Abstractions for Type-Level Programming

THÈSE N. TBD (2022)

PRÉSENTÉE LE 8 FÉVRIER 2022
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
pour l'obtention du grade de Docteur ès Sciences
par

Olivier Blanvillain

acceptée sur proposition du jury :

Prof Name Surname, président du jury
Prof Name Surname, directeur de thèse
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur

Lausanne, EPFL, 2022

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Lausanne, February 8, 2022 Olivier Blanvillain

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Résumé

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Contents

List of Figures

1 Introduction

In March 2017, our research group went on a ski retreat in the Swiss Alps. After a full day of skiing on the Diablerets massif, we gathered for a lab dinner. Denys Shabalin, who was working on Scala Native at the time [Shabalin, 2020], started a conversation about manual memory management. The discussion revolved around the following question: could a Rust-like ownership system be viable for Scala? Denys' answer was clear: ownership is fundamentally at odds with the way Scala handles references, and without making deep changes to Scala's type system, the task was simply impossible.

The next day, I came up with a toy domain-specific language (DSL) that implements the basis of a linear type system using type-level programming. Here is an example of a short program written in this DSL:

```scala
def main(ctx: Context[HNil]): Context[HNil] =
  ctx.malloc(32, "mem")
     .malloc(1, "bool")
     .call(f)
     .deref("mem") { (m: Array[Byte]) =>
       println(new String(m))
     }
     .free("mem")
     .free("bool")
```

The type argument of `Context` is a type-level list of strings that corresponds to the memory regions allocated at each program point. Methods of `Context` use type-level programming techniques to enforce the following properties:

1. memory regions must be allocated (`malloc`) *before* they are deallocated (`free`),
2. all memory regions must be deallocated by the end of the program,
3. dereferencing (`deref`) is only allowed on previously allocated regions.

The implementation makes use of Scala's implicits to enforce these properties. While this small DSL is obviously too simplistic to be of any practical use, it demonstrates the power of type-level programming.

I was delighted with my solution! Denys, however, was not impressed. I attribute this apathy to his dislike of implicits. Despite their widespread usage, implicits are notorious for their complexity [Kvrikava et al., 2019]. In particular, using implicits for type-level computations requires carefully crafted definitions following a specific pattern. To give the reader a glimpse of this pattern, we show the definition of the `free` method on `Context`:

```
trait Context[Ps <: HList]:
  def free[V <: Singleton, Out <: HList]
    (v: V)
    (implicit ev: Remove[V, Ps, Out])
    : Context[Out]
```

This definition uses three parameter lists, one for type parameters (`V` and `Out`), one for a value parameter (`v`), and one of an implicit value parameter (`ev`). Only the second parameter list is intended to be specified at use site; the type and implicit parameters are meant to be inferred. When a user writes `.free("mem")` he only sets the `v` parameter (`v="mem"`); the compiler takes care of finding valid assignments for `V`, `Out` and `ev`. The implicit parameter of type `Remove[V,Ps,Out]` is the entry point to the world of type-level programming, it specifies how `V` (constrained by to be `v`'s type), `Ps` (defined in the class), and `Out` (unconstrained) are interrelated:

```
trait Remove[V, Ps <: HList, Out <: HList]

object Remove:
  implicit def casehead[V, Ps <: HList]
    : Remove[V, V :: Ps, Ps] = new Remove {}

  implicit def casetail[V, Ph, Pt <: HList, Out <: HList]
    (implicit ev: Remove[V, Pt, Out])
    : Remove[V, Ph :: Pt, Ph :: Out] = new Remove {}
```

Implicit-prefixed definitions can be understood as *facts* and *rules* of a logic program. The first definition, `casehead`, specifies a fact: the result of removing `V` from the list `V::Ps` is `Ps`, which is expressed as an instance of `Remove[V,V::Ps,Ps]`. The second definition, `casetail`, specifies a rule: if the result of removing `V` from the list `Pt` is `Out`, the results of removing `V` from the list `Ph::Pt` is `Ph::Out` (Section 2.2 develops this example in more details). When a user writes `ctx.free("mem")` on a `ctx` of type `Context["bool"::"mem"::HNil]`, the compiler uses `casehead` and `casetail` to compute a type `Out` such that `Out` is the results of removing `"mem"` from `"bool"::"mem"::HNil`.

