Type Checking Heterogeneous Sequences

Jim E. Newton

EPITA Research Laboratory

November 14, 2024

Overview

History

Regular Type Expressions by Example

Implementation Challenges
- RTE Representation in Scala
- DFA Construction
- Embedding a Type System
- Determinism
- Efficiency: Redundant Type Checks

Conclusion

History

History

- Regular Type Expressions (RTEs) were introduced at 2016 European Lisp Conference, 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:

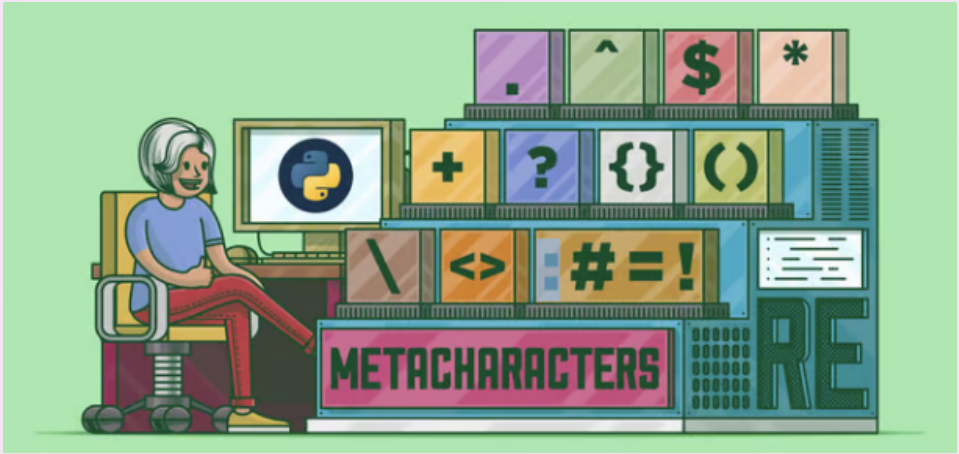
Scala	`https://github.com/jimka2001/scala-rte`
Clojure	`https://github.com/jimka2001/clojure-rte`
Python	`https://github.com/jimka2001/python-rte`
Common Lisp	`https://github.com/jimka2001/cl-rte`

Goal

- Efficiently recognize patterns in heterogeneously typed 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]

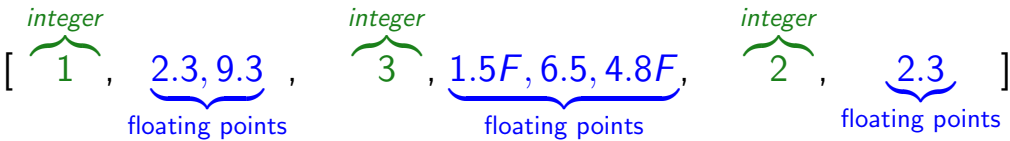
What's the pattern?

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?

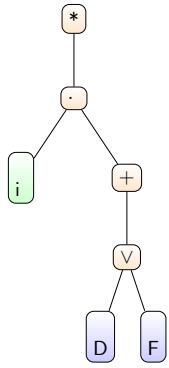


Regular Type Expressions (RTEs)

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: $(i \cdot (D \vee F)^{+})^{*}$,



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 · (Double ∪ Float)$^{+}$)*
- *Challenge №1:* We need a surface syntax 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]
```

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

Example 1

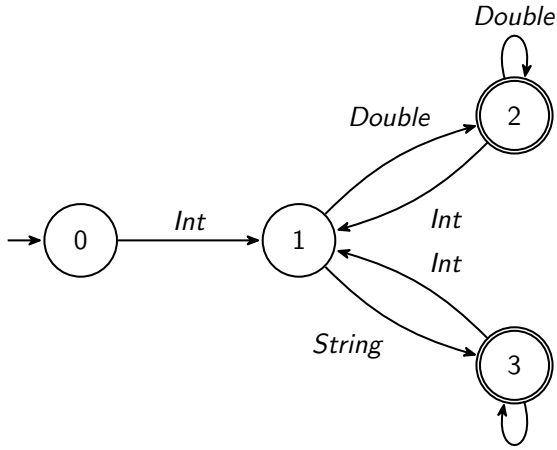
How does a pattern predicate work?

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

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

We construct a *deterministic* finite automaton (DFA).

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

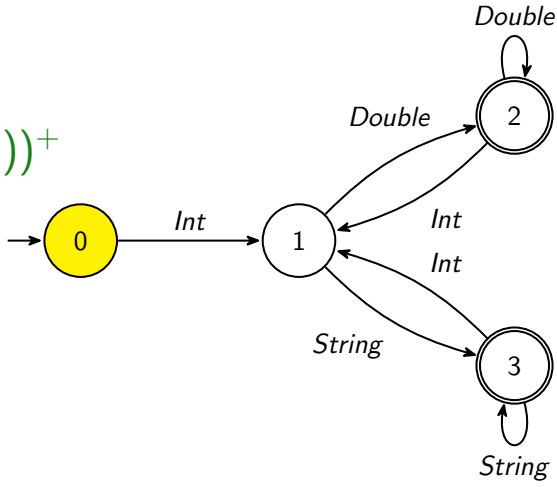


Example 1

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]
```

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



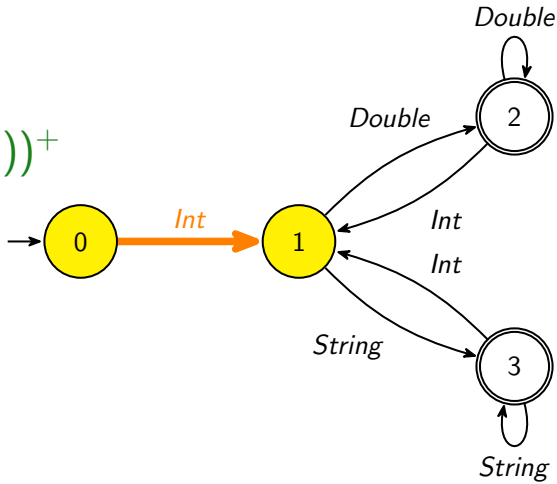
Does it match?

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]
```

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



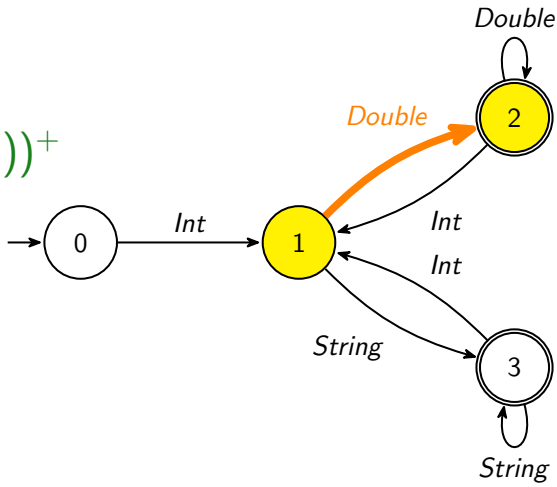
Does it match?

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]`

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



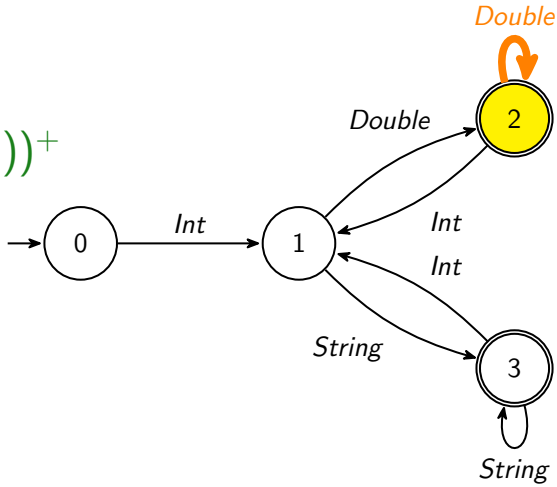
Does it match?

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]
```

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



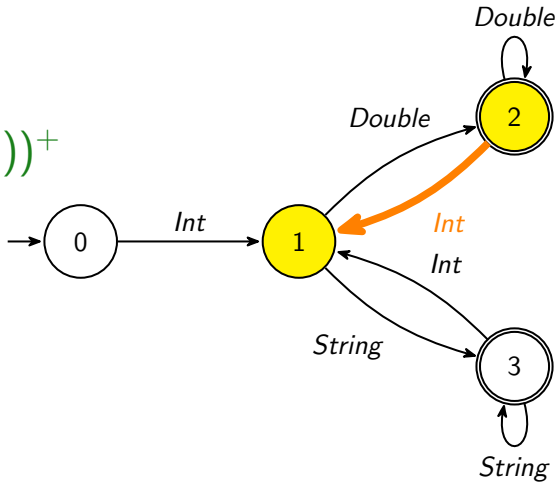
Does it match?

