

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)

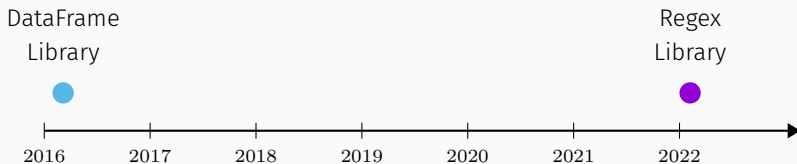
Regular expressions in Scala's standard library:

```
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?

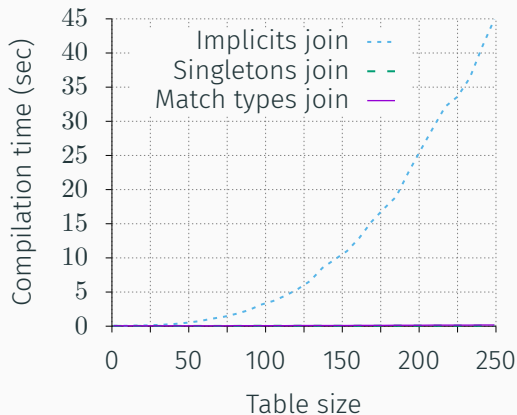
The Spark library uses a hack, implicits:

- convoluted to use
- slow to compile

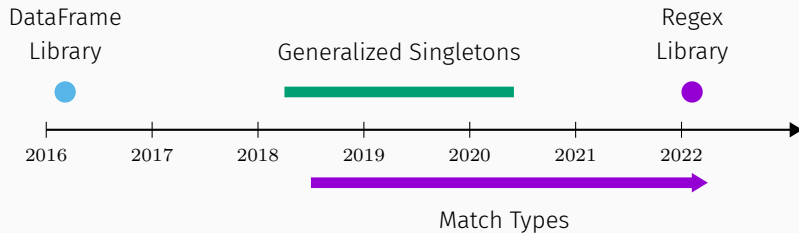
The regex library uses a first-class language construct:

- easier to use
- faster to compile

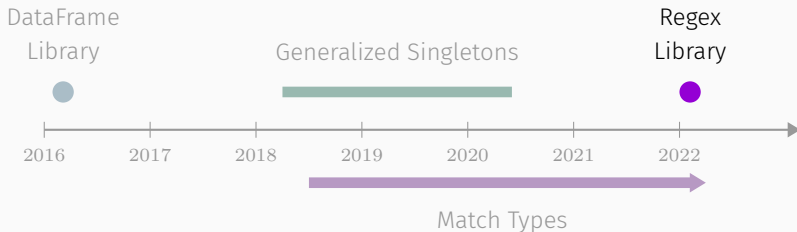
Compilation times for the type-level join operation



Timeline of my PhD

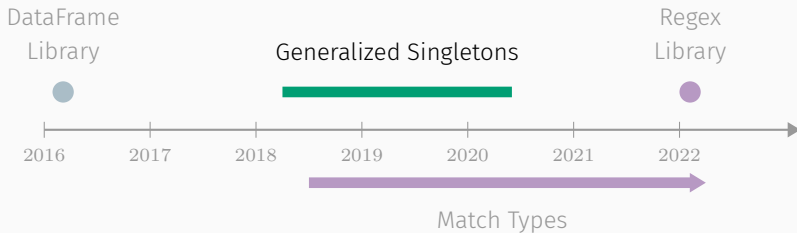


Timeline of my PhD

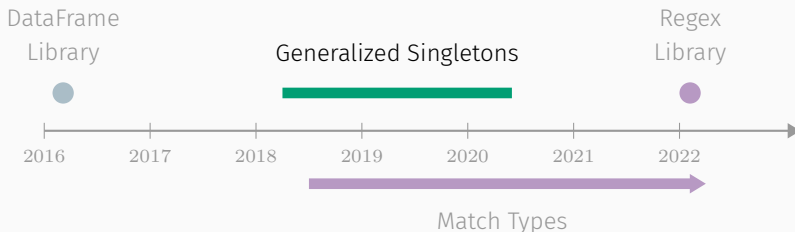


- [1] Olivier Blanvillain. “Type-Safe Regular Expressions”. In: Proceedings of the ACM SIGPLAN International Symposium on Scala. SCALA’22. New York, NY, USA: ACM, 2022.

Timeline of my PhD



Timeline of my PhD



- [1] Georg Stefan Schmid, Olivier Blanvillain, Jad Hamza, and Viktor Kuncak. “Coming to Terms with Your Choices: An Existential Take on Dependent Types”. In: CoRR (2020). arXiv: 2011.07653.

Singleton types

Definition

A singleton type is a type that contains exactly one value.

Scala has a few of those:

- `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 more of the language can we represent as singleton types?

Singleton types

Definition

A singleton type is a type that contains exactly one value.

Scala has a few of those:

- `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 more of the language can we represent as singleton types?

Singleton types

Definition

A singleton type is a type that contains exactly one value.

Scala has a few of those:

- `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 more of the language can we represent as singleton 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[]`, `Call[]`, `TypeTest[]`, 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[]`, `Call[]`, `TypeTest[]`, 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[]`, `Call[]`, `TypeTest[]`, 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[]`, `Call[]`, `TypeTest[]`, 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[]`, `Call[]`, `TypeTest[]`, 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[]`, `Call[]`, `TypeTest[]`, 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  
  
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:

- $A <: B$ if $\text{eval}(A) <: \text{eval}(B)$

Straightforward for if-then-else:

- $\text{eval}(\text{If}[\text{true}, A, B]) = A$
- $\text{eval}(\text{If}[\text{false}, A, B]) = B$

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

Type evaluation

During subtyping, we evaluate both sides:

- $A <: B$ if $\text{eval}(A) <: \text{eval}(B)$

Straightforward for if-then-else:

- $\text{eval}(\text{If}[\text{true}, A, B]) = A$
- $\text{eval}(\text{If}[\text{false}, A, B]) = B$

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

Type evaluation

During subtyping, we evaluate both sides:

- $A <: B$ if $\text{eval}(A) <: \text{eval}(B)$

Straightforward for if-then-else:

- $\text{eval}(\text{If}[\text{true}, A, B]) = A$
- $\text{eval}(\text{If}[\text{false}, A, B]) = B$

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

Type evaluation: type tests

We evaluate type tests using subtyping and type disjointness:

- $\text{eval}(x.\text{isInstanceOf}[T]) = \text{true}$ if $x.\text{type}$ is a subtype of T
- $\text{eval}(x.\text{isInstanceOf}[T]) = \text{false}$ if $x.\text{type}$ and T are disjoint
- $\text{eval}(x.\text{isInstanceOf}[T]) = x.\text{isInstanceOf}[T]$ otherwise

Type evaluation: pattern matching example

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

