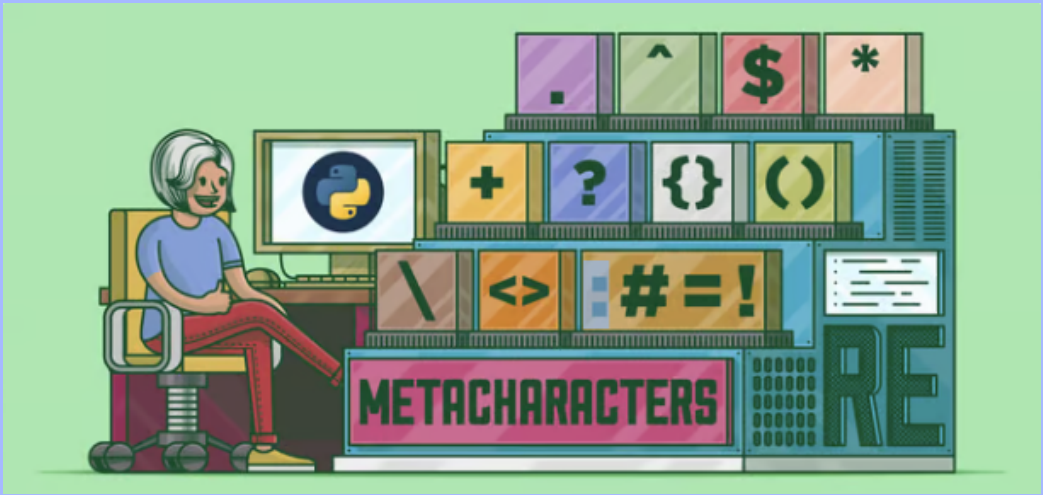
- PADL 2025, Practical Aspects of Declarative Languages:
  Type-Checking Heterogeneous Sequences in a Simple Embeddable Type System

Goal

- Efficiently recognize *regular patterns* in mixed-type sequences.
- Supported in Scala as `Seq[Any]`.

Example

Regular Type Expressions (RTEs)



Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

[1, 2, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

[1, 2, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

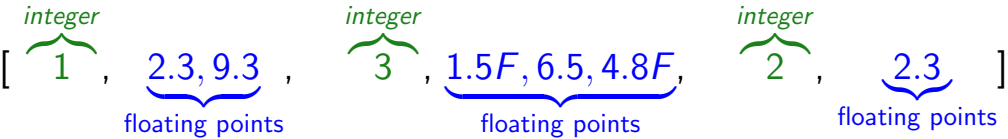
What's the pattern?

Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

[1, 2, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

What's the pattern?

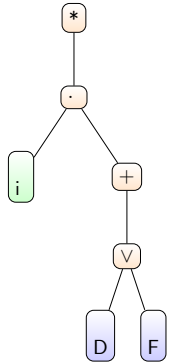


Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

String-based regular expressions

- Match strings like: `"iDDiFDFiD"`,
- ... we use surface syntax: `"(i(D|F)+)*"`.
- ... representing expression: $(i \cdot (D \vee F)^{+})^{*}$,



Abstract Syntax Tree (AST)

Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

[1, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

String-based regular expressions

- Match strings like: `"iDDiFDFiD"`,
- ... we use surface syntax: `"(i(D|F)+)*"`.
- ... representing expression: $(i \cdot (D \vee F)^+)^*$,

We propose Rational Type Expressions (RTEs)

- Rational type expression: (Int $\cdot$ (Double $\cup$ Float)$^+$)*
- *Challenge №1:* We need a surface syntax (and AST) for Scala.

Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

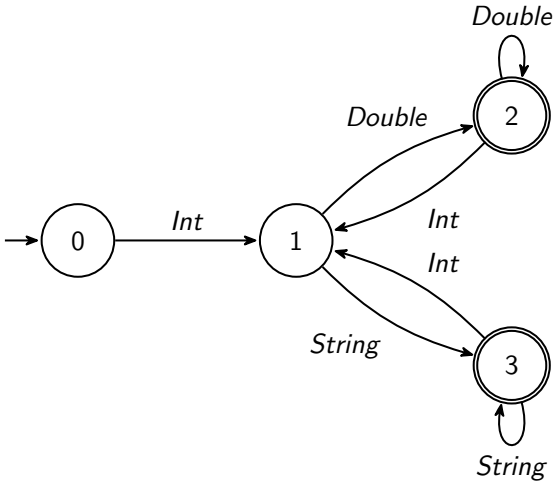
Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

Does the sequence match the pattern?
$(Int \cdot (Double^+ \ \vee String^+))^+$

We construct a finite automaton (DFA).

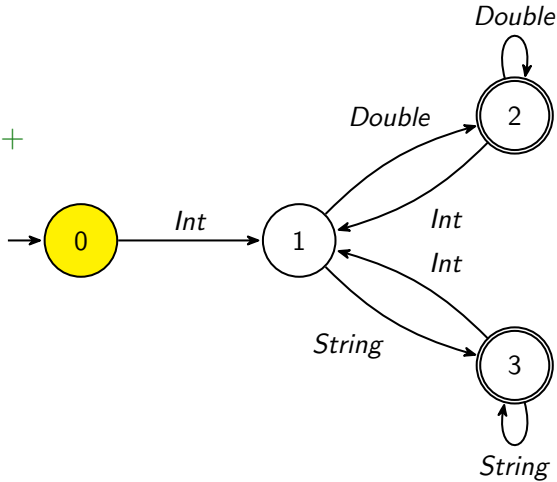
Challenge #2: How to construct a finite automaton from an RTE?



Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \quad \vee String^{+}))^{+}$$

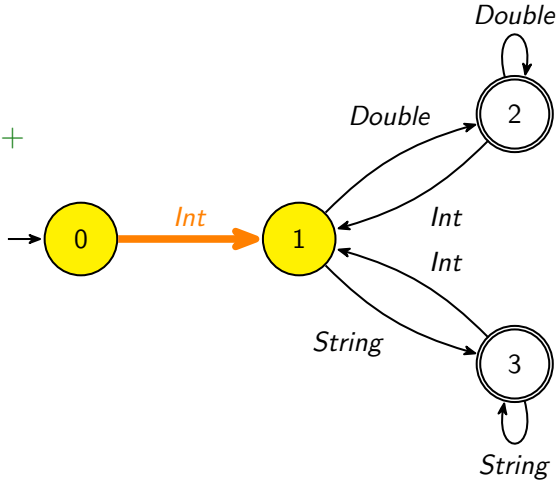


Does it match?

Example 1: How does a pattern predicate work?

`[ 13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \quad \vee String^{+}))^{+}$

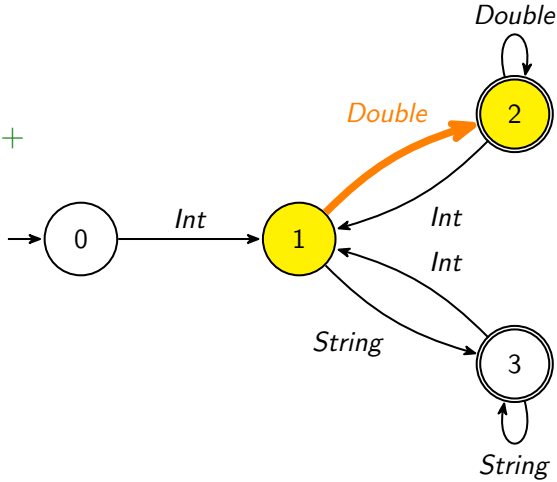


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^+ \vee String^+))^+$

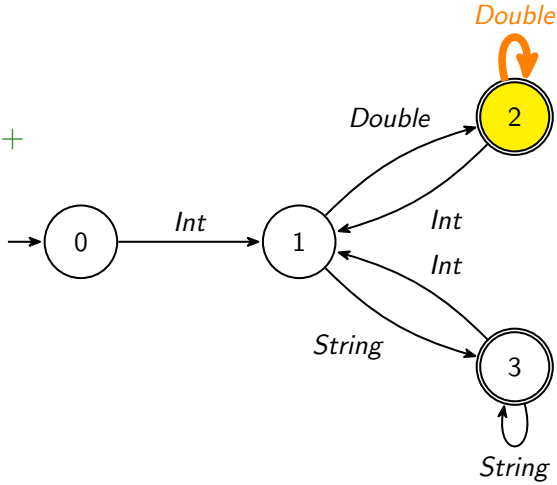


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

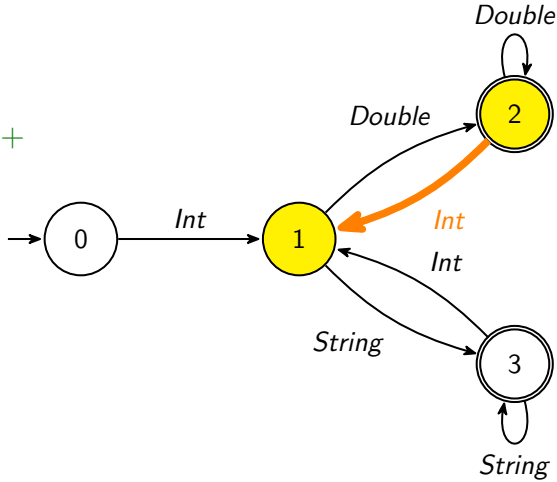


Does it match?

Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$(Int \cdot (Double^+ \quad \vee String^+))^+$

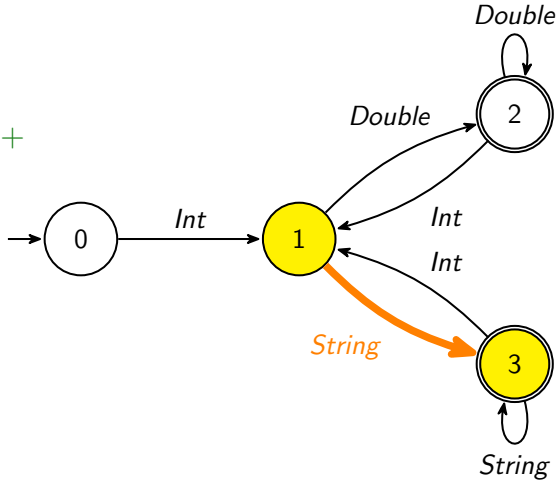


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

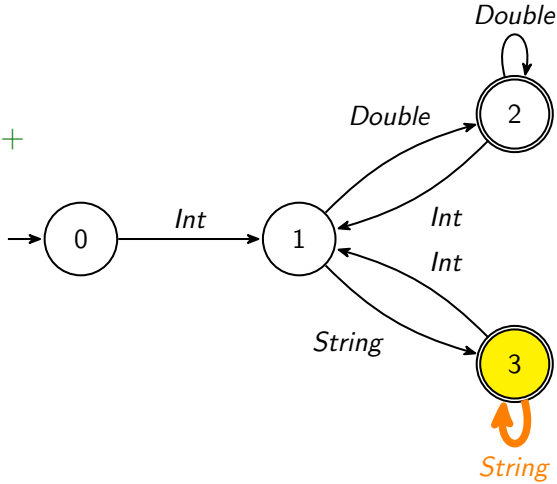


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

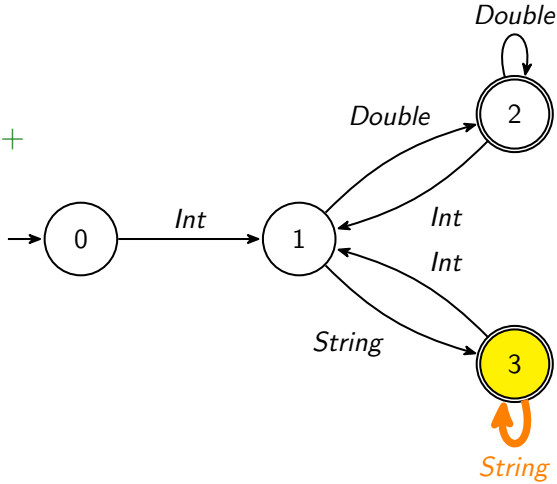


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

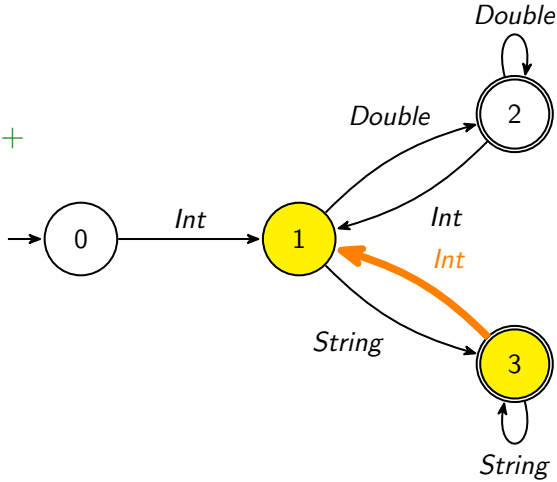


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \quad \vee String^{+}))^{+}$

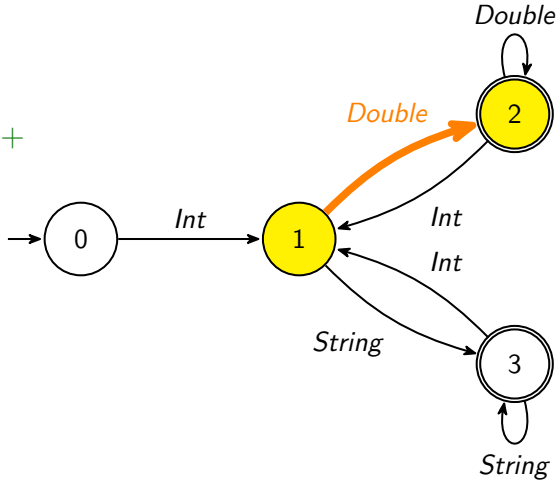


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \quad \vee String^{+}))^{+}$

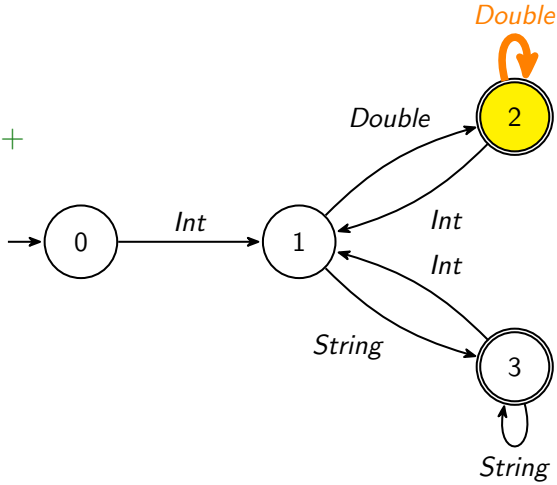


Does it match?

Example 1: How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

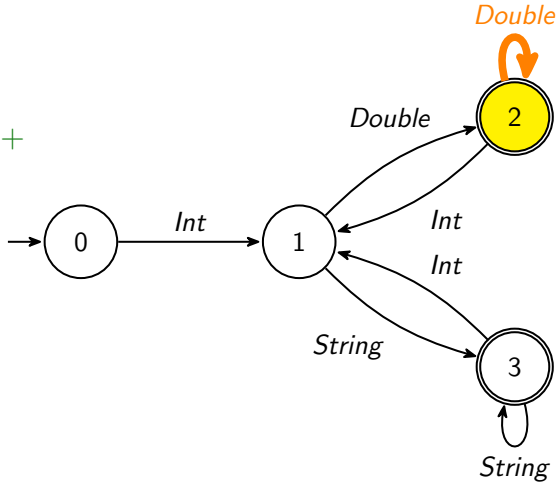


Does it match?

Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$

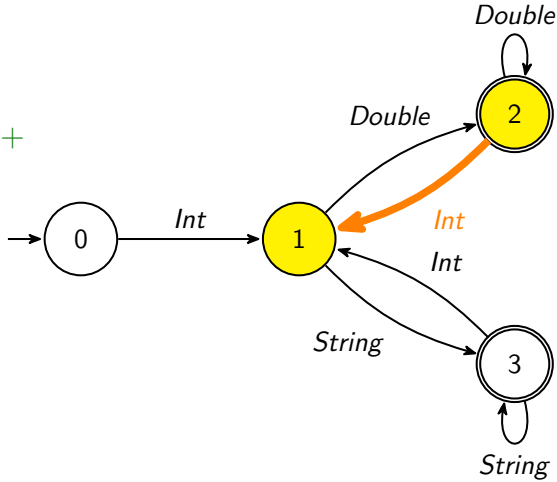


Does it match?

Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$(Int \cdot (Double^+ \vee String^+))^+$

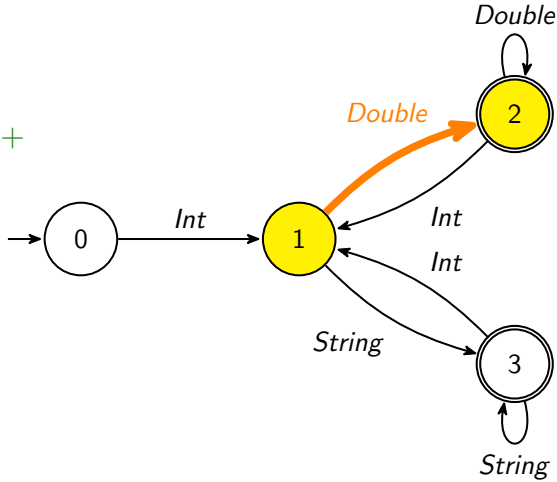


Does it match?

