Abstractions for Type-Level Programming

Olivier Blanvillain

Tuesday, 22 March 2022

Example 1: Database queries (2016)

Example 1: Database queries (2016)

Spark APIs are practically untyped:

```
class DataFrame {
  /** Inner equi-join with another DataFrame on the given column.
   *  The join column will only appear once in the output. */
  def join(right: DataFrame, column: String): DataFrame
}
```

Revisited with type-level programming:

```
class DF[X] {
  def join[Y](df: DF[Y], col: String): DF[col+(X-col)++(Y-col)]
}
```

Example 1: Database queries (2016)

Spark APIs are practically untyped:

```
class DataFrame {
  /** Inner equi-join with another DataFrame on the given column.
   *  The join column will only appear once in the output. */
  def join(right: DataFrame, column: String): DataFrame
}
```

Revisited with type-level programming:

```
class DF[X] {
  def join[Y](df: DF[Y], col: String): DF[col+(X-col)++(Y-col)]
}
```

Example 2: Regular expressions (2022)

Regular expressions in Scala's standard library:

```
val rational = new Regex("(\\d+)\\.?(\\d+)?")
rational.unapply("3.1415"): Option[Seq[String]]
```

Revisited with type-level programming:

```
rational.unapply("3.1415"): Option[(String, Option[String])]
```

```
class Regex(pattern: String) {
  def unapply(s: String): Option[GroupsOf[pattern]]
}
```

Example 2: Regular expressions (2022)

Regular expressions in Scala's standard library:

```
val rational = new Regex("(\\d+)\\.?(\\d+)?")
rational.unapply("3.1415"): Option[Seq[String]]
```

Revisited with type-level programming:

```
rational.unapply("3.1415"): Option[(String, Option[String])]
```

```
class Regex(pattern: String) {
  def unapply(s: String): Option[GroupsOf[pattern]]
}
```

Example 2: Regular expressions (2022)

Regular expressions in Scala's standard library:

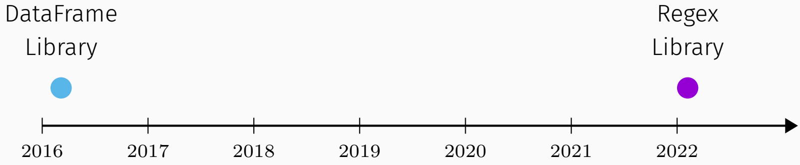
```
val rational = new Regex("(\\d+)\\.?(\\d+)?")
rational.unapply("3.1415"): Option[Seq[String]]
```

Revisited with type-level programming:

```
rational.unapply("3.1415"): Option[(String, Option[String])]
```

```
class Regex(pattern: String) {
  def unapply(s: String): Option[GroupsOf[pattern]]
}
```

What changed between 2016 and 2022?



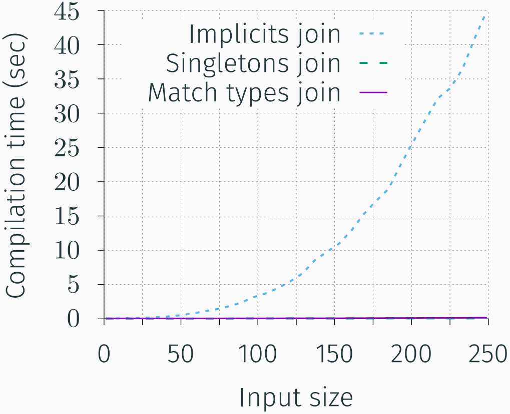
What changed between 2016 and 2022?

Example 1 uses a hack, implicits:

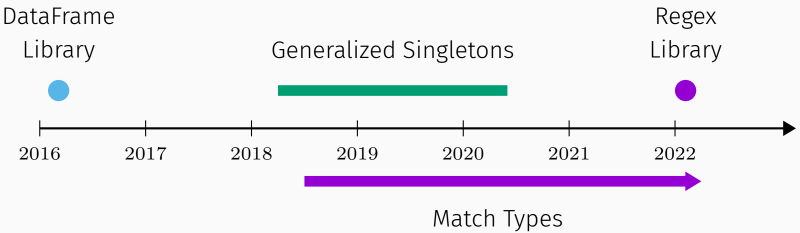
- convoluted to use
- slow to compile

Example 2 uses a new language construct, match types, one of the main contributions of this thesis.