In retrospect, I have to agree with Denys' judgment at the time: this style of programming is convoluted, to say the least. Aesthetics and pragmatism aside, programming with implicits requires a complete paradigm shift. Instead of using pattern matching and functions, implicits require algorithms to be expressed using relations and constraints, which makes the task harder than it should be.

Can we do better? This is the question that motivates the work presented in this dissertation.

Our contributions are as follows:

- In Chapter 2, we present techniques for programming at the type-level with implicits. In particular, we develop the example presented in the introduction, and show how to use Scala's implicit resolution mechanism to compute types, in a style that resembles logic programming.
- In Chapter 3, we propose a generalization of Scala's singleton types, whose goal is to enable type-level programming in an accessible style. Concretely, we extend Scala's type system with the ability to lift term-level programs to the type level and evaluate those programs during type-checking. We implement our system as an extension of the Scala 3 compiler, and show its practicality with a case study in which we develop a strongly-typed interface for Spark datasets.
- In Chapter 4, we introduce match types, a type-level equivalent of pattern matching. Match types integrate seamlessly into programming languages with subtyping and, despite their simplicity, offer significant additional expressiveness. We formalize match types in a self-contained calculus based on System $F_{<:}$ and prove its soundness. We demonstrate the practicality of our system by implementing match types in the Scala 3 compiler, thus making type-level programming readily available to a broad audience of programmers.
- In Chapter 5, we evaluate the performance of the various type-level programming techniques introduced in prior chapters, when confronted with large type-level programs. We show that the techniques introduced in Chapter 3 and 4 systematically outperform implicits in terms of compilation times and binary sizes.

2 (Ab)Using Implicits

Scala's implicit parameters have outgrown their roots as a simple syntactic construct to the extent that they provide basic support for type-level programming. In this chapter, we present techniques for implicit-based type-level programming in Scala. In particular, through extended examples, we show how to use Scala's implicit resolution mechanism to compute types, in a style that resembles logic programming.

Attribution

The first section of this chapter is based on the introduction to implicits from [Odersky et al., 2018], which was written in collaboration with Martin Odersky, Fengyun Liu, Aggelos Biboudis, Heather Miller and Sandro Stucki, and published in POPL'18.

Code samples for the remaining of this chapter are based on the implementation of heterogeneous lists from the Shapeless library [Sabin and Shapeless contributors, 2011–2022].

2.1 Implicit Parameters: Overview

Implicit parameters offer a convenient way to write code without the need to pass all arguments explicitly. The ability to omit function arguments gives rise to many interesting coding styles and patterns. On every call to functions with implicit parameters, the compiler looks for an implicit definition in scope to satisfy the call. So, instead of passing a parameter explicitly:

```
val modulo: Int = 3
def addm(x: Int, y: Int)(m: Int) = (x + y) % m
addm(4, 5)(modulo)
```

we can mark a set of parameters as implicit (a single parameter in this example) and let the compiler retrieve the missing argument for us. In the following example, `addm` is a method with one implicit parameter and `modulo` is an implicit definition:

```
implicit val modulo: Int = 3
def addm(x: Int, y: Int)(implicit m: Int) = (x + y) % m
addm(4, 5)
```

The process of implicit parameter discovery performed by the compiler is called *implicit* *resolution.* The resolution algorithm looks for implicits in the current scope and in the companion objects of all classes associated with the query type. In the previous example, the implicit definition is declared in the current scope. Since that definition has type `Int`, the compiler resolves the method call by passing `modulo` automatically.

The type class pattern

Implicits can be used to implement type classes [Wadler and Blott, 1989] as a design pattern [Oliveira et al., 2010]. We give an example of an implementation of the `Ordering` type class. This example consists of three parts:

1. `Ordering[T]`, which is a regular trait with a single method, `compare`,
2. the generic function `comp`, which compares two arguments and accepts an implicit argument, providing an *implicit evidence* that these two values can be compared,
3. the implicit definition `intOrdering`, which provides an *instance* of the `Ordering` type class for integers.

```
trait Ordering[T]:
  def compare(a: T, b: T): Boolean

def comp[T](x: T, y: T)(implicit ev: Ordering[T]): Boolean =
  ev.compare(x, y)

implicit def intOrdering: Ordering[Int] =
  new Ordering[Int]:
    def compare(a: Int, b: Int): Boolean = a < b

comp(1, 2)
```

We have briefly introduced implicit parameters and showed how they can be used to avoid clutter in function applications. In the next section, we present recursive implicit resolution.