Example 1: How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0 ]
```

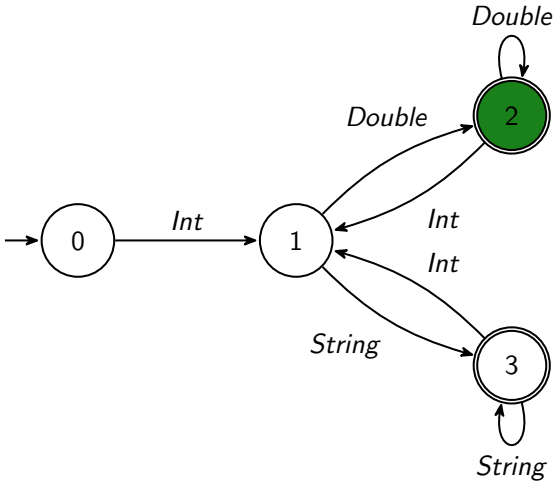
$(Int \cdot (Double^{+} \quad \vee String^{+}))^{+}$



Does it match?

Example 1: How does a pattern predicate work?

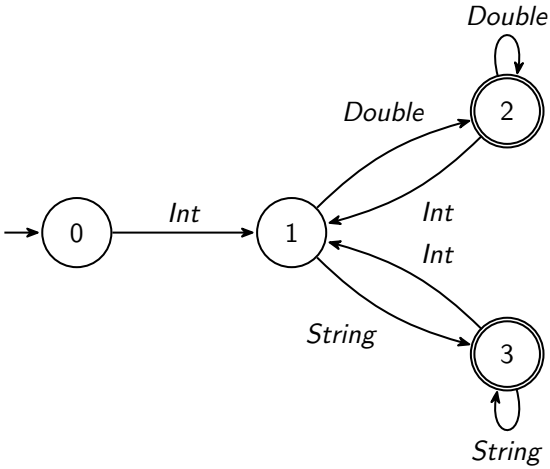
```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```



Yes, it's a match!

Example 1: How does a pattern predicate work?

Decision procedure is $O(n)$, *independent of syntactical complexity* of the RTE.

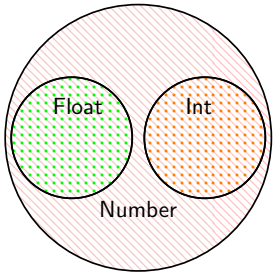


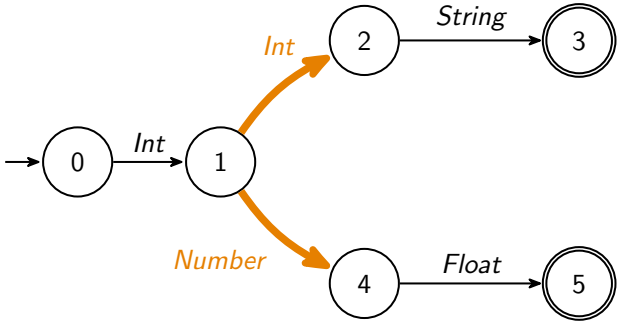
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



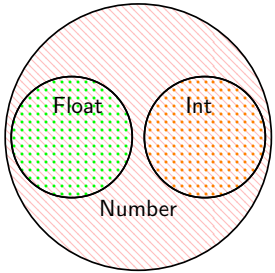


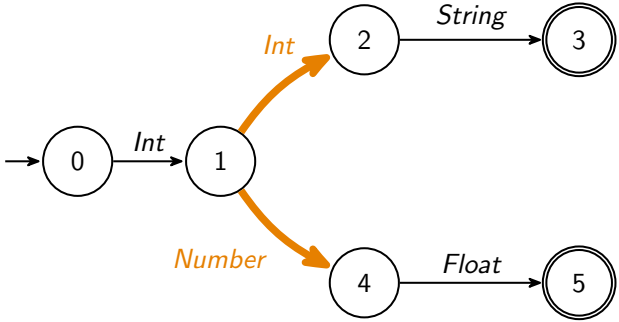
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





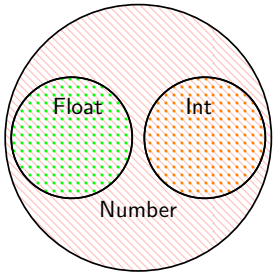
Backtracking?

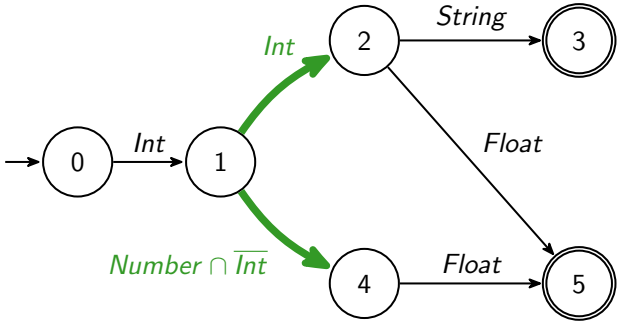
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



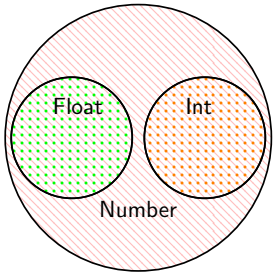


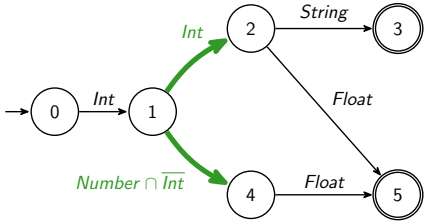
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





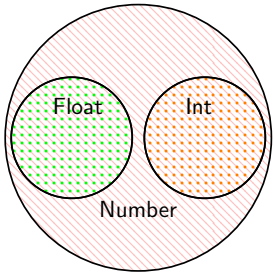
Challenge №3: How to support $Int \cap \overline{Number}$ in Scala?

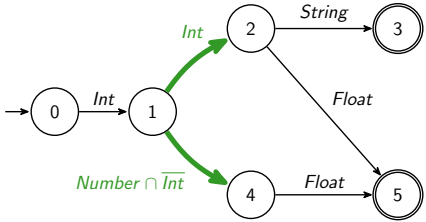
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





Challenge №3: How to support $Int \cap \overline{Number}$ in Scala?

Challenge №4: How to partition types?

E.g., type decomposition

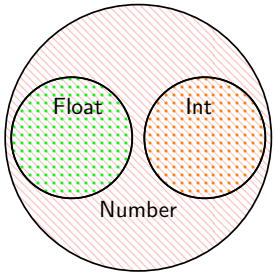
$\{String, Int, Number\} \rightarrow \{String, Int, Number \cap \overline{Int}\}$

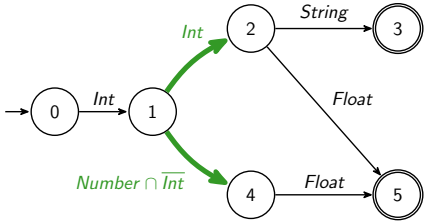
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





Challenge №3: How to support $Int \cap \overline{Number}$ in Scala?

Challenge №4: How to partition types?

Challenge №5: How to avoid duplicate type checks?

Challenges of the Project

- *Challenge № 1:* RTE Representation: Representing an RTE in Scala?
- *Challenge № 2:* DFA Construction: Constructing from RTE?
- *Challenge № 3:* Type Lattice: Union, intersection, complement types?
- *Challenge № 4:* Determinism: Type partitioning?
- *Challenge № 5:* Efficiency: Avoiding redundant type checks at run-time?

Challenge №1: RTE Representation

How to represent an RTE in Scala?

- Surface syntax: declarative, expressive, composable
- Programmatic interface: reflective, algebraic manipulation

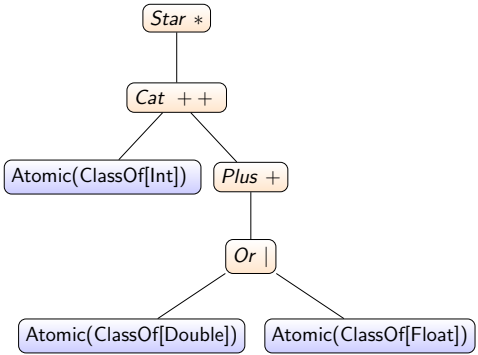
RTEs

What are Regular Type Expressions?