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]
```

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



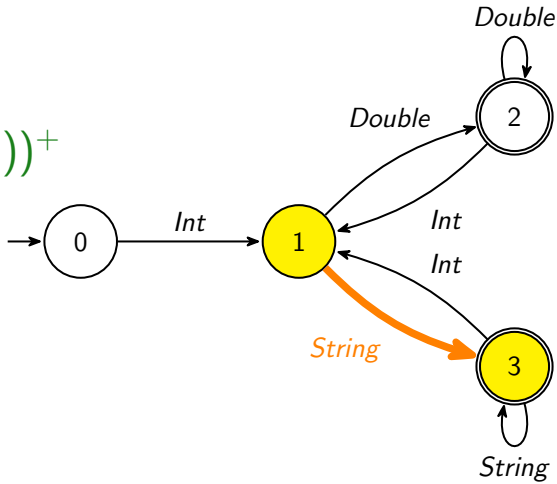
Does it match?

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]
```

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



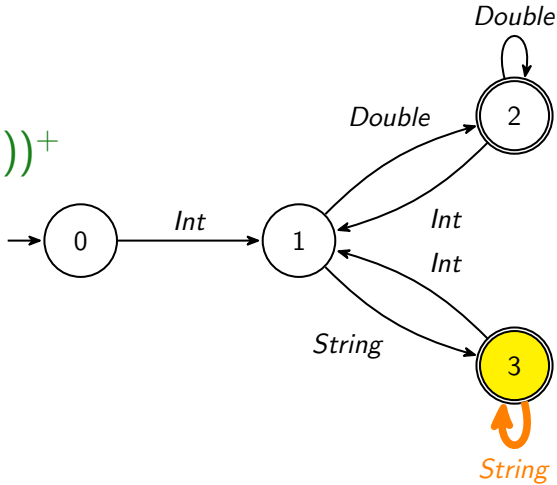
Does it match?

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]
```

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



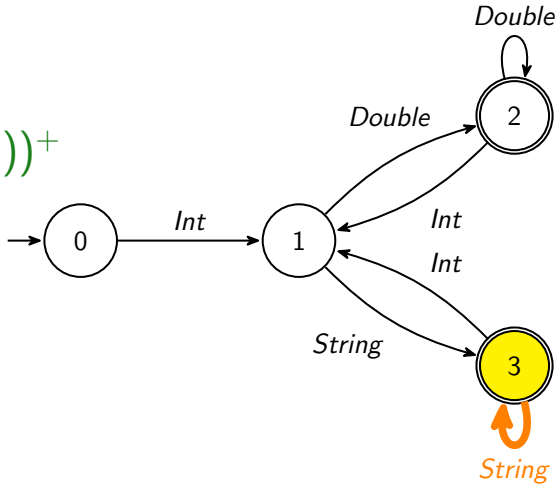
Does it match?

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]
```

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



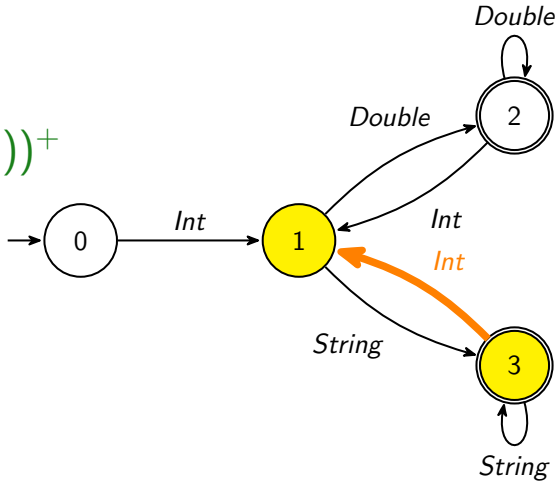
Does it match?

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]
```

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



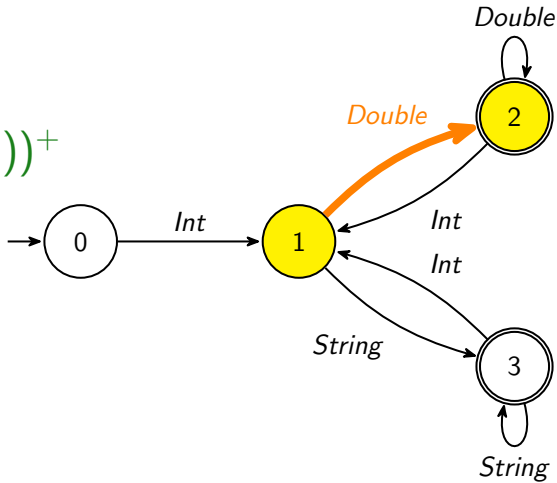
Does it match?

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]
```

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



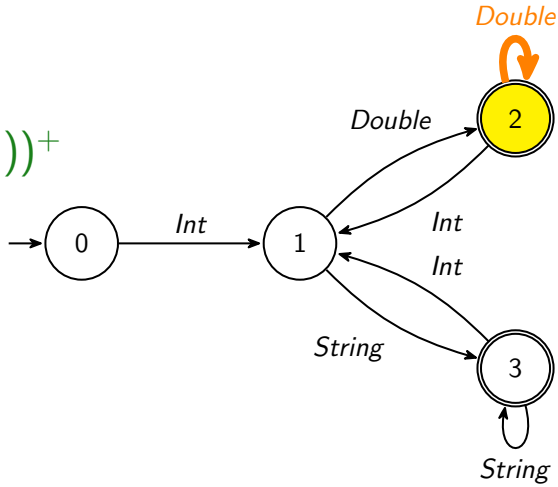
Does it match?

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]
```

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



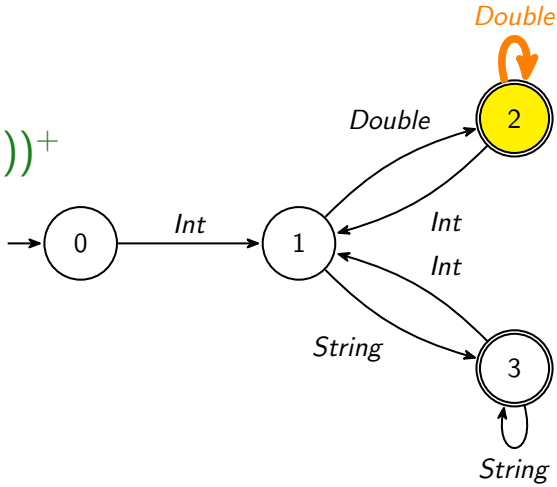
Does it match?

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]
```

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



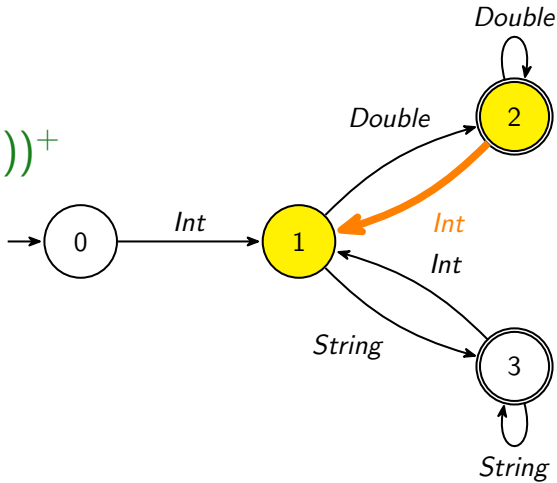
Does it match?