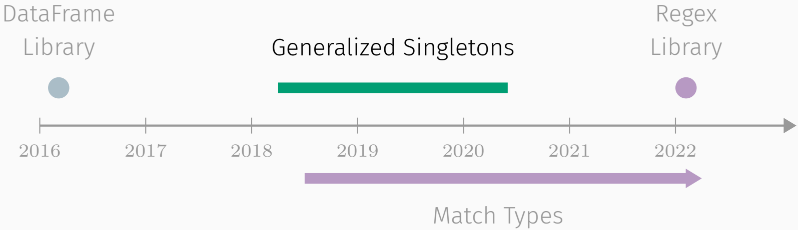
Compilation times for the type-level join operation



Timeline of my PhD



Part I: Generalized Singleton Types



Singleton types

Scala supports a few singleton types:

- `x.type`, the type of the variable `x` (since forever)
- `42`, the type of the integer literal `42` (since 2016)
- `+`, the type of integer addition (since 2020)

Question: How much of the language can we represent in types?

Singleton types

Scala supports a few singleton types:

- `x.type`, the type of the variable `x` (since forever)
- `42`, the type of the integer literal `42` (since 2016)
- `+`, the type of integer addition (since 2020)

Question: How much of the language can we represent in types?

Generalized singleton types: A proposal

Our proposal consists of 3 changes:

1. new types for if-then-else, pattern matching, constructors, functions calls
2. “precise mode” of type inference
3. type evaluation, during subtyping

New types

What’s the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

New types

What's the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

New types

What's the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

New types

What's the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

New types

What's the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

New types

What's the type of `if (x == 0) "zero" else "one"`?

- `String` (Scala 2)
- `"zero" | "one"` (Scala 3)
- `If[x.type == 0, "zero", "one"]` (proposed)
- `{ if (x == 0) "zero" else "one" }` (proposed, syntactic sugar)

Similarly, we add new types for other constructs: `Match[]`, `New[]` and `Call[]`, and the corresponding syntactic sugar.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
def int2str(x: Int) =
  if (x == 0) "zero" else "one"
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
def int2str(x: Int): String =
  if (x == 0) "zero" else "one"
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
dependent
def int2str(x: Int)
  : { if (x == 0) "zero" else "one" }
  =   if (x == 0) "zero" else "one"
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
dependent
def int2str(x: Int)
  : If[x.type == 0, "zero", "one"]
  =   if (x == 0) "zero" else "one"
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
dependent
def int2str(x: Int)
  : If[x.type == 0, "zero", "one"]
  =   if (x == 0) "zero" else "one"
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

"Precise mode" of type inference

We need two modes of type inference, for backwards compatibility.

```
dependent
def int2str(x: Int)
  : If[x.type == 0, "zero", String]
  =   if (x == 0) "zero" else readString()
```

The "precise mode" is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference

We need two modes of type inference, for backwards compatibility.

```
dependent
def int2str(x: Int)
  : { if (x == 0) "zero" else (_: String) }
  =   if (x == 0) "zero" else readString()
```

The “precise mode” is best effort: it lifts whatever possible to the type level and approximates the rest.

“Precise mode” of type inference: list concatenation example

```
dependent def concat(xs: List, ys: List) <: List =
  xs match
    case x :: xs => x :: concat(xs, ys)
    case Nil => ys

dependent val l1 = "A" :: Nil
dependent val l2 = "B" :: Nil
dependent val l3 = concat(l1, l2)
l3: { "A" :: "B" :: Nil }
```

“Precise mode” of type inference: list concatenation example

```
dependent def concat(xs: List, ys: List) <: List =
  xs match
    case x :: xs => x :: concat(xs, ys)
    case Nil => ys

dependent val l1 = "A" :: Nil
dependent val l2 = "B" :: Nil
dependent val l3 = concat(l1, l2)
l3: { "A" :: "B" :: Nil }
```

"Precise mode" of type inference: list concatenation example

```
dependent def concat(xs: List, ys: List) <: List =
  xs match
    case x :: xs => x :: concat(xs, ys)
    case Nil => ys

val l1: List      = "A" :: Nil
dependent val l2 = "B" :: Nil
dependent val l3 = concat(l1, l2)
l3: { concat(l1, l2) }
```

“Precise mode” of type inference: list concatenation example

```
dependent def concat(xs: List, ys: List) <: List =
  xs match
    case x :: xs => x :: concat(xs, ys)
    case Nil => ys

dependent val l1 = "A" :: Nil
val l2: List      = "B" :: Nil
dependent val l3 = concat(l1, l2)
l3: { "A" :: l2 }
```

Type evaluation

During subtyping, we evaluate both sides:

```
A <: B  iff  eval(A) <: eval(B)
```

Straightforward for if-then-else:

```
eval(If[true, A, B]) = A
eval(If[false, A, B]) = B
```

More interesting for pattern matching: we “desugar” pattern matching expressions into if-then-else & type tests.