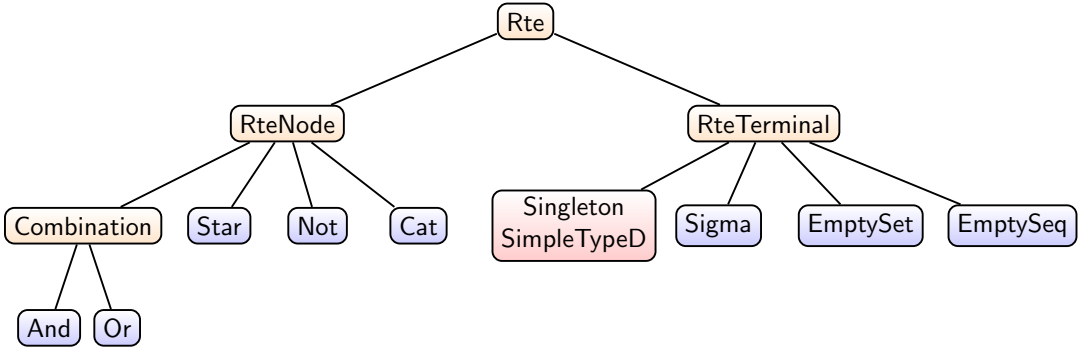
- Mathematical notation:
  $(Int \cdot (Double \cup Float)^{+})^{*}$
- Scala notation: AST

```
1  val I:Rte = Atomic(classOf[Int])
2  val F:Rte = Atomic(classOf[Float])
3  val D:Rte = Atomic(classOf[Double])
4
5  val re:Rte = (I ++ (D | F).+ ).*
```

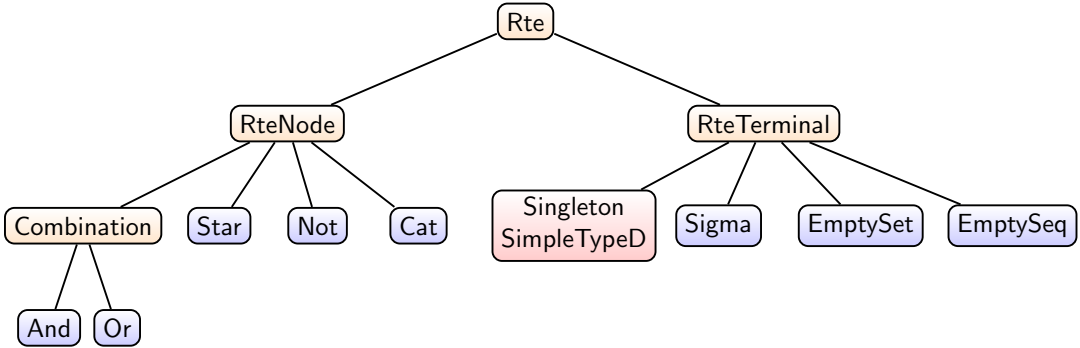
- Leaf nodes interface to Scala Type System



`Rte` class quasi-ADT, Algebraic Data Type



`Rte` class quasi-ADT, Algebraic Data Type



More on `SimpleTypeD` later ...

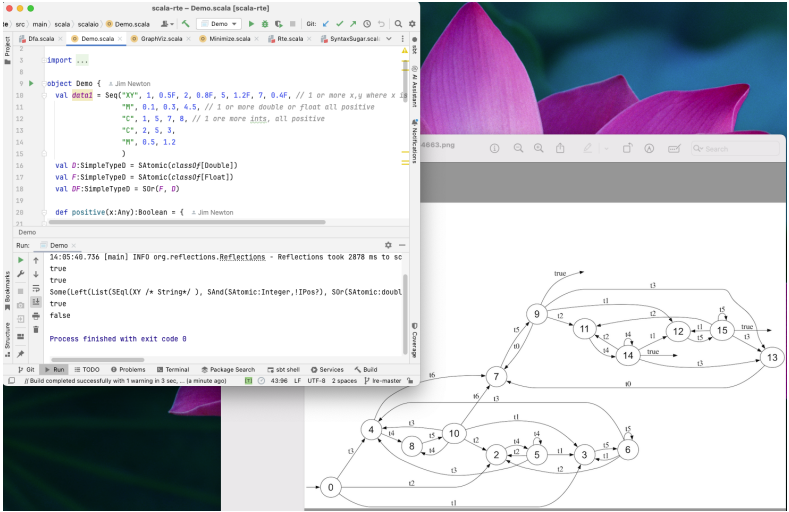
Challenge №2: DFA Construction

Given an RTE, generate a finite automaton.

- Well-known techniques exists to construct DFAs from RE
  - Brzozowski Derivative
- Adapt them to work with RTEs
- Enforce determinism

Demo

Sample Flow



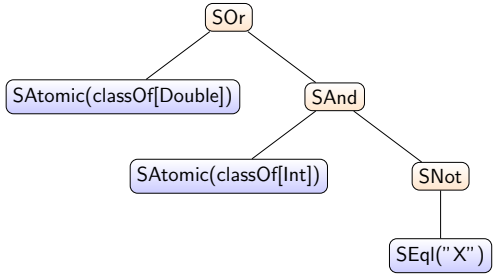
Challenge №3: Type Lattice

How to support types like $Int \cap \overline{Number}$ in Scala?

- Support *complemented type lattice*
- Embed a dynamic type system into an existing programming language.
- Answer type membership and subtype questions.

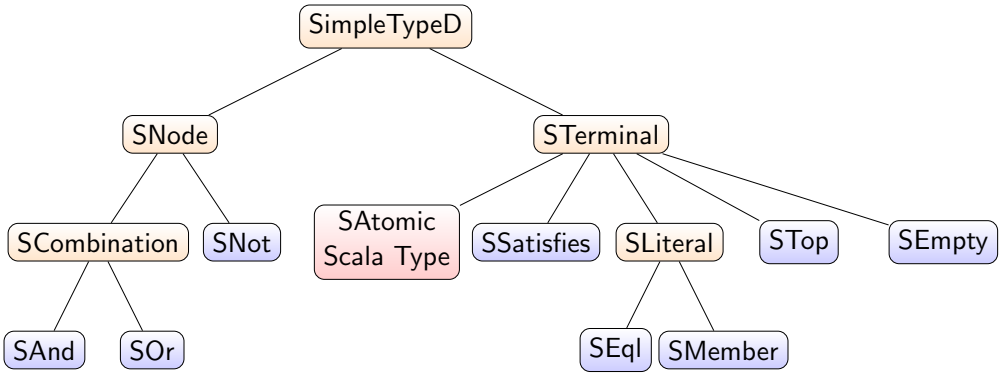
Example `SimpleTypeD` Expression Tree: AST

```
1 val Int     = SAtomic(classOf[Int])
2 val Double = SAtomic(classOf[Double])
3 val td:SimpleTypeD = SOr(Double, SAnd(Int, SNot(SEql("X"))))
```



- A type designator is an expression tree (AST).
- Leaf nodes interface to Scala classes via `SAtomic(...)`.
- ... and to literal Scala values via `SEql(...)`
- ... and to predicate functions via `SSatisfies(...)`

`SimpleTypeD` class quasi-ADT



Type Membership Predicate

Boolean type membership question is *always answerable*.

```
SAtomic(classOf[Int]).typep(-42) // returns true

// returns true
(SAtomic(classOf[String]) || SAtomic(classOf[Int])).typep(7)


// define predicate
def oddp(a:Any):Boolean = {
  a match
    case a:Int => a % 2 != 0
    case _ => false
}

SSatisfies(oddp).typep(36) // returns false
```

Subtype Predicate

Semi-Boolean Subtype predicate *sometimes unanswerable*.

```
val Str:SimpleTypeD = SAtomic(classOf[String])
val Int:SimpleTypeD = SAtomic(classOf[Int])
val Num:SimpleTypeD = SAtomic(classOf[Number])
val odd:SimpleTypeD = SSatisfies(oddp, "oddp")

Str.subtypep(Int) // returns Some(false)
Int.subtypep(Num) // returns Some(true)
SSatisfies(oddp).subtypep(Int) // returns None
```

Unanswerable because:

- Impossible to compute, *e.g.* `SSatisfies`.
- Code is incomplete.
- JVM supports run-time loaded classes.
- Limitations of support libraries (discussed later).

Simple Embedded Type System: SETS

At the point, *what have we done*?

- Wrapped the Scala type system
- ... with a simple type system
- ... which supports a complemented type lattice
- ... with membership Boolean predicate
- ... with subtype semi-Boolean predicate
- ... which supports reflection

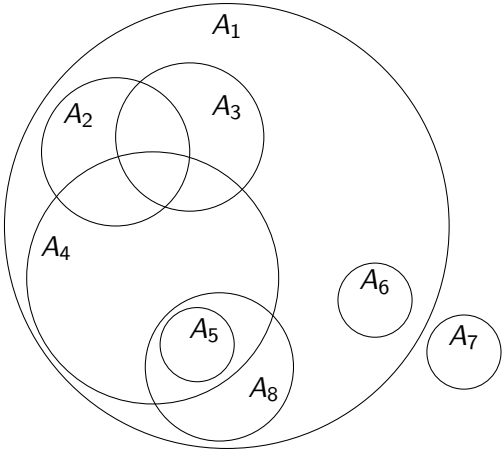
Cf: *A Portable, Simple, Embeddable Type System* [Newton, Pommellet] 2021 ELS

Challenge №4: Deterministic State Machines

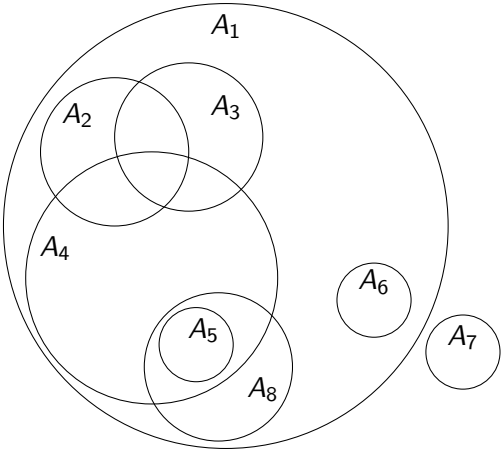
How to assure DFAs are deterministic by construction?