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]
```

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



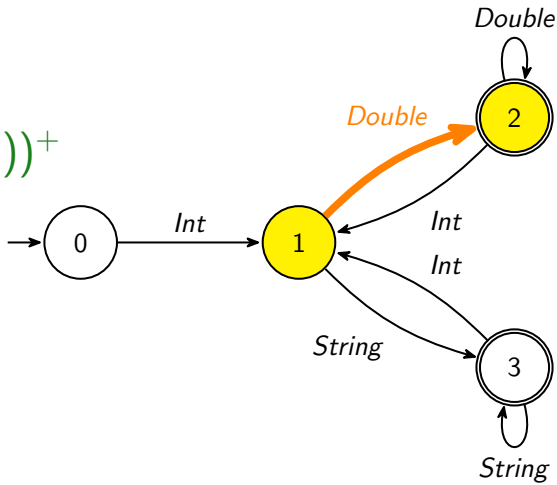
Does it match?

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]
```

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

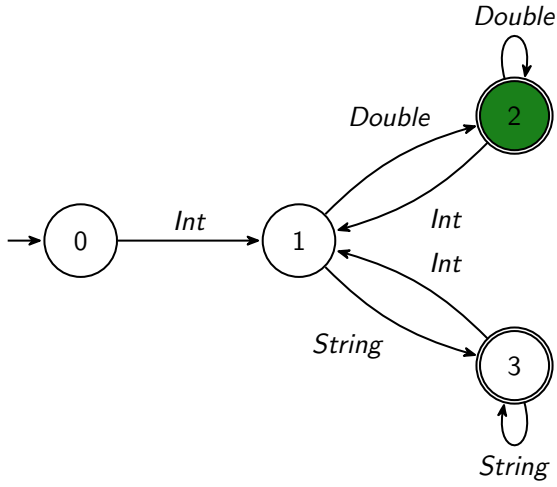


Does it match?

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]
```

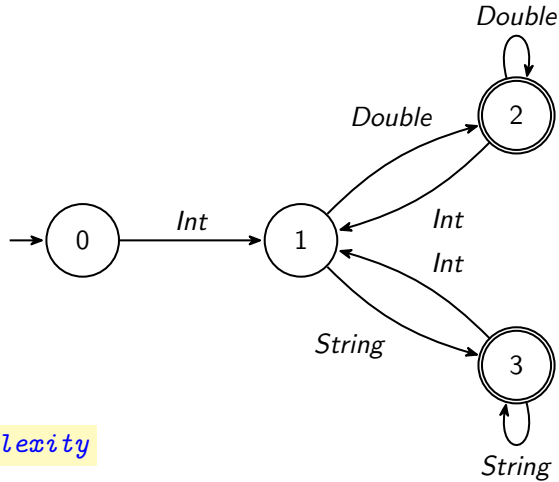


Yes, it's a match!

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]
```



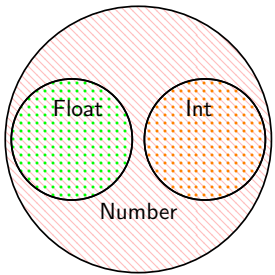
Decision procedure is $O(n)$,
independent of syntactical complexity
of the RTE.

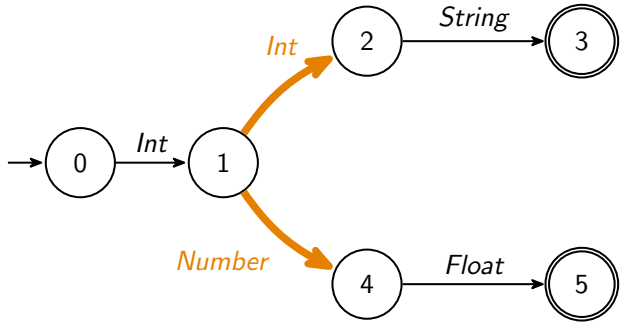
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



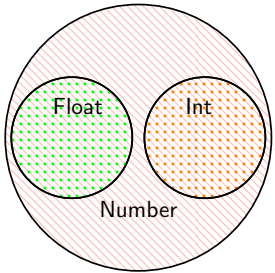


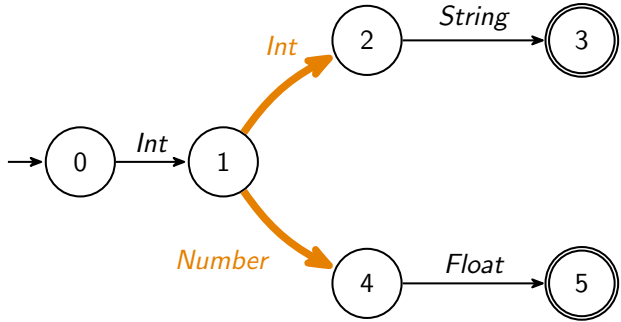
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





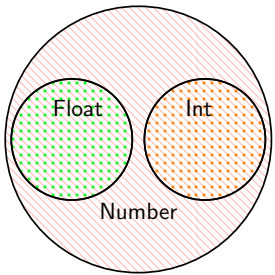
Backtracking?

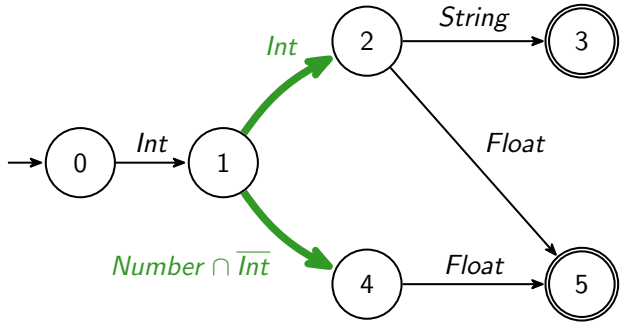
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



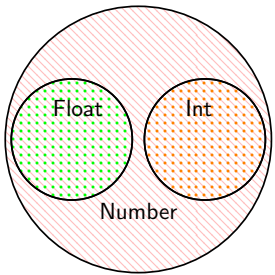


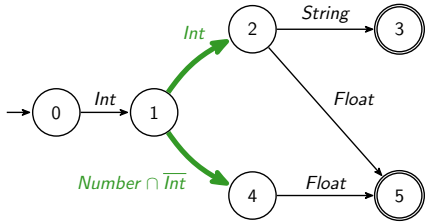
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$





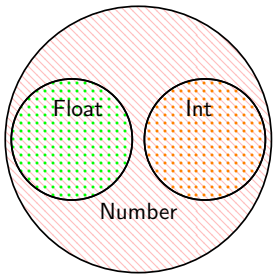
Challenge №3: How to support $Int \cap \overline{Number}$ in Scala?