2.2 Recursive Implicit Resolution

Implicit methods can themselves take implicit parameters. For example, we can define the lexicographic list ordering as follows:

```
implicit def listOrdering[T](implicit ev: Ordering[T]): Ordering[List[T]] =
  new Ordering[List[T]]:
    def compare(a: List[T], b: List[T]): Boolean =
      (a, b) match
        case (a :: as, b :: bs) => ev.compare(a, b) && compare(as, bs)
        case (_, Nil) => false
        case (Nil, _) => true
```

This definition is parametrized by the list's element type and by an ordering of that type, passed as an implicit parameter. Since `listOrdering` is itself implicit, it defines a *rule*: it allows the compiler to materialize an implicit `Ordering[List[T]]`, given an implicit `Ordering[T]` (for any type `T`).

Parametrized implicit definitions can lead to recursive implicit resolution. For example, the compiler will use `listOrdering` twice to synthesize an implicit `Ordering[List[List[T]]]`. This is where the (type-level) fun begins!

Heterogeneous lists

A heterogeneous list, or `HList` for short [Kiselyov et al., 2004], is a datatype capable of storing data of different types. In Scala 3, we can define the `HList` datatype as follows:

```
enum HList:
  case HNil()
  case ::[+H, +T <: HList](head: H, tail: T)
```

The `::` constructor offers an interesting symmetry between the term and type level, which allows `HList` types to capture the same structure that their term-level counterparts. For example, the term `::(1,::(2,HNil()))` can be typed as `::[1,::[2,HNil]]`, which is a perfect reification of that term (we use literal singleton types to represent constant literals at the type level [Leontiev et al., 2014]).

HList's remove

Scala's implicits allow us to define type-level operations for heterogeneous lists. We develop the example presented in Chapter 1 by looking into the remove operation on `HLists`. Remove operation takes as argument an element and a list, and returns that list with the first occurrence of the element removed. This operation should yield an error if the element is not part of the list.

An implicit-based type-level operation typically takes the form of a trait, the operation's entry point, and several implicit definitions, one for each "case" of the operation's algorithm. Let us consider the implementation of the remove operation in more detail:

```
trait Remove[V, Ps <: HList, Out <: HList]

object Remove:
  implicit def casehead[V, Ps <: HList]
    : Remove[V, V :: Ps, Ps] = new Remove {}

  implicit def casetail[V, Ph, Pt <: HList, Out <: HList]
    (implicit ev: Remove[V, Pt, Out])
    : Remove[V, Ph :: Pt, Ph :: Out] = new Remove {}
```

The `Remove` trait takes 3 type parameters: 2 inputs, `V` (the element to remove) and `Ps` (the list), and one output, `Out` (the list, with the element removed). The two implicit definitions, `casehead` and `casetail`, correspond to the base case and the recursive case of the list removal operation, respectively. The Remove trait is intended to be used as an implicit parameter, with constrained input types, and an unconstrained output type.

When the given element is not part of the list, implicit resolution fails with an “implicit not found” error, which indicates the incorrect use of Remove. More precisely, the implicit search first iterates through the list by repeatedly using casetail until it reaches the end of the list, at which point it fails to find an implicit value of type Remove[V,HNil,Out] (neither casehead nor casetail produce an instantiation of Remove with Ps=HNil).

As an example usage of Remove, consider the following stringly-typed, JavaScript-inspired method:

```
/** Attaches an event handler.
 *  @param event     The event type, one of "mousedown", "mouseup",
 *                   "mouseover", "mousewheel" and "contextmenu".
 *  @param listener  The function to run when the event occurs.
 */
def addEventListener(event: String, listener: Event => Unit): Unit
```

This method’s documentation specifies 5 valid alternatives for the event argument, but that constraint is not reflected in the method’s type signature. In JavaScript, calling this method with an erroneous element event type is a *no-op*, which can make this kind of error particularly hard to spot.