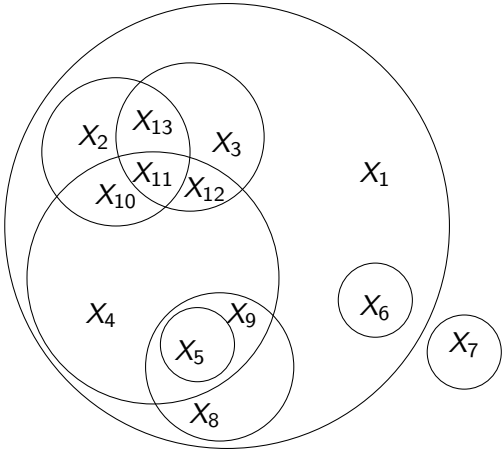
- Compute a partition of a given set of type designators,
- ... even (especially) when subtype relation is unknown.

MDTD: Maximal Disjoint Type Decomposition

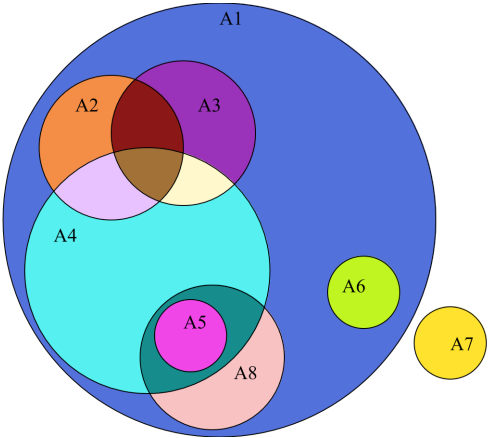


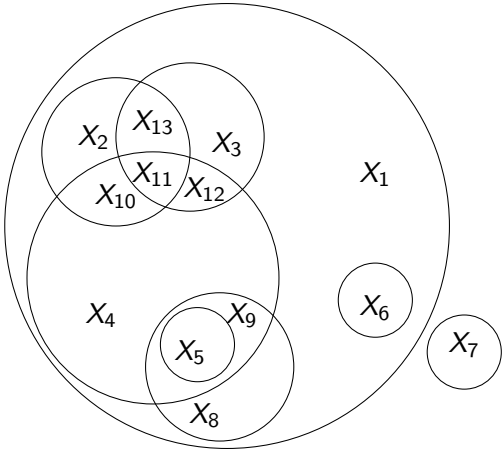
MDTD: Maximal Disjoint Type Decomposition



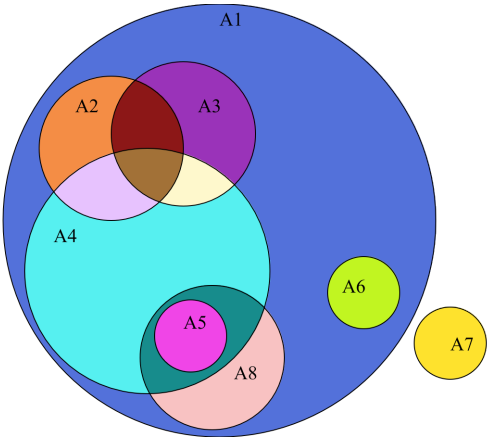


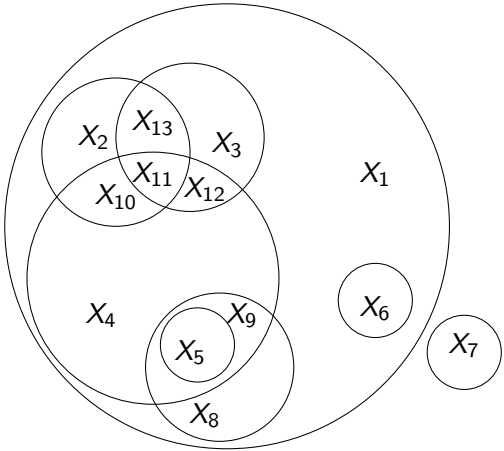
MDTD: Maximal Disjoint Type Decomposition





MDTD: Maximal Disjoint Type Decomposition



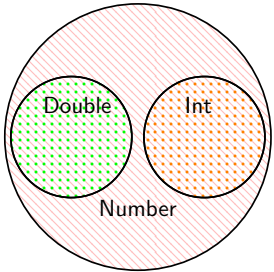


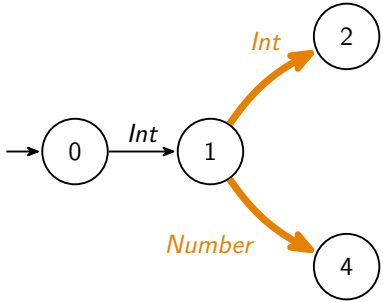
Cf: *An Elegant and Fast Algorithm for Partitioning Types* [Newton] 2023 ELS

Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$

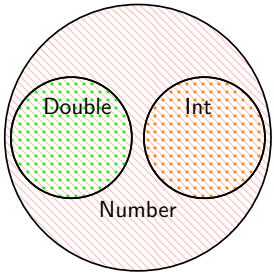


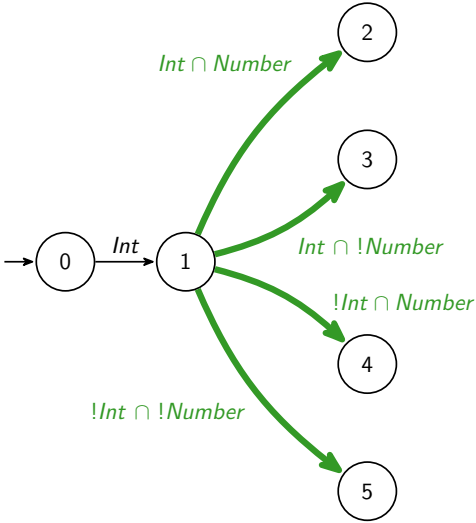


Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$

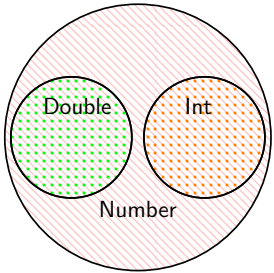


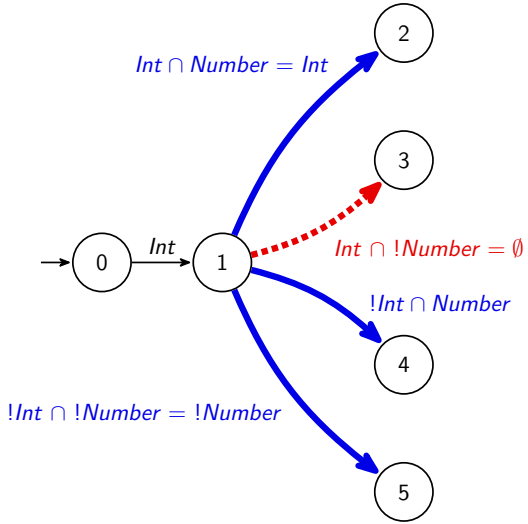


Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



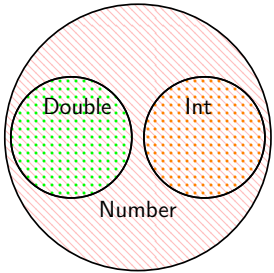


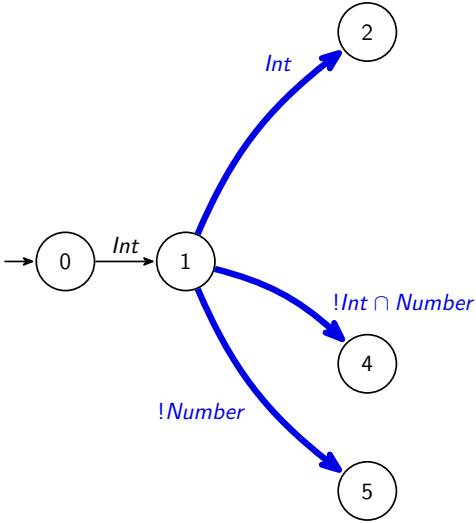
We can *decide* that state ③ is unreachable.

Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



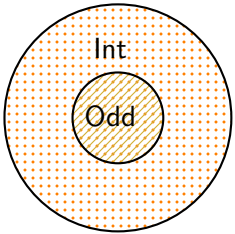


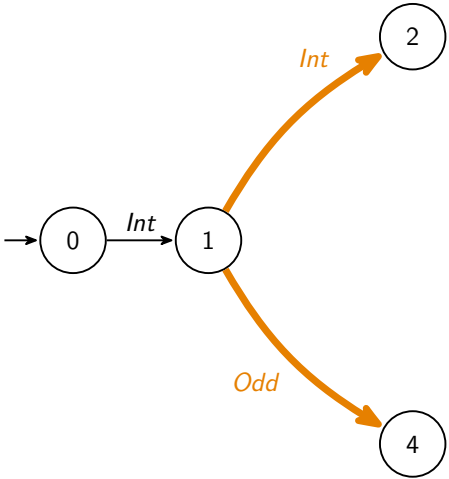
We can *decide* that state ③ is unreachable.