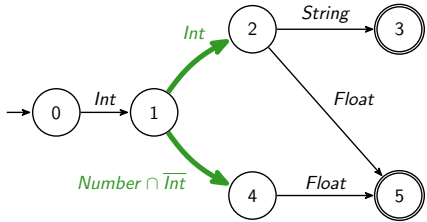
Deterministic (DFA) vs Non-deterministic (NFA)

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

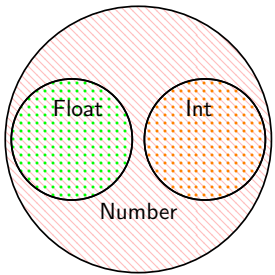
$\{String, Int, Number\} \rightarrow \{String, Int, Number \cap \overline{Int}\}$

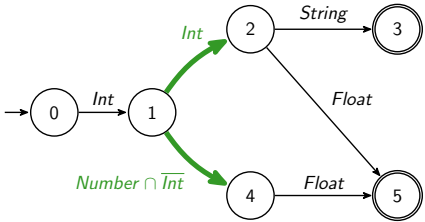
Deterministic (DFA) vs Non-deterministic (NFA)

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: Constucting 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

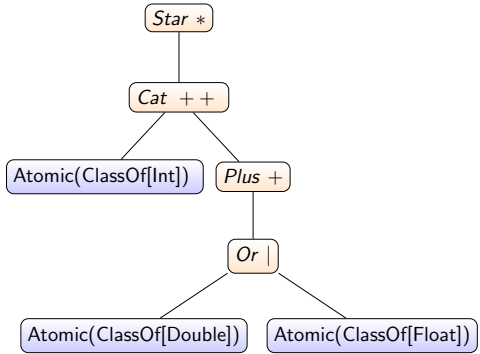
RTEs

What are Regular Type Expressions?

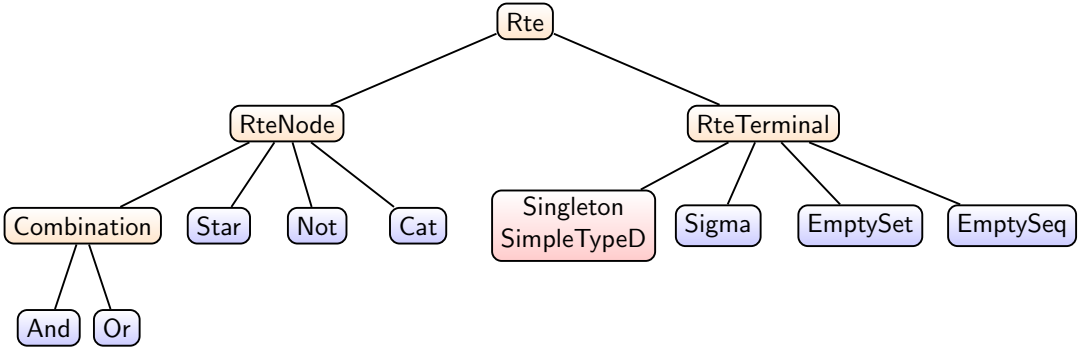
- Mathematical notation:
  $(Int \cdot (Double \cup Float)^{+})^{*}$
- Scala notation: AST

```
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



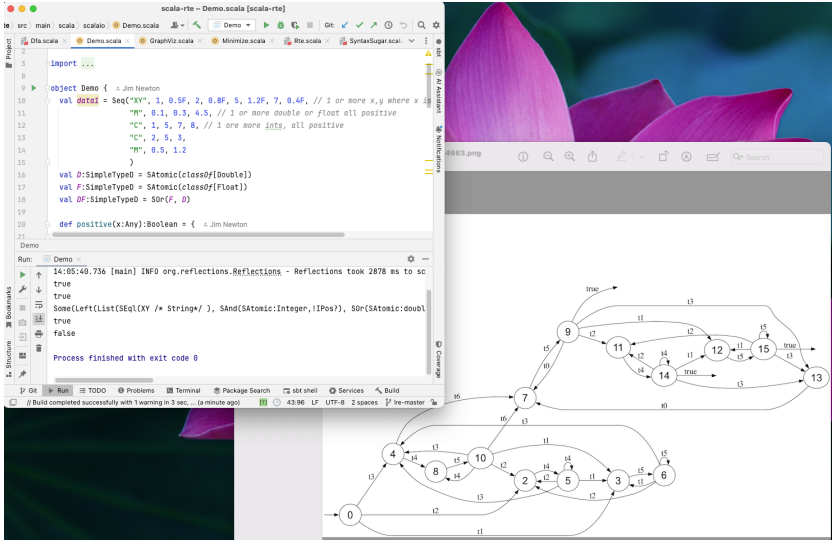
Challenge №2: DFA Construction

Given an RTE, generate a finite automaton.

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

Demo

Sample Flow



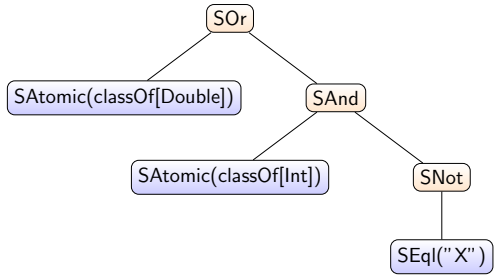
Challenge №3: Type Lattice

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

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

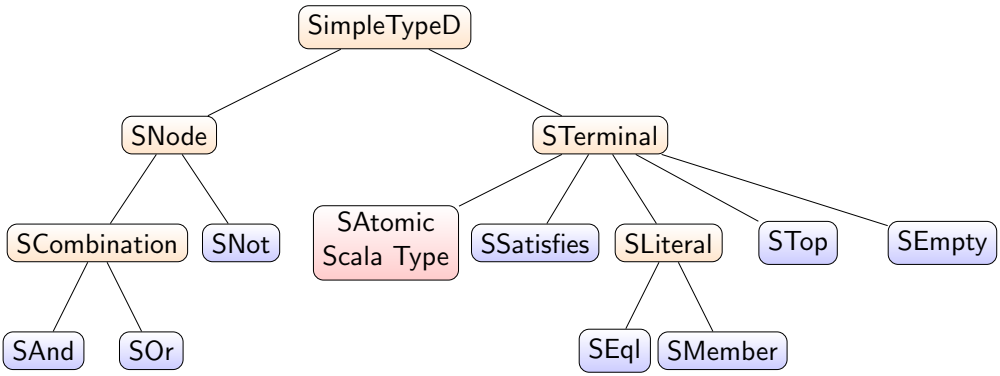
Example SimpleTypeD Expression Tree: AST

```
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*.

```
1  SAtomic(classOf[Int]).typep(-42) // returns true
2
3  // returns true
4  (SAtomic(classOf[String]) || SAtomic(classOf[Int])).typep(7)
5
6
7  // define predicate
8  def oddp(a:Any):Boolean = {
9    a match
10     case a:Int => a % 2 != 0
11     case _ => false
12 }
13
14 SSatisfies(oddp).typep(36) // returns false
```

Subtype Predicate

Semi-Boolean Subtype predicate *sometimes unanswerable*.

```
1 val Str:SimpleTypeD = SAtomic(classOf[String])
2 val Int:SimpleTypeD = SAtomic(classOf[Int])
3 val Num:SimpleTypeD = SAtomic(classOf[Number])
4 val odd:SimpleTypeD = SSatisfies(oddp, "oddp")
5
6 Str.subtypep(Int) // returns Some(false)
7 Int.subtypep(Num) // returns Some(true)
8 SSatisfies(oddp).subtypep(Int) // returns None
```

Unanswerable because:

- Impossible to compute, *e.g.* `SSatisfies`.
- Code is incomplete.
- JVM supports run-time loaded classes.
- No dependable way of finding subtypes in JVM $> 8.x$.

Simple Embedded Type System

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

Challenge №4: Deterministic State Machines

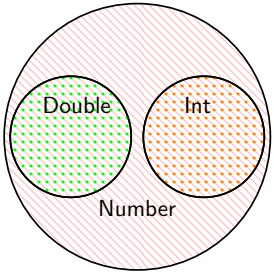
How to assure DFAs are deterministic by construction?