We evaluate type tests using subtyping and type disjointness:

```
eval(x.asInstanceOf[Foo]) = true   iff  x.type <: Foo
eval(x.asInstanceOf[Foo]) = false  iff  disj(x.type, Foo)
```

Type evaluation

During subtyping, we evaluate both sides:

```
A <: B  iff  eval(A) <: eval(B)
```

Straightforward for if-then-else:

```
eval(If[true, A, B]) = A
eval(If[false, A, B]) = B
```

More interesting for pattern matching: we “desugar” pattern matching expressions into if-then-else & type tests.

We evaluate type tests using subtyping and type disjointness:

```
eval(x.asInstanceOf[Foo]) = true   iff  x.type <: Foo
eval(x.asInstanceOf[Foo]) = false  iff  disj(x.type, Foo)
```

Type evaluation

During subtyping, we evaluate both sides:

```
A <: B  iff  eval(A) <: eval(B)
```

Straightforward for if-then-else:

```
eval(If[true, A, B]) = A
eval(If[false, A, B]) = B
```

More interesting for pattern matching: we “desugar” pattern matching expressions into if-then-else & type tests.

We evaluate type tests using subtyping and type disjointness:

```
eval(x.asInstanceOf[Foo]) = true   iff  x.type <: Foo
eval(x.asInstanceOf[Foo]) = false  iff  disj(x.type, Foo)
```

Type evaluation

During subtyping, we evaluate both sides:

```
A <: B  iff  eval(A) <: eval(B)
```

Straightforward for if-then-else:

```
eval(If[true, A, B]) = A
eval(If[false, A, B]) = B
```

More interesting for pattern matching: we "desugar" pattern matching expressions into if-then-else & type tests.

We evaluate type tests using subtyping and type disjointness:

```
eval(x.asInstanceOf[Foo]) = true   iff  x.type <: Foo
eval(x.asInstanceOf[Foo]) = false  iff  disj(x.type, Foo)
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any) =
  x match
    case s: String => s
    case i: Int => i+1
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  x match
    case s: String => s
    case i: Int => i+1
} =
  x match
    case s: String => s
    case i: Int => i+1
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1
```

Type evaluation: pattern matching example

```scala
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): { foo(42) }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): { 42.asInstanceOf[Int] + 1 }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): { 42 + 1 }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): { 43 }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): 43
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1
```

```
foo(42): 43
```

```
foo(readString())
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): 43
foo(readString()): { foo(_: String) }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1
```

```
foo(42): 43
foo(readString()): { (_: String).asInstanceOf[String] }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1

foo(42): 43
foo(readString()): { (_: String) }
```

Type evaluation: pattern matching example

```
dependent val foo(x: Any): {
  if (x.isInstanceOf[String]) x.asInstanceOf[String]
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1
  else throw new MatchError()
} =
  x match
    case s: String => s
    case i: Int => i+1
```

```
foo(42): 43
```

```
foo(readString()): String
```

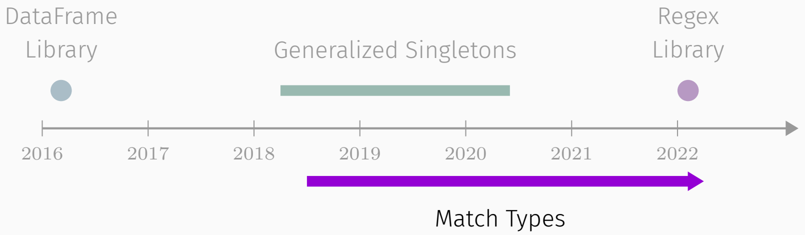
Generalized Singletons: Recap

We proposed the following 3 changes:

1. lift a subset of Scala's language constructs to the type level
2. add a "precise mode" of type inference
3. evaluate those types during subtyping

TODO

Part II: Match Types



Overview

```scala
type Elem[X] = X match
  case String => Char
  case List[t] => Elem[t]
  case Any => X
```