Non-determinism by `SSatisfies`

$Odd \subseteq Int$ unknown

$Odd \cap !Int = \emptyset$ unknown

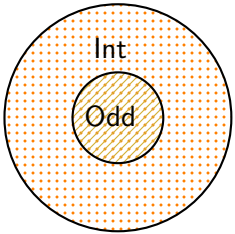


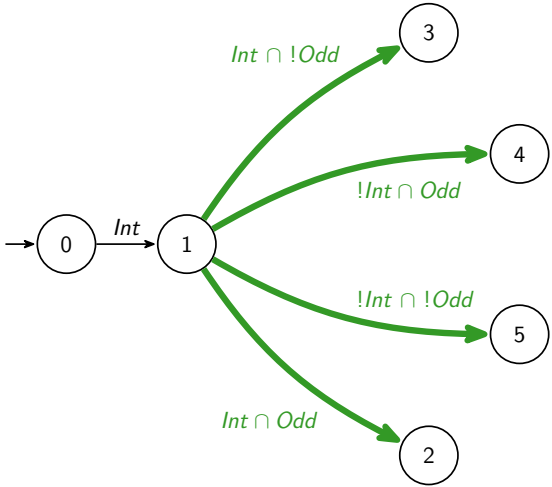


Non-determinism by `SSatisfies`

$Odd \subseteq Int$ unknown

$Odd \cap !Int = \emptyset$ unknown

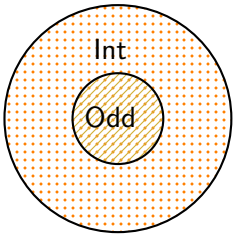


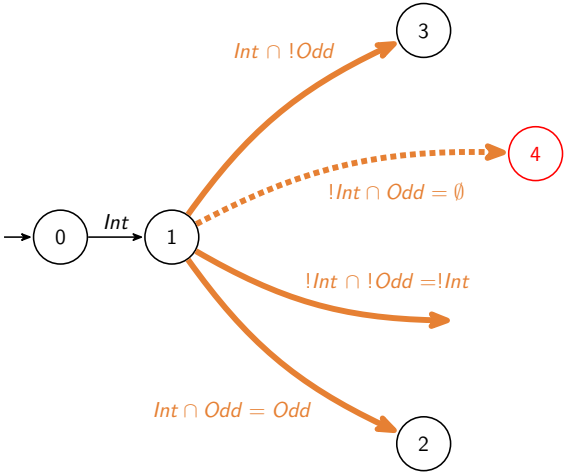


Non-determinism by SSatisfies

$Odd \subseteq Int$ unknown

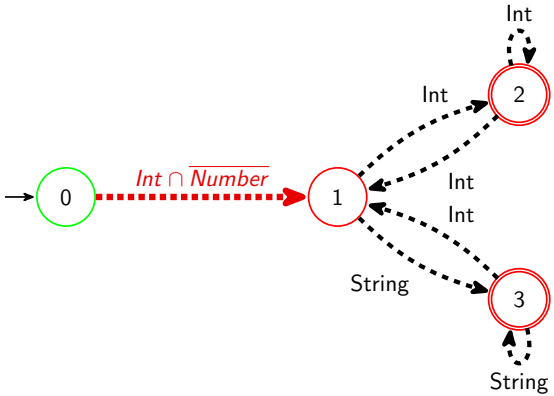
$Odd \cap !Int = \emptyset$ unknown





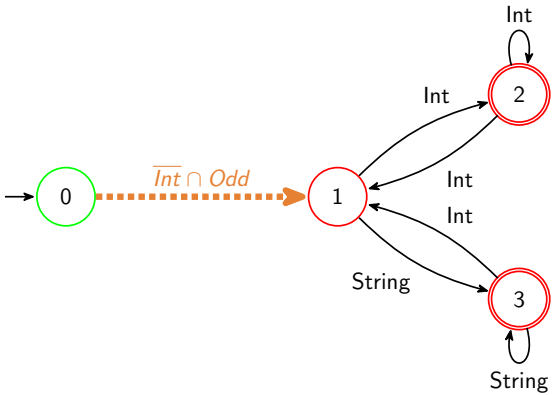
Unreachable state ④, but *undecidable*.

Unsatisfiable Transitions



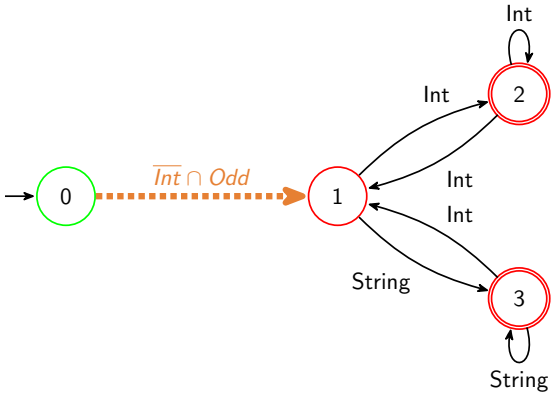
- If we determine a type is empty, then the transition is *unsatisfiable*.
- Thus we *can eliminate* the transition and unreachable states.

Indeterminate Transitions



- If we cannot determine a type is empty, the transition may *still be unsatisfiable*.
- However, we *cannot eliminate* the transition and unreachable states.

Indeterminate Transitions



- We *can always* determine *type membership*.
- DFAs with indeterminate transitions *correctly* match sequences in $O(n)$.

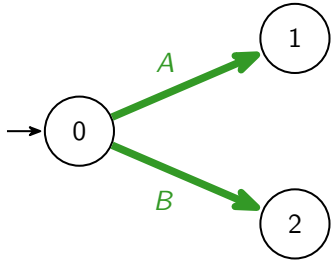
Determining whether two classes are disjoint

```
final class A() {}
final class B() {}

class C() {}
trait D() {}

abstract class E() {}
trait F {}

class G() extends E with F {}
```



$$A \cap B = \emptyset$$

Both are `final`; they have no common *inhabited* subclass.

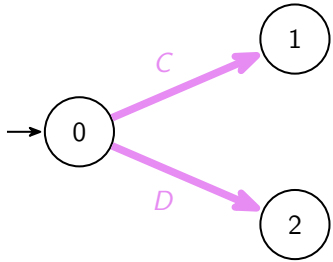
Determining whether two classes are disjoint

```
final class A() {}
final class B() {}

class C() {}
trait D() {}

abstract class E() {}
trait F {}

class G() extends E with F {}
```



$$C \cap D = \textit{unknown}$$

Is there a class *somewhere* which inherits from both?

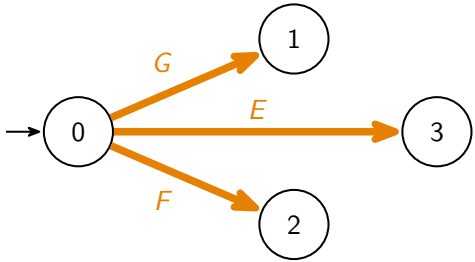
Determining whether two classes are disjoint

```
final class A() {}
final class B() {}

class C() {}
trait D() {}

abstract class E() {}
trait F {}

class G() extends E with F {}
```



$$G \subset E \implies G \cap E \neq \emptyset$$

Explicit subtype relation.

Determining whether two classes are disjoint

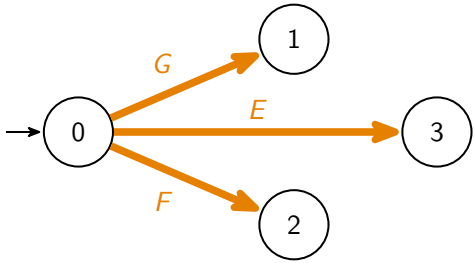
```
final class A() {}
final class B() {}

class C() {}
trait D() {}

abstract class E() {}
trait F {}

class G() extends E with F {}
```

G inherits from E and F.



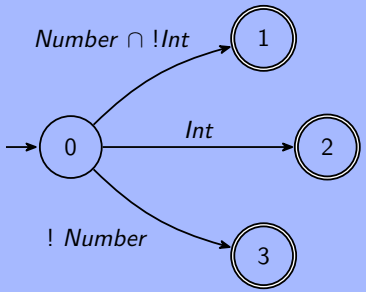
$$E \cap F \neq \emptyset$$

Determining whether two classes are disjoint: Caveats

- If *currently* no subclass exists, JVM might *later* run-time load `subclass.jar`
- With Java $> 8.x$, Scala cannot dependably compute list of subclasses.
- Library `github.com/ronmamo/reflections` no longer maintained.
  - `https://github.com/ronmamo/reflections/issues/324`
  - `https://users.scala-lang.org/t/help-with-org-reflections-api`
  - `https://stackoverflow.com/questions/52879584`

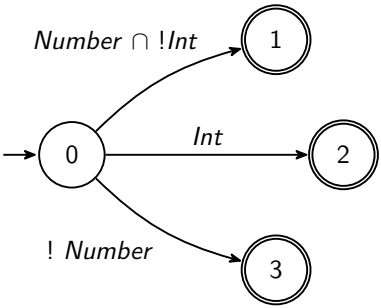
Challenge №5: Redundant Type Check

Select correct transition, avoiding *redundant type checks*.