Instead of specifying that constraint in the documentation, we can represent the valid event types in a HList, and use an implicit parameter of type Remove to statically enforce that property:

```
type EventTypes =
  "mousedown" :: "mouseup" :: "mouseover" :: "mousewheel" :: "contextmenu" ::
    HNil

def addEventListener[E <: String & Singleton]
  (event: E, handler: Event => Unit)
  (implicit ev: Remove[E, EventTypes, ?]): Unit
```

The Singleton type bound is a marker that instructs type inference to preserve literal singleton types (by default, the compiler widens those types). The “?” type is Scala 3’s new syntax for wildcard types [Odersky and Dotty contributors, 2013–2022, Wildcard Arguments in Types].

With this updated signature, the compiler is able to detect invalid event types at compile time. When given a valid event type E, the compiler synthesizes an implicit evidence of type Remove[E,EventTypes,?] which witnesses E’s validity. After implicit resolution, calls to the addEventListener method are expanded to automatically insert an implicit parameter for the second parameter list, such as in the following example:

```
addEventListener("mouseover", myHandler)(
  // Evidence that "mouseover" is a valid event type, infered automatically.
  Remove.casetail(Remove.casetail(Remove.casehead))
)
```

This concludes our presentation of the implicit-based encoding of type-level computation. Despite its verbosity, this pattern generalizes to arbitrary recursive computations and enables Scala programmers to write elaborate type-level programs solely based on implicits. In the next section, we discuss ambiguities and priorities between implicit definitions.

2.3 Ambiguities and Priorities

Scala's implicit resolution relies on an intricate priority system to establish the precedence of implicit definitions. Implicit-based programs sometimes rely on ambiguities and priorities of implicit definition, as we will see in this section through a series of example.

2.3.1 Implicit Ambiguities

The Scala compiler rejects programs with *ambiguous* implicit definition. For instance, if we write two identically-typed implicit definitions in the same scope, the compiler will consider them ambiguous and report an error:

```
implicit val number: Int = 1
implicit val modulo: Int = 3
def addm(x: Int, y: Int)(implicit m: Int) = (x + y) % m
addm(4, 5) // Error: ambiguous implicit arguments: both value number and
           // value modulo match type Int of parameter m of method addm.
```

While ambiguities typically indicate programming errors, we can also use them purposefully to implement the error case of a type-level program. As an example, consider the `NotIn` operation on `HList` that is only defined if the given element is *not* part of the list. We define `NotIn` using two ambiguous implicits; the implementation is lengthy, but straightforward:

```
trait NotIn[V, Ps <: HList]

object NotIn:
  implicit def casenil[V]: NotIn[V, HNil] = new NotIn {}

  implicit def casecons[V, Ph, Pt <: HList]
    (implicit xs: NotIn[V, Pt]): NotIn[V, Ph :: Pt] = new NotIn {}

  implicit def ambiguous1[V, Ph, Pt <: HList]
    (implicit ev: V =:= Ph): NotIn[V, Ph :: Pt] = new NotIn {}

  implicit def ambiguous2[V, Ph, Pt <: HList]
    (implicit ev: V =:= Ph): NotIn[V, Ph :: Pt] = new NotIn {}
```

The `casenil` and `casecons` implicits simply iterate through the list. The `ambiguous` implicits are two identical definitions that will cause the compiler to raise an error if `V` and `Ph` are equal, for any element `Ph` of the list ("`=:=`" is a type from Scala's standard library that witnesses mutual subtyping).

As an example usage of `NotIn`, we revisit the DSL presented in Chapter 1. Programs in our

DSL consist of sequences of method calls on a `Context`, which tracks the memory regions that are currently allocated in the program, using a `HList` of names. We use the `NotIn` operation to constrain the allocation method, `malloc`, to prevent allocating regions with already used names:

```
trait Context[Ps <: HList]:
  def malloc[V <: Singleton]
    (size: Int, v: V)
    (implicit ev: NotIn[V, Ps])
    : Context[V :: Ps]
```