```
foo(42): { 43 }
```

```
foo(readInt()): { (_: Int) + 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
```

```
foo(42): { 43 }
```

```
foo(readInt()): { (_: 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): { 43 }
```

```
foo(readInt()): { (_: 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): { 43 }  
  
foo(readInt()): { (_: Int) + 1 }
```

Type evaluation: pattern matching example

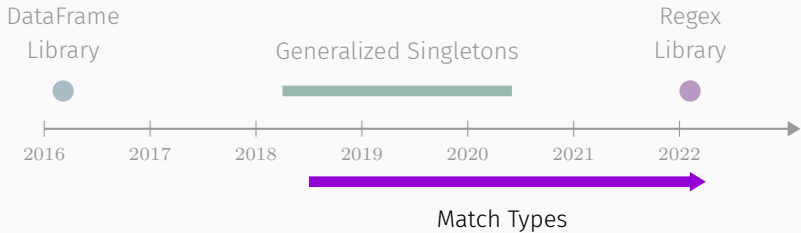
```
dependent val foo(x: Any): {  
  if (x.isInstanceOf[String]) x.asInstanceOf[String]  
  else if (x.isInstanceOf[Int]) x.asInstanceOf[Int] + 1  
  else throw new MatchError()  
}  
=  
x match  
  case s: String => s  
  case i: Int => i+1  
  