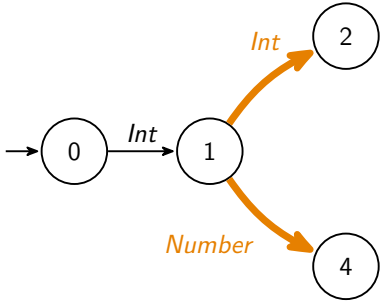
- Compute a partition of a given set of type designators,
- ... even (especially) when subtype relation is unknown.

Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$

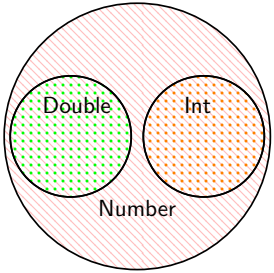


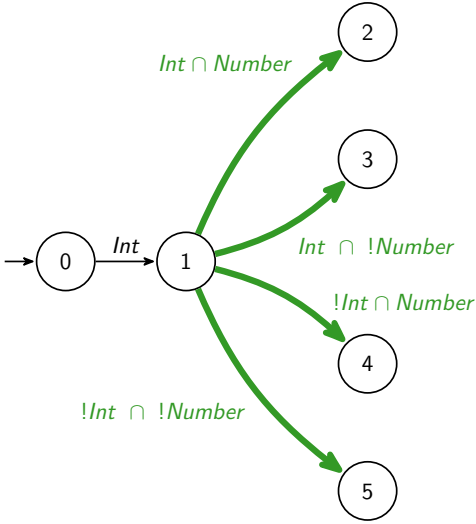


Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$

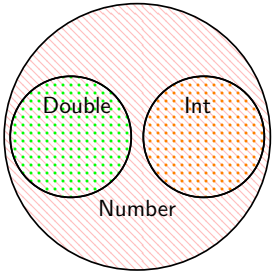


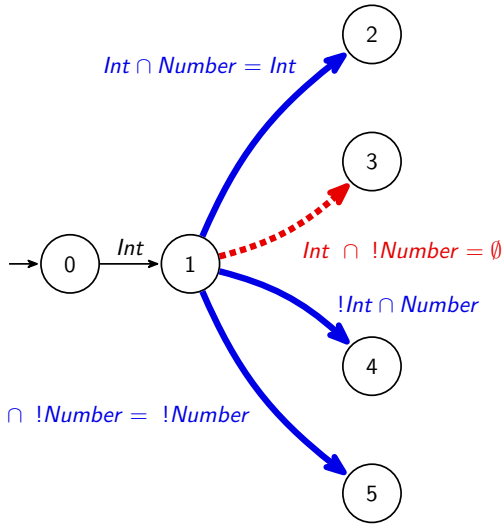


Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



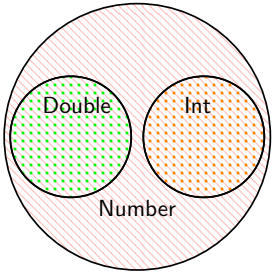


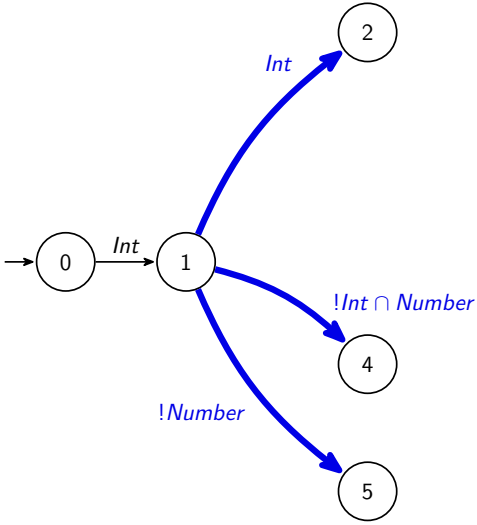
We can *decide* that state ③ is unreachable.

Non-determinism by subtype

$Int \subseteq Number$

$Int \cap Number \neq \emptyset$



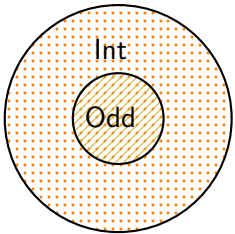


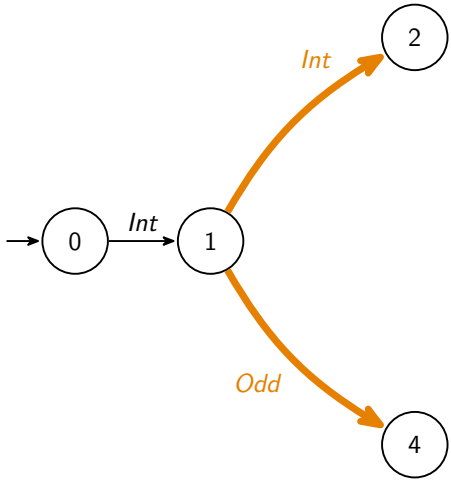
We can *decide* that state ③ is unreachable.

Non-determinism by SSatisfies

$Odd \subseteq Int$ unknown

$Odd \cap !Int = \emptyset$ unknown

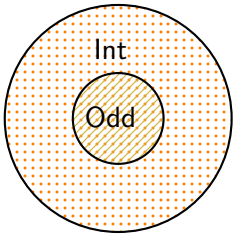


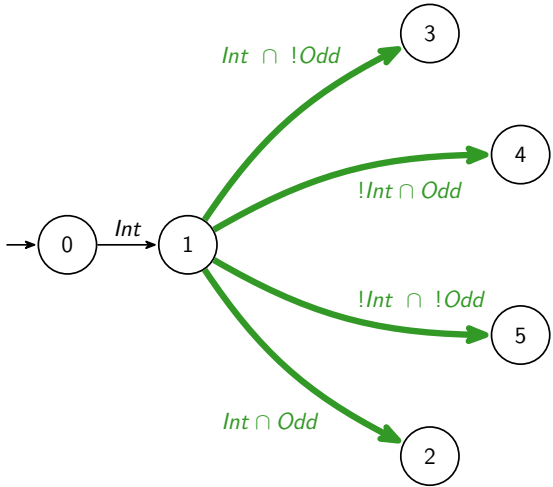


Non-determinism by SSatisfies

$Odd \subseteq Int$ unknown

$Odd \cap !Int = \emptyset$ unknown

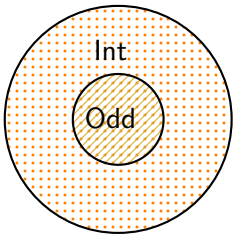


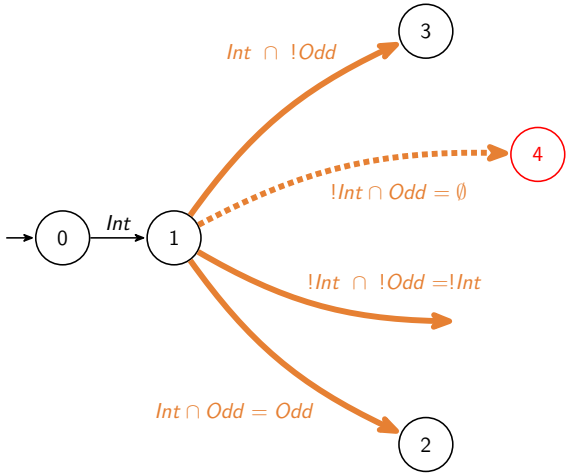


Non-determinism by SSatisfies

$Odd \subseteq Int$ unknown

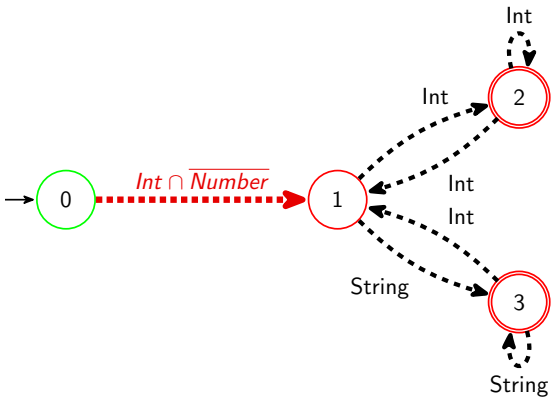
$Odd \cap !Int = \emptyset$ unknown





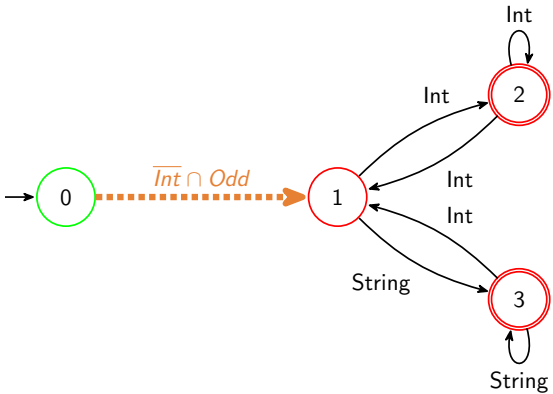
Unreachable state ④, but *undecidable*.