Scala 3 introduced an alternative to the ambiguous implicit pattern in the form of the `scala.util.NotGiven` type [Odersky and Dotty contributors, 2013–2022, Given Instances]. The compiler will synthesize an implicit parameter of type `NotGiven[T]`, if and only if there is no implicit value of type `T` in scope. We can use a "negative" implicit evidence to simplify the definition of `NotIn` by removing the ambiguous implicits and changing `casecons` to the following:

```
implicit def casecons[V, Ph, Pt <: HList]
  (implicit
    no: NotGiven[V =:= Ph],
    xs: NotIn[V, Pt]
  ): NotIn[V, Ph :: Pt] = new NotIn {}
```

2.3.2 Implicit Priorities

When looking for implicits, the Scala compiler visits scopes sequentially, in a precisely defined order. At any point in the search, if the implicits defined in the subset of scopes considered so far can fulfill the implicit query, the search succeeds and immediately terminates. This incremental process has two consequences. First, it allows the compiler to efficiently look for implicits by naturally pruning some of the search space. Second, it provides an adhoc mechanism to disambiguate implicits. Scala programmer can artificially partition their implicit definitions into multiple scopes to implement a priority system.

Let us consider an example of operation on `HList` whose implementation uses implicit priorities. `RemoveAll` is a generalization of `Remove` that removes every occurrence of the given element instead of the first occurrence. The implicit-based implementation takes the form of a trait with 3 type parameters: 2 inputs, `V` (the element to remove) and `Ps` (the list), and one output, `Out` (the list, with the elements removed):

```
trait RemoveAll[V, Ps <: HList, Out <: HList]
```

From an algorithmic standpoint, we can implement `RemoveAll` as a recursive function with 3 cases, a base case for the empty list (`casenil`), a recursive case for when the head of the list matches the element to remove (`casematch`), and another recursive case for when the head doesn't match (`casedoesnt`):

```
object RemoveAll:
  implicit def casenil[V]
```

```
    : RemoveAll[V, HNil, HNil] = new RemoveAll {}

  implicit def casematch[V, Ps <: HList, Out <: HList]
    (implicit ev: RemoveAll[V, Ps, Out])
    : RemoveAll[V, V :: Ps, Out] = new RemoveAll {}

  implicit def casedoesnt[V, Ph, Pt <: HList, Out <: HList]
    (implicit ev: RemoveAll[V, Pt, Out])
    : RemoveAll[V, Ph :: Pt, Ph :: Out] = new RemoveAll {}
```

This direct implementation of `RemoveAll` using implicit definitions is, unfortunately, incorrect. The issue is that the `casematch` and `casedoesnt` definitions are ambiguous when the head of the list matches the element to remove. To work around that ambiguity, we split those definitions into two different scopes, so that the compiler always tries to apply the more specialized case (`casematch`) before considering the less specialized case (`casedoesnt`). Concretely, we define low priority implicits in a separate trait, and have `RemoveAll`'s companion object extend that trait:

```
trait RemoveAllLowPrio:
  implicit def casenil[V]
    : RemoveAll[V, HNil, HNil] = new RemoveAll {}

  implicit def casedoesnt[V, Ph, Pt <: HList, Out <: HList]
    (implicit ev: RemoveAll[V, Pt, Out])
    : RemoveAll[V, Ph :: Pt, Ph :: Out] = new RemoveAll {}

object RemoveAll extends RemoveAllLowPrio:
  implicit def casematch[V, Ps <: HList, Out <: HList]
    (implicit ev: RemoveAll[V, Ps, Out])
    : RemoveAll[V, V :: Ps, Out] = new RemoveAll {}
```

2.4 Conclusion

In this chapter, we presented several techniques for type-level programming with implicits. This style of programming is, unfortunately, quite cumbersome. In addition to the heavy syntax, type-level programming with implicits also requires a deep understanding of the implicit resolution algorithm. As we will see in Chapter 5, those techniques come at a high cost in terms of compilation time, which hinders their usability on a large scale. Yet, despite those shortcomings, Scala programmers have shown a persistent interest in this style of programming, as demonstrated by its popularity in the open-source community [Sabin and Shapeless contributors, 2011–2022; Pilquist and Scodec contributors, 2013–2022; Blanvillain et al., 2016–2022].

3 Generalizing Scala's Singleton Types

Type-level programming is an increasingly popular way to obtain additional type safety. Unfortunately, it remains a second-class citizen in the majority of industrially-used programming languages. We propose a new dependently-typed system with subtyping and singleton types whose goal is enabling type-level programming in an accessible style. To this end, we have prototyped our system as an extension of the Scala programming language. We demonstrate the practicality of our system with a case study in which we develop a strongly-typed interface for Spark datasets. Through our formalization and implementation in the context of an industrial-strength compiler, we hope to provide valuable insights for language designers interested in dependent types.