Disjointness

`Elem[Seq[Int]]` is stuck

Relating match terms and types: a defining property

`Elem[Seq[Int]]` is stuck

Formalizing Match Types

System FM is an extension of System $F_{<:}$ with

- pattern matching
- opaque classes
- match types

Design goals:

- explain the essence of match types
- give us confidence in our design
- be as simple as possible

System FM is parametric

System FM is parametrized by a set of externally defined classes:

FM(C, Ψ, Ξ)

- C := set of classes
- Ψ := class inheritance relation
- Ξ := class disjointness relation

For example:

`class A`	$C = \{A\}$	$\Psi = \varnothing$	$\Xi = \varnothing$
`class A; class B extends A`	$C = \{A, B\}$	$\Psi = \{(B, A)\}$	$\Xi = \varnothing$
`class A; class D`	$C = \{A, D\}$	$\Psi = \varnothing$	$\Xi = \{(A, D)\}$
`class A; trait T`	$C = \{A, T\}$	$\Psi = \varnothing$	$\Xi = \varnothing$

System FM is parametric

System FM is parametrized by a set of externally defined classes:

FM(C, Ψ, Ξ)

- C $\coloneqq$ set of classes
- Ψ $\coloneqq$ class inheritance relation
- Ξ $\coloneqq$ class disjointness relation

For example:

`class A`	$C = \{A\}$	$\Psi = \varnothing$	$\Xi = \varnothing$
`class A; class B extends A`	$C = \{A, B\}$	$\Psi = \{(B, A)\}$	$\Xi = \varnothing$
`class A; class D`	$C = \{A, D\}$	$\Psi = \varnothing$	$\Xi = \{(A, D)\}$
`class A; trait T`	$C = \{A, T\}$	$\Psi = \varnothing$	$\Xi = \varnothing$

System FM's syntax

$t ::=$		
	x	*variable*
	$\lambda x{:}T.\ t$	*abstraction*
	$\lambda X{<:}T.\ t$	*type abstraction*
	$t\ t$	*application*
	$t\ T$	*type application*
	$\mathit{new}\ C$	*constructor call*
	$t\ \ \mathit{match}\{\overline{x{:}C \Rightarrow t}\}\mathit{or}\ \ t$	*match expr.*
$v ::=$		
	$\lambda x{:}T.\ t$	*abstraction*
	$\lambda X{<:}T.\ t$	*type abstraction*
	$\mathit{new}\ C$	*constructor call*

$T ::=$		
	X	*type variable*
	$T \to T$	*type of functions*
	$\forall X{<:}T.\ T$	*universal type*
	Top	*maximum type*
	C	*class*
	$\{\mathit{new}\ C\}$	*constructor singleton*
	$T\ \ \mathit{match}\{\overline{T \Rightarrow T}\}\mathit{or}\ \ T$	*match type*
$\Gamma ::=$		
	$\varnothing$	*empty context*
	$\Gamma, x{:}T$	*term binding*
	$\Gamma, X{<:}T$	*type binding*

System FM's pattern matching evaluation

Given a concrete scrutinee, we evaluate match expressions with a few lookups in the inheritance relation:

$$\frac{(\mathrm{C}, \mathrm{C}_n) \in \Psi \qquad \forall m < n.\ (\mathrm{C}, \mathrm{C}_m) \notin \Psi}{\mathit{new}\,\mathrm{C}\ \ \mathit{match}\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\}\mathit{or}\ \ \mathrm{t}_d \longrightarrow [\mathrm{x}_n \mapsto \mathit{new}\,\mathrm{C}]\mathrm{t}_n} \text{ (E-MATCH2)}$$

System FM's disjointness relation

$$\frac{(C_1, C_2) \in \Xi}{\Gamma \vdash \mathrm{disj}(C_1, C_2)} \quad \text{(D-XI)}$$

$$\frac{(C_1, C_2) \notin \Psi}{\Gamma \vdash \mathrm{disj}(\{\mathit{new}\ C_1\}, C_2)} \quad \text{(D-PSI)}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash \mathrm{disj}(U, T)}{\Gamma \vdash \mathrm{disj}(S, T)} \quad \text{(D-SUB)}$$

$$\Gamma \vdash \mathrm{disj}(T_1 \to T_2, C) \quad \text{(D-ARROW)}$$

$$\Gamma \vdash \mathrm{disj}(\forall X <: T_1.\ T_2, C) \quad \text{(D-ALL)}$$

System FM's match types evaluation rule (subtyping)

$$\frac{\Gamma \vdash \mathrm{T}_s <: \mathrm{S}_n \qquad \forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)}{\Gamma \vdash \mathrm{T}_s \;\; match\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\} or \;\; \mathrm{T}_d =:= \mathrm{T}_n}$$