Sequential Type Check

A DFA state may have several *disjoint* transitions, each with its own type label.

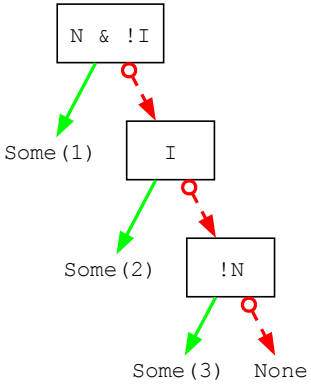



```
1  val N = SAtomic(classOf[Number])
2  val I = SAtomic(classOf[Int])
3
4  if (N & !I).typep(x)
5    Some(1)
6  else if I.typep(x)
7    Some(2)
8  else if (!N).typep(x)
9    Some(3)
10 else
11   None
```

Some types may be checked multiple times. We can rewrite the code to *eliminate redundant checks*.

Decision Tree Structure

We programmatically manipulate `if then ... else ...` using a lazy decision tree.

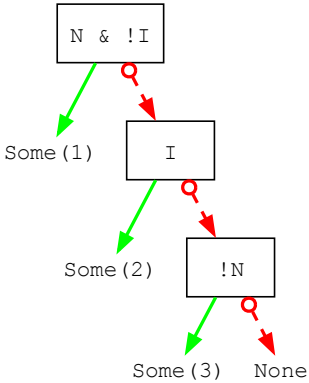


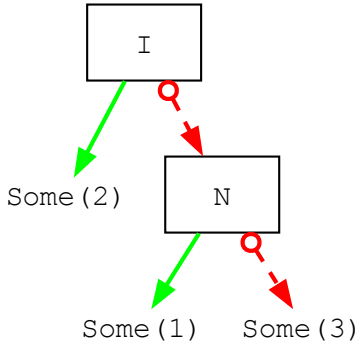

```
1  val N = SAtomic(classOf[Number])
2  val I = SAtomic(classOf[Int])
3
4  if (N & !I).typep(x)
5    Some(1)
6  else if I.typep(x)
7    Some(2)
8  else if (!N).typep(x)
9    Some(3)
10 else
11   None
```

Decision Tree, Before and After

Viewing the decision tree before/after

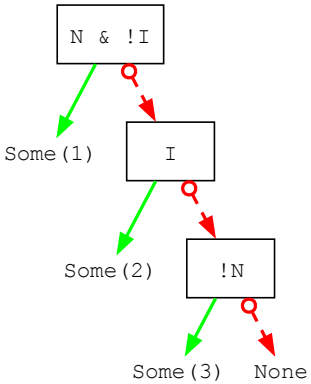
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$





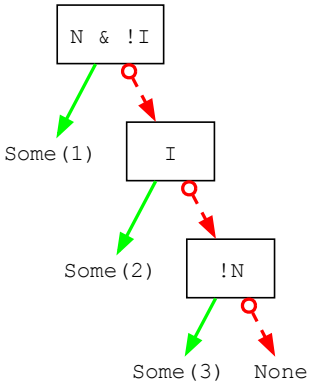
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

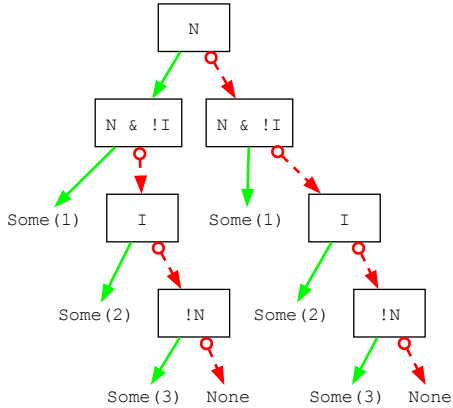
Duplicate tree, and introduce `if N.typep(x) then ... else ...`



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

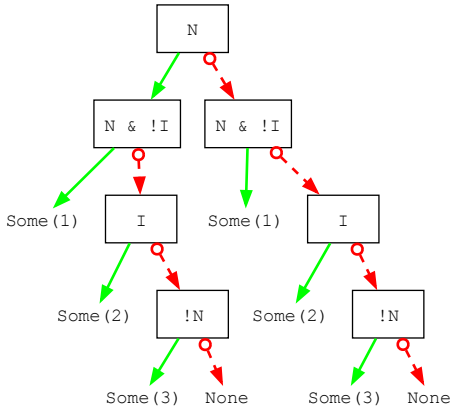
Duplicate tree, and introduce `if N.typep(x) then ... else ...`





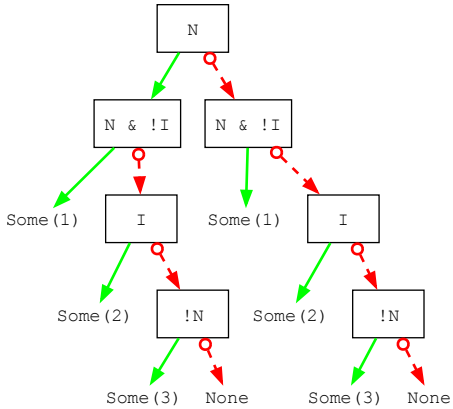
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

Duplicate tree, and introduce `if N.typep(x) then ... else ...`



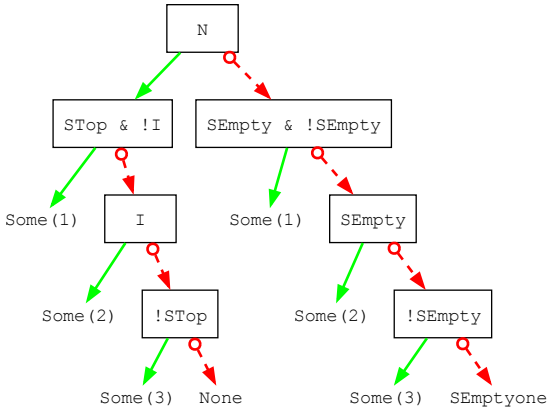
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then` part: Supertypes of `N` $\to$ `STop`. In `else` part: Subtypes of `N` $\to$ `SEmpty`.



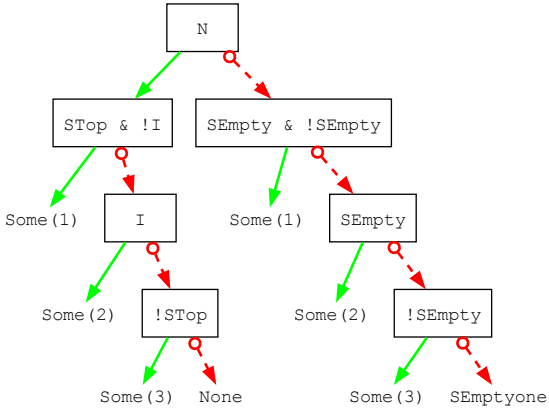
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then` part: Supertypes of `N` $\to$ `STop`. In `else` part: Subtypes of `N` $\to$ `SEmpty`.



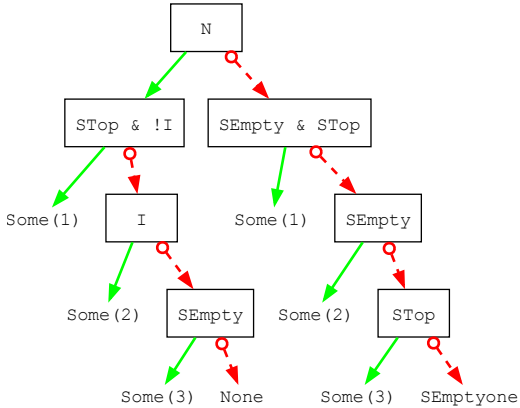
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

`!STop → SEmpty` `!SEmpty → STop`



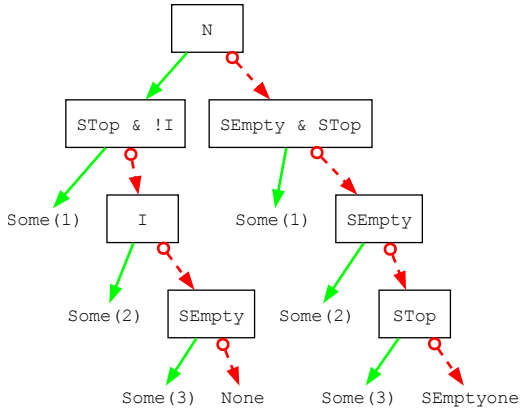
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

`!STop → SEmpty` `!SEmpty → STop`



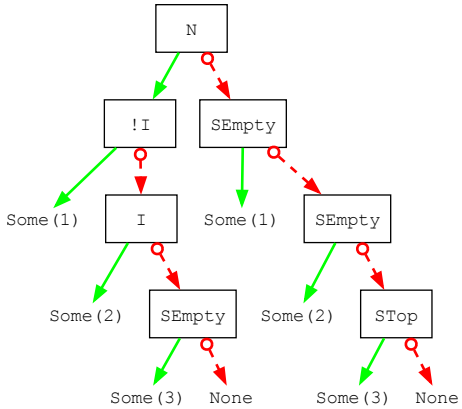
Rewrite: $1 \to 2 \to 3 \to \boxed{4} \to 5 \to 6 \to 7 \to 8 \to 9$