Attribution

This chapter is based on [Schmid et al., 2020], which was written in collaboration with Georg Schmid, Jad Hamza, and Viktor Kuncak. Georg and I worked hand-in-hand on this project: we collaborated through countless whiteboard discussions and pair programming sessions, which resulted in shared first authorship of the implementation and most of the text. The said report covers more material than what is included in this chapter, in particular, our formalization and the associated metatheory are out of the scope of this dissertation.

3.1 Introduction

Dependent types have been met with considerable interest from the research community in recent years. Their primary application so far has been in proof assistants such as Coq [Bertot and Castéran, 2004] and Agda [Norell, 2007], where they provide a sound and expressive foundation for theorem proving. However, dependent types are still largely absent from general-purpose programming languages, despite a long history of lightweight approaches [Xi and Pfenning, 1998]. In the context of Haskell, much research has gone into extending the language to support computations on types, for instance in the form of functional dependencies [Jones, 2000], type families [Kiselyov et al., 2010] and promoted datatypes [Yorgey et al., 2012]. These techniques have seen vivid adoption by Haskell programmers, showing that there is a real demand for such mechanisms. Furthermore, recent research has explored how dependent types could be added to the language for the same purpose [Eisenberg, 2016;

Weirich et al., 2017].

Dependently-typed languages often rely on a unified syntax to describe both terms and types. The simplicity of this approach is unfortunately at odds with the design of most programming languages, where types and terms are expressed using separate syntactic categories. Singleton types provide a simple solution to this problem by allowing terms to be represented as types.

In this chapter, we report on our attempt to generalize Scala's singleton types to support type-level programming, as well as a lightweight form of dependently-typed programming. Unlike proof assistants, we do not aim to use types as a general-purpose logic, which would favor designs ensuring the totality of functions through termination checks. Instead, our focus is on improving type safety by increasing the expressive power of the type system.

With these goals in mind, we extended Scala's type system to lift programs to the type level and partially evaluate them as part of type-checking. Users can manipulate those types either directly, using new syntactic forms in Scala's type language, or have them inferred automatically using a new language keyword. This effectively allows users to execute programs involving functions and pattern matching at the type level.

The remaining of the chapter is organized as follows:

- We begin by motivating why type-level programming is desirable and how one might use our Scala extension to improve type safety (Section 3.2). Our example demonstrates how to design a strongly-typed API in a functional style accessible to programmers.

- We describe how we extended Scala with a generalization of singleton types (Section 3.3). We prototyped our type system on top of Dotty, the reference Scala 3 compiler. This practical introduction would be of interest to any Scala programmer willing to learn how to use our system, as well as programming language designers interested in dependent types.

- We show a concrete use-case of our system by implementing a strongly-typed wrapper for Apache Spark [Zaharia et al., 2016] (Section 3.4). Thanks to our generalized singleton types, we can statically ensure the type safety of database operations such as join and filter. We compare our implementation with an equivalent implicit-based one and show remarkable compilation time savings.

Our original presentation of generalized singleton types contains a formalization [Schmid et al., 2020, Section 3 and 4], which is out of the scope of this chapter.

3.2 Motivating Examples

We begin by motivating why type-level programming is desirable in general purpose programming. In our first example, we design an API that keeps track of database tables' schemas in the type, and uses that information to improve type safety. The examples in this section are written in our Scala extension described in Section 3.3.