foo(42): { 43 }  
  
foo(readInt()): { (_: Int) + 1 }
```

Generalized singletons: recap

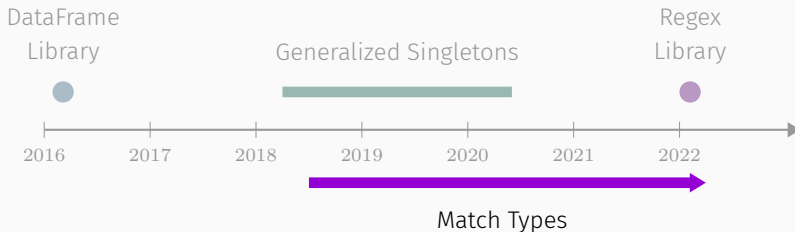
We proposed a generalization of Scala's singleton types:

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

Part II: Match Types



Part II: Match Types



- [1] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. “Type-Level Programming with Match Types”. In: Proc. ACM Program. Lang. POPL’22. New York, NY, USA: ACM, 2022.

Pattern matching on types

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

Pattern matching on types

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

Some examples of reduction:

- `Elem[String] ::= Char`
- `Elem[Int] ::= Int`
- `Elem[List[Int]] ::= Int`
- `Elem[Any] ::= Elem[Any]`

Pattern matching on types

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

We evaluate type tests using subtyping and type disjointness:

- `eval(X.isInstanceOf[T]) = true` if `X` is a subtype of `T`
- `eval(X.isInstanceOf[T]) = false` if `X` and `T` are disjoint
- `eval(X.isInstanceOf[T]) = X.isInstanceOf[T]` otherwise

Pattern matching on types

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

Match types can be explained in terms of generalized singletons:

```
type Elem[X] = { (_: X) match
  case _: String => (_: Char)
  case _: List[t] => (_: Elem[t])
  case _: Any => (_: X)
}
```

Pattern matching on types

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

The type/term correspondence dictates match type semantics:

```
def elem[X](x: X): Elem[X] = x match
  case x: String => x.charAt(0)
  case x: List[t] => elem(x.head)
  case x: Any => x
```

Pattern matching on types

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

The type/term correspondence dictates match type semantics:

```
def elem[X](x: X): Elem[X] = x match
  case x: String => x.charAt(0)
  case x: List[t] => elem(x.head)
  case x: Any => x
```

```
elem[Any]("foo")
```

System FM is an extension of System F_{\leq} with

- classes (defined externally)
- pattern matching
- match types

Design goals:

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

System FM is an extension of System F_{\leq} with

- classes (defined externally)
- pattern matching
- 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:

$\text{FM}(\mathcal{C}, \Psi, \Xi)$

- $\mathcal{C} :=$ set of classes
- $\Psi :=$ class inheritance relation
- $\Xi :=$ class disjointness relation

For example:

<code>class A; class B extends A</code>	$\mathcal{C} = \{A, B\}$	$\Psi = \{(B, A)\}$	$\Xi = \emptyset$
<code>class A; class D</code>	$\mathcal{C} = \{A, D\}$	$\Psi = \emptyset$	$\Xi = \{(A, D)\}$
<code>class A; trait T</code>	$\mathcal{C} = \{A, T\}$	$\Psi = \emptyset$	$\Xi = \emptyset$

System FM is parametric

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

$FM(C, \Psi, \Xi)$

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

For example:

<code>class A; class B extends A</code>	$C = \{A, B\}$	$\Psi = \{(B, A)\}$	$\Xi = \emptyset$
<code>class A; class D</code>	$C = \{A, D\}$	$\Psi = \emptyset$	$\Xi = \{(A, D)\}$
<code>class A; trait T</code>	$C = \{A, T\}$	$\Psi = \emptyset$	$\Xi = \emptyset$

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*

$\text{new } C$ *constructor call*

$t\ \text{match}\{\overline{x:C \Rightarrow t}\}\ \text{or}\ t$ *match expr.*

$v ::=$

$\lambda x:T. t$ *abstraction*

$\lambda X<:T. t$ *type abstraction*

$\text{new } C$ *constructor call*

$T ::=$

X *type variable*

$T \rightarrow T$ *type of functions*

$\forall X<:T. T$ *universal type*

Top *maximum type*

C *class*

$\{\text{new } C\}$ *constructor singleton*

$T\ \text{match}\{\overline{T \Rightarrow T}\}\ \text{or}\ T$ *match type*

$\Gamma ::=$

\emptyset *empty context*

$\Gamma, x:T$ *term binding*

$\Gamma, X<:T$ *type binding*

System FM's disjointness relation

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

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

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

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

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

System FM's evaluation rules

We evaluate term-level matches by querying the class inheritance relation:

$$\frac{(C, C_n) \in \Psi \quad \forall m < n. (C, C_m) \notin \Psi}{\text{new } C \text{ match}\{x_i : C_i \Rightarrow t_i\} \text{ or } t_d \longrightarrow [x_n \mapsto \text{new } C] t_n}$$

System FM's evaluation rules

We evaluate term-level matches by querying the class inheritance relation:

$$\frac{(C, C_n) \in \Psi \quad \forall m < n. (C, C_m) \notin \Psi}{new\ C \ match\{x_i : C_i \Rightarrow t_i\} \text{ or } t_d \longrightarrow [x_n \mapsto new\ C]t_n}$$

We evaluate type-level matches using subtyping and disjointness:

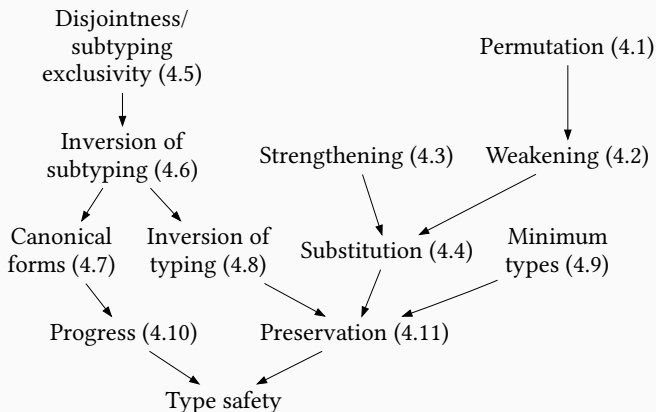
$$\frac{\Gamma \vdash T_s <: S_n \quad \forall m < n. \Gamma \vdash \text{disj}(T_s, S_m)}{\Gamma \vdash T_s \ match\{S_i \Rightarrow T_i\} \text{ or } T_d ::= T_n}$$

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



Formalization vs implementation

Our implementation goes much further than System FM:

- recursion / non-termination
- type parameter bindings (`case List[t]`)
- empty types
- variance

Empty types

Definition

An empty type is a type that contains no values.

Our system does not like empty types.

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