Unsatisfiable Transitions



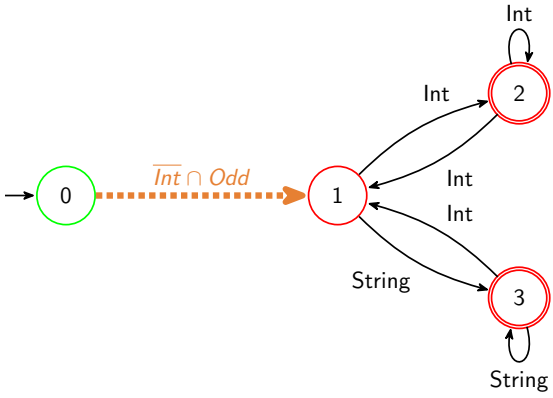
- If we determine a type is empty, then the transistion is *unsatifiable*.
- Thus we *can eliminate* the transition and unreachable states.

Indeterminant Transitions



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

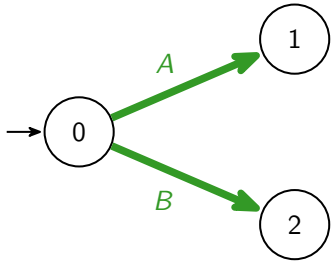
Indeterminant Transitions



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

Non-determinism with classes

```
1  final class A() {}
2  final class B() {}
3
4  class C() {}
5  trait D() {}
6
7  abstract class E() {}
8  trait F {}
9
10 class G() extends E with F {}
```

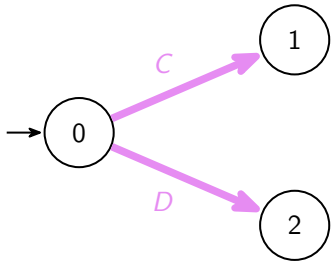


$$A \cap B = \emptyset$$

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

Non-determinism with classes

```
1  final class A() {}
2  final class B() {}
3
4  class C() {}
5  trait D() {}
6
7  abstract class E() {}
8  trait F {}
9
10 class G() extends E with F {}
```



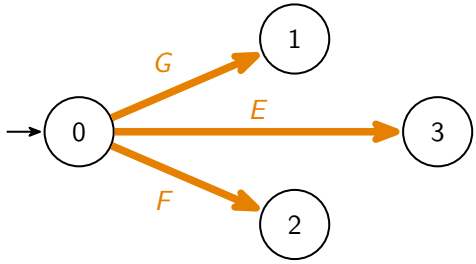
$$C \cap D = \textit{unknown}$$

Is there *somewhere* some class inherits from both.

Even if no, JVM might run-time load a library creating a common subclass.

Non-determinism with classes

```
1  final class A() {}
2  final class B() {}
3
4  class C() {}
5  trait D() {}
6
7  abstract class E() {}
8  trait F {}
9
10 class G() extends E with F {}
```

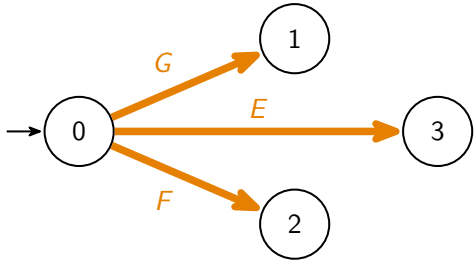


$$G \subset E \implies G \cap E \neq \emptyset$$

Explicit subtype relation.

Non-determinism with classes

```
1  final class A() {}
2  final class B() {}
3
4  class C() {}
5  trait D() {}
6
7  abstract class E() {}
8  trait F {}
9
10 class G() extends E with F {}
```



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

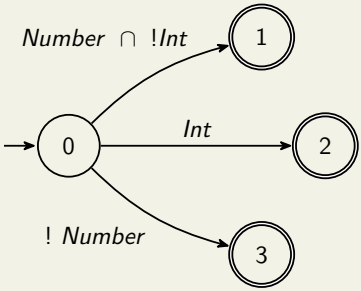
`G` inherits from `E` and `F`.

Java $> 8.x$, cannot compute subclasses.

`github.com/ronmamo/reflections` no longer maintained.

Challenge №5: Redundant Type Check

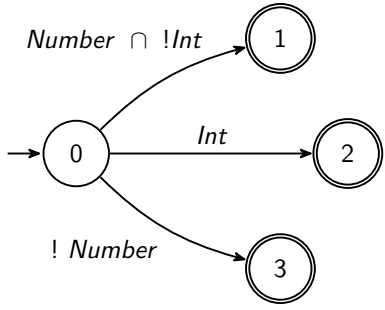
Select correct transition, avoiding *redundant type checks*.



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 ... else ...` using a lazy, `Ite` (if/then/else) data structure similar to the following.

```
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
```

```
1  Ite(N & !I, Some(1),
2             Ite(I, Some(2),
3                    Ite(!N, Some(3),
4                          None)))
```

For this presentation, we represent the decision tree as *human readable* Scala code.

Decision Tree, Before and After

Viewing the `if ... else ...` before and after as decision trees.

Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

```
Ite(N & !I, Some(1),
           Ite(I, Some(2),
                  Ite(!N, Some(3),
                          None)))
```

```
Ite(N, Ite(I, Some(2),
               Some(1)),
       Some(3))
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if N.typep(x) ... else ...`

```
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
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if N.typep(x) ... else ...`

```
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
```

```
1  if N.typep(x) {
2    ... original code ...
3  } ELSE {
4    ... original code ...
5  }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if N.typep(x) ... else ...`

```
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
```

```
1  if N.typep(x) {
2    if (N & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!N).typep(x)
7      Some(3)
8    else      None
9  } ELSE {
10   if (N & !I).typep(x)
11     Some(1)
12   else if I.typep(x)
13     Some(2)
14   else if (!N).typep(x)
15     Some(3)
16   else      None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

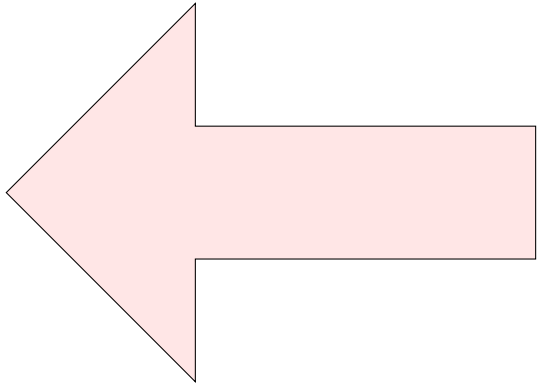
Introduce `if N.typep(x) ... else ...`

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

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if N.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if (N & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!N).typep(x)
7      Some(3)
8    else      None
9  } else {
10   if (N & !I).typep(x)
11     Some(1)
12   else if I.typep(x)
13     Some(2)
14   else if (!N).typep(x)
15     Some(3)
16   else      None
17 }
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if N.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if (N & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!N).typep(x)
7      Some(3)
8    else      None
9  } else {
10   if (N & !I).typep(x)
11     Some(1)
12   else if I.typep(x)
13     Some(2)
14   else if (!N).typep(x)
15     Some(3)
16   else      None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if (N & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!N).typep(x)
7      Some(3)
8    else      None
9  } else {
10   if (N & !I).typep(x)
11     Some(1)
12   else if I.typep(x)
13     Some(2)
14   else if (!N).typep(x)
15     Some(3)
16   else      None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

In `then` part: Supertypes of `N` → `STop`.