3.2.1 Safe Join

As a first step, we show how our system supports type-level programming in the style of term-level programs. Consider the following definition of the list datatype, which is standard Scala up the `dependent` keyword:

```
sealed trait Lst { … }
dependent case class Cons(head: Any, tail: Lst) extends Lst
dependent case class Nil() extends Lst
```

We can define list concatenation in the usual functional style of Scala, that is, using pattern matching and recursion:

```
sealed trait Lst:
  dependent def concat(that: Lst) <: Lst =
    this match
      case Cons(x, xs) => Cons(x, concat(xs, that))
      case Nil() => that
```

By annotating a method as `dependent`, the user instructs our system that the result type of `concat` should be as precise as its implementation. Effectively, this means that the body of `concat` is lifted to the type level in a singleton type, and will be partially evaluated at every call site to compute a precise result type which *depends* on the given inputs. For recursive `dependent` methods such as `concat`, we infer types that include calls to `concat` itself. The `<:` annotation lets us provide an upper bound on `concat`'s result type, which will be used while type checking the method's definition. Finally, by qualifying the definition of `Cons` and `Nil` as `dependent`, we also allow their constructors and extractors to be lifted to the type level. Using these definitions, we can now request the precise type whenever we manipulate lists by annotating `val` bindings as `dependent`:

```
dependent val l1 = Cons("A", Nil())
dependent val l2 = Cons("B", Nil())
dependent val l3 = l1.concat(l2)
l3.size: { 2 }
l3: { Cons("A", Cons("B", Nil())) }
```

Enclosing a pure term in braces (`{ ... }`) denotes the singleton type of that term. In the last two lines of this example, we are therefore asking our system to prove that `l3` has size 2 and is equivalent to `Cons("A", Cons("B", Nil()))`. Similarly, we can define remove on `Lst`:

```
sealed trait Lst:
  dependent def remove(e: String) <: Lst =
    this match
      case Cons(head, tail) =>
        if (e == head) tail
        else Cons(head, tail.remove(e))
      case _ => throw new Error("element not found")
```

Removing `"B"` yields the expected result, while trying to remove `"C"` from `l3` leads to a *compilation error*, since the given program will provably fail at runtime.

```
l3.remove("B"): { Cons("A", Nil()) }
l3.remove("C") // Error: element not found
```

The lists we defined so far can be used to implement a type-safe interface for database tables:

```
dependent case class Table(schema: Lst, data: spark.DataFrame):
  dependent def join(right: Table, col: String) <: Table =
    val s1 = this.schema.remove(col)
    val s2 = right.schema.remove(col)
    val newSchema = Cons(col, s1.concat(s2))
    val newData = this.data.join(right.data, col)
    new Table(newSchema, newData)
```

In this example, we wrap a Spark's `DataFrame` in the **`dependent`** class `Table`. The first argument of this class represents the schema of the table as a precisely-typed list. The second argument is the underlying `DataFrame`. In the implementation of `join`, we execute the join operation on the underlying tables (`newData`) and compute the resulting schema corresponding to that join (`newSchema`). By annotating the `join` method as **`dependent`**, the resulting schema is reflected in the type:

```
dependent val schema1 = Cons("age", Cons("name", Nil()))
dependent val schema2 = Cons("name", Cons("unit", Nil()))
dependent val table1  = Table(schema1, …)
dependent val table2  = Table(schema2, …)
dependent val joined  = table1.join(table2, "name")
joined: { Table(Cons("name", Cons("age", Cons("unit", Nil()))), _: DataFrame) }
```

Reflecting table schemas in types increases type safety over the existing weakly-typed interface. For instance, it becomes possible to raise compile-time errors when a user tries to use non-existent columns. This is an improvement over the underlying Spark implementation that would instead fail at runtime.

3.3 Implementation

In this section we give an overview of how we extended Scala with a generalized notion of singleton types, and how we manipulate these types during type-checking. Our presentation is divided into three parts.

First, we introduce new types that reflect a subset of Scala's term language at the type level, and give an informal description of our type evaluation algorithm. We then introduce the **`dependent`** qualifier, which influences type inference to assign generalized singleton types to the annotated definition. Finally, we discuss how our extension interacts with other aspects of Scala, such as side effects, virtual dispatch, and recursion.

This development was an experiment to explore the feasibility of adding a lightweight form of dependent types to Scala. We implemented our prototype as an extension of Dotty, the reference Scala 3 compiler.

3.3.1 Reflecting Terms in Types

The fundamental difference between our and Scala's existing type system is that we take Scala's notion of singleton types, that is, precise types for variable bindings and literals, and extend them to cover Scala's core functional expressions. Concretely we add new types for the following constructs:

- Variable bindings and member selections
- Primitive literals
- Method calls
- If-then-else expressions
- Pattern matching expressions
- Constructor calls

These new types can be expressed using the `{` *e* `}` type syntax, where *e* is an expression in the core functional subset of Scala listed above. For instance, suppose foo stands for the expression

```
if (x % 2 == 0) "even" else "odd"
```

then foo can be typed as `{` foo `}`. More precisely, `{` foo `}` is the singleton type of foo and corresponds to the unique value of foo in a given context.