```
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

Definition

An empty type is a type that contains no values.

Our system does not like empty types.

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

```
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

Definition

An empty type is a type that contains no values.

Our system does not like empty types.

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

```
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

Definitions

F is covariant in T, written $F[+T]$, if $T1 <: T2$ implies $F[T1] <: F[T2]$.

G is contravariant in T, $G[-T]$, if $T1 <: T2$ implies $G[T1] >: G[T2]$.

We cannot prove disjointness when variance is involved, $F[T1]$ and $F[T2]$ always overlap:

$\forall T1, T2. \quad F[Nothing] <: F[T1] \quad \text{and} \quad F[Nothing] <: F[T2]$

$\forall T1, T2. \quad G[Any] <: G[T1] \quad \text{and} \quad G[Any] <: G[T2]$

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

Definitions

F is covariant in T, written $F[+T]$, if $T1 <: T2$ implies $F[T1] <: F[T2]$.

G is contravariant in T, $G[-T]$, if $T1 <: T2$ implies $G[T1] >: G[T2]$.

We cannot prove disjointness when variance is involved, $F[T1]$ and $F[T2]$ always overlap:

$\forall T1, T2. \quad F[\text{Nothing}] <: F[T1] \quad \text{and} \quad F[\text{Nothing}] <: F[T2]$

$\forall T1, T2. \quad G[\text{Any}] <: G[T1] \quad \text{and} \quad G[\text{Any}] <: G[T2]$

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

Definitions

F is covariant in T , written $F[+T]$, if $T1 <: T2$ implies $F[T1] <: F[T2]$.

G is contravariant in T , $G[-T]$, if $T1 <: T2$ implies $G[T1] >: G[T2]$.

We cannot prove disjointness when variance is involved, $F[T1]$ and $F[T2]$ always overlap:

$\forall T1, T2. \quad F[\text{Nothing}] <: F[T1] \quad \text{and} \quad F[\text{Nothing}] <: F[T2]$

$\forall T1, T2. \quad G[\text{Any}] <: G[T1] \quad \text{and} \quad G[\text{Any}] <: G[T2]$

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: recap

We presented match types, a new language construct for type-level programming:

- implemented in the Scala 3 compiler
- formalized in an extension of System F_{\leq}
- already in active use

Thank you!