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

In `else` part: Subtypes of `N` → `SEmpty`.

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

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

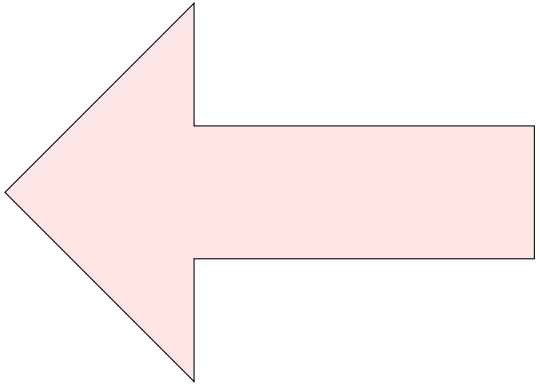
In `then` part: Supertypes of `N` → `STop`. In `else` part: Subtypes of `N` → `SEmpty`.

```
1  if N.typep(x) {
2    if (STop & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!STop).typep(x)
7      Some(3)
8    else     None
9  } else {
10   if (SEmpty & !SEmpty).typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if (!SEmpty).typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if (STop & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!STop).typep(x)
7      Some(3)
8    else      None
9  } else {
10   if (SEmpty & !SEmpty).typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if (!SEmpty).typep(x)
15     Some(3)
16   else None
17 }
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if (STop & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!STop).typep(x)
7      Some(3)
8    else     None
9  } else {
10   if (SEmpty & !SEmpty).typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if (!SEmpty).typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(STop & x)` → `x` `(SEmpty & x)` → `SEmpty`

`!STop` → `SEmpty` `!SEmpty` → `STop`

```
1  if N.typep(x) {
2    if (STop & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!STop).typep(x)
7      Some(3)
8    else     None
9  } else {
10   if (SEmpty & !SEmpty).typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if (!SEmpty).typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(STop & x)` → `x` `(SEmpty & x)` → `SEmpty`

`!STop` → `SEmpty` `!SEmpty` → `STop`

```
1  if N.typep(x) {
2    if (STop & !I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if (!STop).typep(x)
7      Some(3)
8    else      None
9  } else {
10   if (SEmpty & !SEmpty).typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if (!SEmpty).typep(x)
15     Some(3)
16   else None
17 }
```

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(STop & x)` → `x`

`!STop` → `SEmpty`

`(SEmpty & x)` → `SEmpty`

`!SEmpty` → `STop`

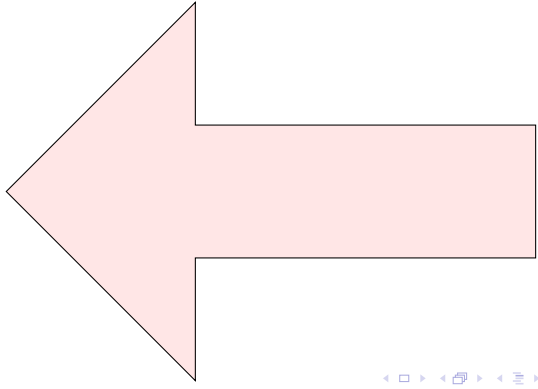
```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(STop & x) → x` `(SEmpty & x) → SEmpty`

`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(STop & x)` → `x` `(SEmpty & x)` → `SEmpty`

`!STop` → `SEmpty` `!SEmpty` → `STop`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if SEmpty.typep(x)
7      Some(3)
8    else None
9  } else {
10   if SEmpty.typep(x)
11     Some(1)
12   else if SEmpty.typep(x)
13     Some(2)
14   else if STop.typep(x)
15     Some(3)
16   else None
17 }
```

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if FALSE
7      Some(3)
8    else     None
9  } else {
10   if false
11     Some(1)
12   else if FALSE
13     Some(2)
14   else if TRUE
15     Some(3)
16   else     None
17 }
```

Rewrite: 1 → 2 → 3 → **4** → 5 → 6 → 7 → 8 → 9 → 10

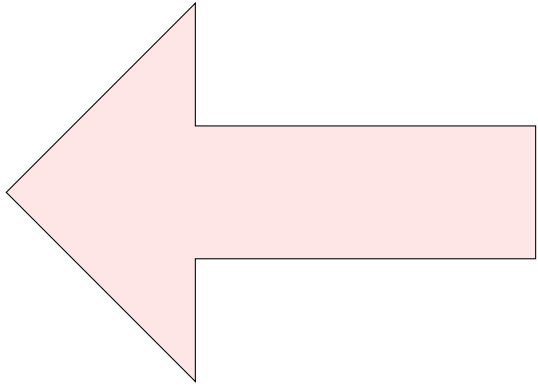
`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if FALSE
7      Some(3)
8    else     None
9  } else {
10   if false
11     Some(1)
12   else if FALSE
13     Some(2)
14   else if TRUE
15     Some(3)
16   else     None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if FALSE
7      Some(3)
8    else      None
9  } else {
10   if false
11     Some(1)
12   else if FALSE
13     Some(2)
14   else if TRUE
15     Some(3)
16   else      None
17 }
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if false
7      Some(3)
8    else      None
9  } else {
10   if false
11     Some(1)
12   else if false
13     Some(2)
14   else if true
15     Some(3)
16   else      None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(if true x else y)` → `x` `(if false x else y)` → `y`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else if false
7      Some(3)
8    else     None
9  } else {
10   if false
11     Some(1)
12   else if false
13     Some(2)
14   else if true
15     Some(3)
16   else     None
17 }
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(if true x else y)` → `x` `(if false x else y)` → `y`

```
if N.typep(x) {
  if (!I).typep(x)
    Some(1)
  else if I.typep(x)
    Some(2)
  else if false
    Some(3)
  else     None
} else {
  if false
    Some(1)
  else if false
    Some(2)
  else if true
    Some(3)
  else     None
}
```

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

Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

`(if true x else y)` $\to$ `x`

`(if false x else y)` $\to$ `y`

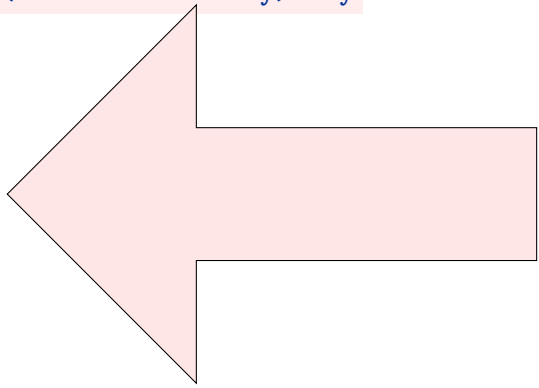
```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else None
7  } else Some(3)
```

Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

`(if true x else y)` $\to$ `x`

`(if false x else y)` $\to$ `y`

```
1 if N.typep(x) {
2   if (!I).typep(x)
3     Some(1)
4   else if I.typep(x)
5     Some(2)
6   else None
7 } else Some(3)
```



Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

`(if true x else y)` $\to$ `x` `(if false x else y)` $\to$ `y`