`(STop & x)` $\to$ `x` `(SEmpty & x)` $\to$ `SEmpty`



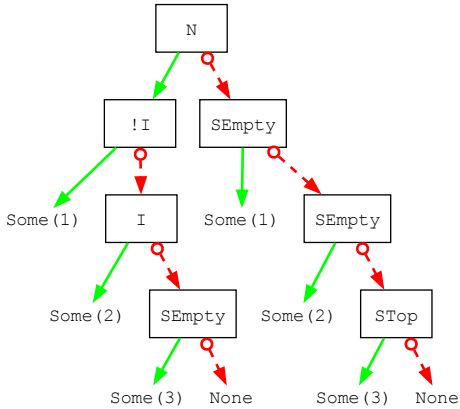
Rewrite: $1 \to 2 \to 3 \to \boxed{4} \to 5 \to 6 \to 7 \to 8 \to 9$

`(STop & x)` $\to$ `x` `(SEmpty & x)` $\to$ `SEmpty`



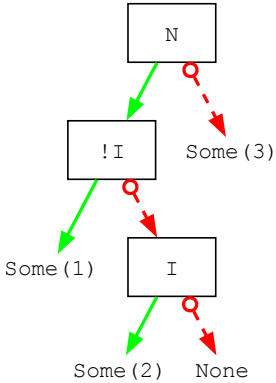
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

Replace `STop` with `then` branch. Replace `SEmpty` with `else` branch.



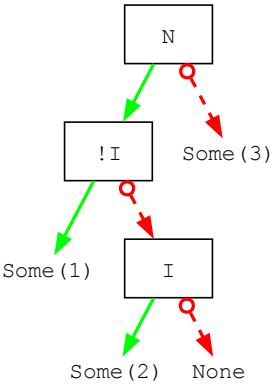
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

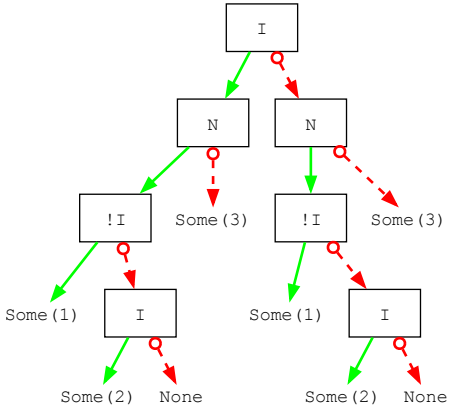
Replace `STop` with `then` branch. Replace `SEmpty` with `else` branch.



Rewrite: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

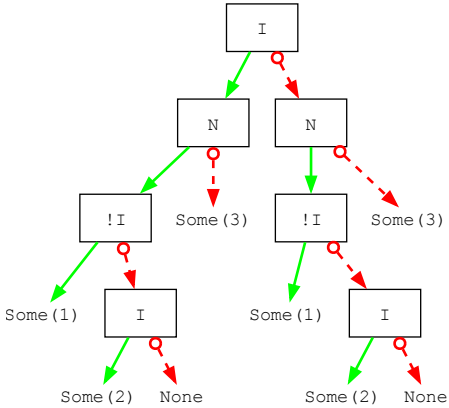
Duplicate tree, and introduce `if I.typep(x) then ... else ...`





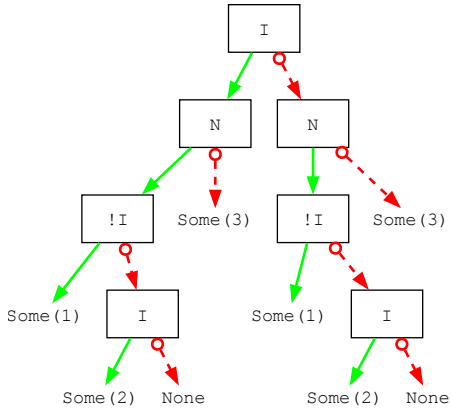
Rewrite: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

Duplicate tree, and introduce `if I.typep(x) then ... else ...`



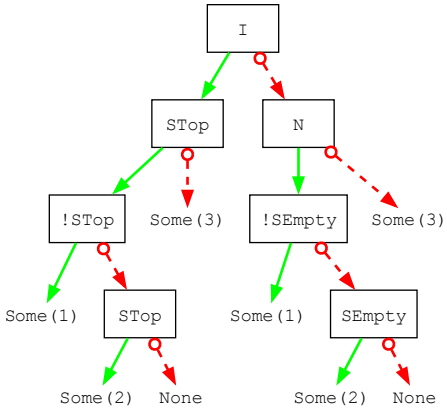
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then` part: Supertypes of `I` $\to$ `STop`. In `else` part: Subtypes of `I` $\to$ `SEmpty`.



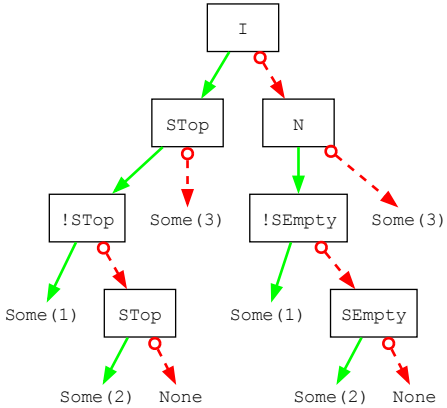
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then` part: Supertypes of `I → STop`. In `else` part: Subtypes of `I → SEmpty`.



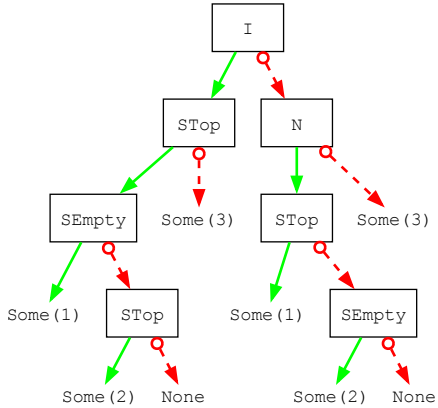
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

`!STop → SEmpty` `!SEmpty → STop`



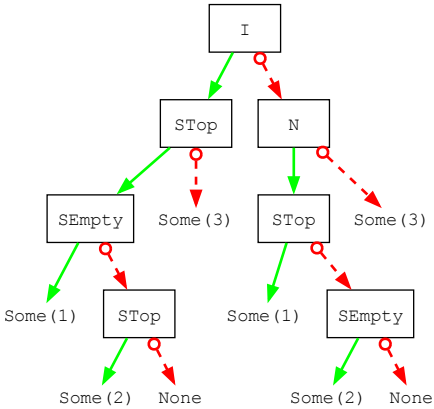
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

`!STop → SEmpty` `!SEmpty → STop`



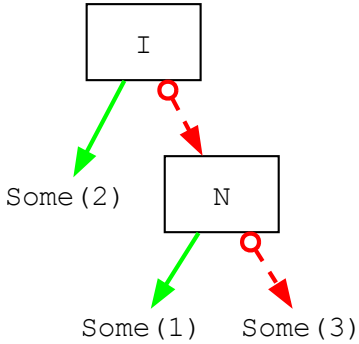
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

Replace `STop` with `then` branch. Replace `SEmpty` with `else` branch.



Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

Replace `STop` with `then` branch. Replace `SEmpty` with `else` branch.



Rewrite: Summary

Code has been rewritten so that *any type check occurs no more than once*.

```
val N = SAtomic(classOf[Number])
val I = SAtomic(classOf[Int])

if (N & !I).typep(x)
  Some(1)
else if I.typep(x)
  Some(2)
else if (!N).typep(x)
  Some(3)
else
  None
```

```
1 if I.typep(x)
2   Some(2)
3 else if N.typep(x)
4     Some(1)
5 else
6   Some(3)
```

And it is clear the code never returns `None`.

Summary: Challenges of the Project

- *Challenge № 1:* RTE Representation: Representing an RTE in Scala?
- *Challenge № 2:* DFA Construction: Constructing from RTE?
- *Challenge № 3:* Type Lattice: Union, intersection, complement types?
- *Challenge № 4:* Determinism: Type partitioning?
- *Challenge № 5:* Efficiency: Avoiding redundant type checks at run-time?

Perspectives

- Open/Closed world-view of Java types/classes
- Publish a summary of our techniques and results (PADL 2025).
- Move away from Scala 2
- Find replacement (or fix) for abandoned library: `github.com/ronmamo/reflections`

Conclusion

- Efficient pattern recognition for heterogeneous sequences for Scala
- Available here:

Scala	`https://github.com/jimka2001/scala-rte`
Clojure	`https://github.com/jimka2001/clojure-rte`
Python	`https://github.com/jimka2001/python-rte`
Common Lisp	`https://github.com/jimka2001/cl-rte`

- *Critical Feedback Welcome!*
- Contact me:
  - jimka.issy@gmail.com
  - Discord: jimka2001

functional scala 2024

THANK YOU

Sponsored by

ZIVERGE