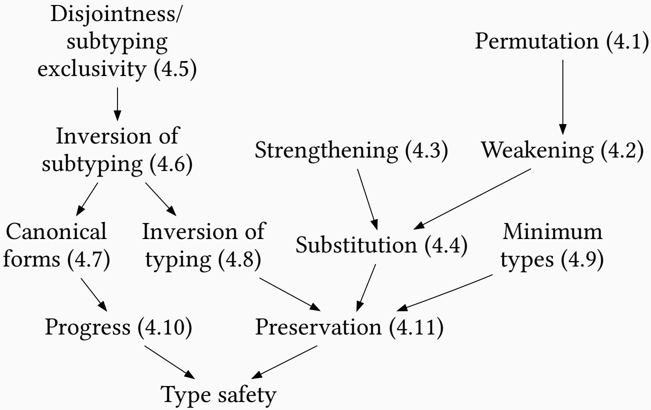
System FM's type safety

We show type safety through the progress and preservation.

Our proof comes in two versions:

1. Pen and paper (~30 pages)
2. Coq mechanization (~6000 LOC)

Structure of the type safety proof



Empty Types

Our system does not like empty types.

`Nothing`, in particular, is both subtype and disjoint from every type.

Intersections require special care.

```
type M[X] = X match
  case Int => String
  case String => Int

class C:
  type X
  def f(bad: M[X & String]): Int = bad

class D extends C:
  type X = Int
```

Empty Types

Our system does not like empty types.

`Nothing`, in particular, is both subtype and disjoint from every type.

Intersections require special care.

```
type M[X] = X match
  case Int => String
  case String => Int

class C:
  type X
  def f(bad: M[X & String]): Int = bad

class D extends C:
  type X = Int
```

Variance

At first glance, we can’t prove disjointness when variance is involved.

```
F[+T]:   T1 <: T2   implies   F[T1] <: F[T2]
G[-T]:   T1 <: T2   implies   G[T1] >: G[T2]
```

We make an exception for covariant type that are used as class fields:

```
case class Some[+A](value: A)
```

`Some[String]` and `Some[Int]` are disjoint, since there is no runtime value of type `Some[Nothing]`.

Variance

At first glance, we can't prove disjointness when variance is involved.

```
F[+T]:   T1 <: T2   implies   F[T1] <: F[T2]
G[-T]:   T1 <: T2   implies   G[T1] >: G[T2]
```

We make an exception for covariant type that are used as class fields:

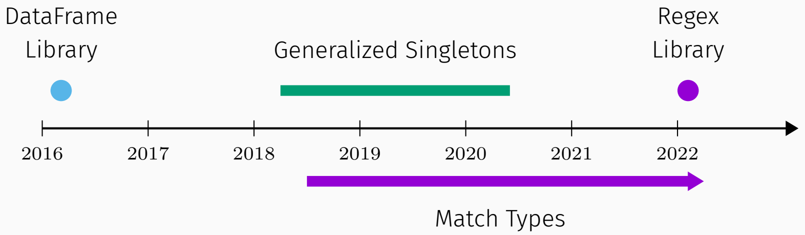
```
case class Some[+A](value: A)
```

`Some[String]` and `Some[Int]` are disjoint, since there is no runtime value of type `Some[Nothing]`.

Match Types, Conclusion

TODO

Thank you!



DataFrame Library
Generalized Singletons
Regex Library
2016
2017
2018
2019
2020
2021
2022
Match Types

Type-Safe Regular Expressions (simplified)

```
val rational = new Regex("(\\d+)\\.(\\d+)")
rational.unapply("3.1415"): Option[(String, String)]

type GroupsOf[P <: String] =
  Reverse[Loop[P, 0, Length[P], EmptyTuple]]

type Loop[P, Lo, Hi, Acc <: Tuple] =
  Lo match
    case Hi => Acc
    case _ => CharAt[P, Lo] match
      case '('  => Loop[P, Lo + 1, Hi, String *: Acc]
      case '\\' => Loop[P, Lo + 2, Hi, Acc]
      case _ => Loop[P, Lo + 1, Hi, Acc]
```