```
1 if N.typep(x) {
2   if (!I).typep(x)
3     Some(1)
4   else if I.typep(x)
5     Some(2)
6   else      None
7 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else      None
7  } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else      None
7  } else Some(3)
```

```
1  if N.typep(x) {
2    if I.typep(x) {
3      ... original code...
4    } ELSE {
5      ... original code...
6    }
7  } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if (!I).typep(x)
3      Some(1)
4    else if I.typep(x)
5      Some(2)
6    else      None
7  } else Some(3)
```

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } ELSE {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

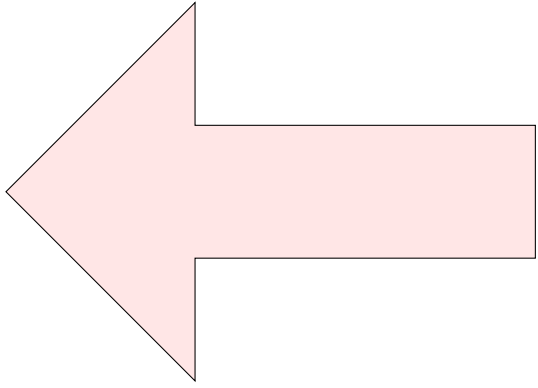
Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Introduce `if I.typep(x) ... else ...`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

In `then` part: Supertypes of `I` → `STop`.

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!I).typep(x)
4        Some(1)
5      else if I.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!I).typep(x)
10       Some(1)
11     else if I.typep(x)
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```

In `else` part: Subtypes of `I` → `SEmpty`.

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

In `then` part: Supertypes of `I` → `STop`.

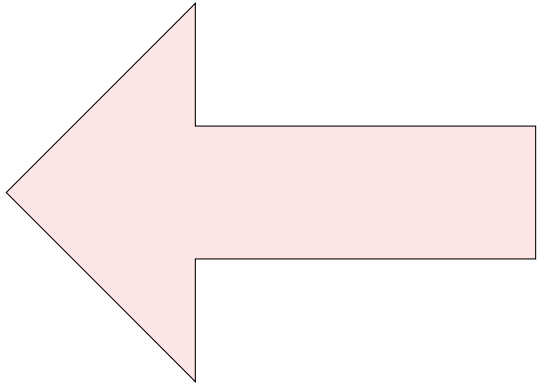
In `else` part: Subtypes of `I` → `SEmpty`.

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

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

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if (!STop).typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else None
8    } else {
9      if (!SEmpty).typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else None
14   }
15 } else Some(3)
```

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else     None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

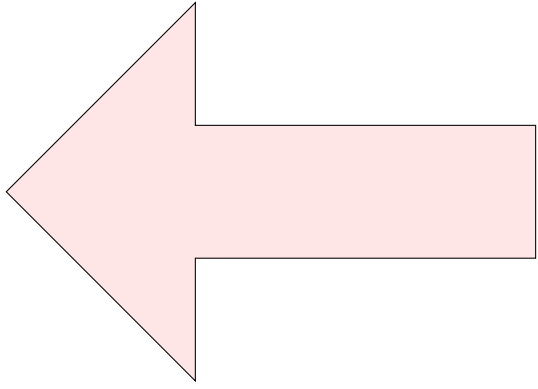
`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else     None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else     None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`!STop → SEmpty` `!SEmpty → STop`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else     None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else     None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if SEmpty.typep(x)
4        Some(1)
5      else if STop.typep(x)
6        Some(2)
7      else      None
8    } else {
9      if STop.typep(x)
10       Some(1)
11     else if SEmpty.typep(x)
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```

`STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if FALSE
4        Some(1)
5      else if TRUE
6        Some(2)
7      else      None
8    } else {
9      if true
10       Some(1)
11     else if FALSE
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false`

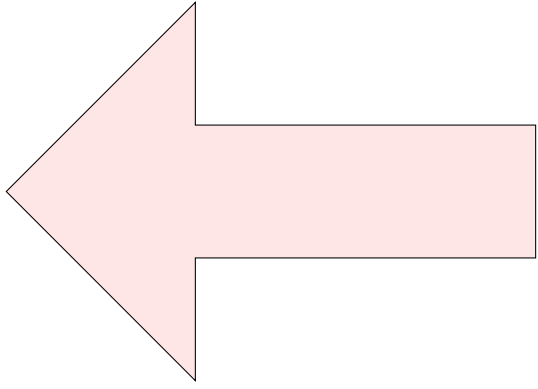
`STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if FALSE
4        Some(1)
5      else if TRUE
6        Some(2)
7      else      None
8    } else {
9      if true
10       Some(1)
11     else if FALSE
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if false
4        Some(1)
5      else if true
6        Some(2)
7      else      None
8    } else {
9      if true
10       Some(1)
11     else if false
12       Some(2)
13     else      None
14   }
15 } else Some(3)
```



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`SEmpty.typep(x)` → `false` `STop.typep(x)` → `true`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if false
4        Some(1)
5      else if true
6        Some(2)
7      else     None
8    } else {
9      if true
10       Some(1)
11     else if false
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

`(if true x else y)` $\to$ `x` `(if false x else y)` $\to$ `y`

```
1  if N.typep(x) {
2    if I.typep(x) {
3      if false
4        Some(1)
5      else if true
6        Some(2)
7      else     None
8    } else {
9      if true
10       Some(1)
11     else if false
12       Some(2)
13     else     None
14   }
15 } else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(if true x else y)` → `x` `(if false x else y)` → `y`

```
if N.typep(x) {
  if I.typep(x) {
    if false
      Some(1)
    else if true
      Some(2)
    else      None
  } else {
    if true
      Some(1)
    else if false
      Some(2)
    else      None
  }
} else Some(3)
```

```
if N.typep(x) {
  if I.typep(x)
    Some(2)
  else
    Some(1)
} else Some(3)
```

Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(if true x else y)` → `x`

`(if false x else y)` → `y`

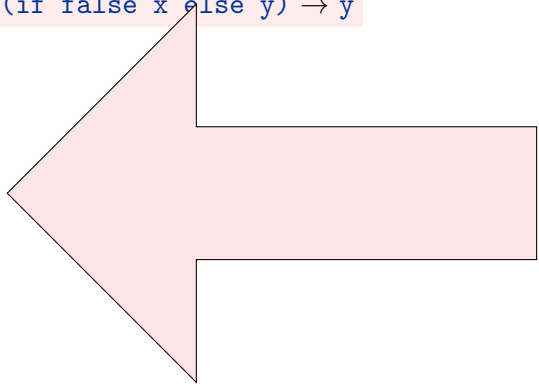
```
1 if N.typep(x) {
2   if I.typep(x)
3     Some(2)
4   else
5     Some(1)
6 } else Some(3)
```

Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

`(if true x else y)` $\to$ `x`

`(if false x else y)` $\to$ `y`

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



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

`(if true x else y)` → `x` `(if false x else y)` → `y`

```
if N.typep(x) {
  if I.typep(x)
    Some(2)
  else
    Some(1)
} else Some(3)
```

Rewrite: Summary

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

```
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
```

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

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

Challenges of the Project

- *Challenge № 1:* RTE Representation: Representing an RTE in Scala?
- *Challenge № 2:* DFA Construction: Constucting 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?

Summary

- Regular Type Expressions in Scala
- ... using symbolic finite automata
- ... extending the Scala type system
- Demo of sequence pattern matching
- Several theoretical interesting problems:
- ... Type partitioning
- ... Efficient elimination of redundant type checks

Perspectives

- Improve predicate satisfaction heuristics.
- Open/Closed world-view of Java types/classes
- Publish a summary of our techniques and results.
- Move away from Scala 2
- Find replacement for abandoned library: `github.com/ronmamo/reflections`

Conclusion

- An implementation of efficient pattern recognition for heterogeneous sequences
- In reflective programming languages
- ... notably in 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`

Questions and Answers