The first two constructs in this list have antecedents in Scala. Types for bindings are available since the early days of Scala with the `x.type` syntax [Odersky et al., 2006–2022, Section 3.2.1], which is equivalent to `{ x }` in our system. Types for primitive literals have recently been added to the language: literals for booleans, strings and the various numeric types are made available in the type language [Leontiev et al., 2014].

In addition to base types and singleton types for every pure term, our system also supports types that lie in-between. We add a new form, `_: T`, which supplements expressions in types of form `{` *e* `}` by allowing the user to intersperse terms and base types.

For instance, consider the following list and its singleton type:

- `Cons(Nil, Nil): { Cons(Nil, Nil) }`

In our system, this term may also be typed less precisely, such as:

- `Cons(Nil, Nil): { Cons(_: Any, Nil) }`
- `Cons(Nil, Nil): { Cons(Nil, _: Lst) }`
- `Cons(Nil, Nil): { Cons(_: Any, _: Lst) }`

all of which are subsumed by

- `Cons(Nil, Nil): Lst.`

3.3.2 Type Evaluation

Our system evaluates types using a call-by-value partial evaluator, which we embedded in the type-checker. The evaluation rules for the term in generalized singleton types are completely standard up to dynamic type tests.

We perform evaluation of types during subtyping. For instance, consider the following definition of a `parity` function:

```
dependent def parity(x: Int) =
  if (x % 2 == 0) "even" else "odd"
```

In order to prove the well-typedness of

```
val p: { "odd" } = parity(5)
```

the type-checker will ensure that `{ parity(5) }` is a subtype of `{ "odd" }`. In the process, the left-hand side is evaluated as follows:

```
  { parity(5) }                          (1)
= { if (5 % 2 == 0) "even" else "odd" }  (2)
= { if (false) "even" else "odd" }       (3)
= { "odd" }
```

In (1), the method call to `parity` is replaced by its result type, where the method parameter has been substituted by the singleton type of the argument, `{ 5 }`. (2) evaluates the boolean expression to `false`. (3) reduces the `if` expression to its else branch.

In general, our evaluator will execute operations on concrete primitive values of types such as `Boolean`, `Int` and `String`, i.e., perform constant-folding.

3.3.3 Pattern Matching

Pattern matching in Scala supports a wide range of matching techniques [Emir et al., 2007]. For example, *extractor patterns* rely on user-defined methods to extract values from objects. As a result, these custom extractors can contain arbitrary side effects. Our implementation limits the kind of patterns available in types to the two simplest forms: decomposition of case classes and the type-tests/type-casts patterns.

During type normalization, our system evaluates pattern matching expressions according to Scala's runtime semantics. Patterns are checked top-to-bottom, and type-tests are evaluated using runtime type information available after type erasure.

For example, consider the following pattern matching expression:

```
s match {
  case _: T1 => v1
  case _: T2 => v2
}
```

When used in a type, this expression reduces to `v1` if the scrutinee's type is a subtype of `T1`. In order to reduce to `v2`, type normalization must make sure `T1` and the scrutinee's type are disjoint, namely that the dynamic type of `s` cannot possibly be smaller than `T1`. Disjointness proofs are built using static knowledge about the class hierarchy, such as the `sealed` and `final` qualifiers, which are Scala's way of declaring closed-type hierarchies.

3.3.4 Two Modes of Type Inference

In order to retain backwards-compatibility, our system supports two modes of type inference: the precise inference mode which infers singleton types, and the default inference mode that corresponds to Scala's type-inference algorithm. Concretely, users opt into our new inference mode using the `dependent` qualifier on methods, values, and classes.

When inferring the result type of a `dependent` method, our system lifts the method's body into a generalized singleton type. This lifting will be precise for the subset of expressions that is representable in types, and approximative for the rest. When we encounter an unsupported construct, we compute its type using the default mode, yielding a type `T` which we then integrate in the lifted body as `_: T`.

For example, given the following definition:

```
dependent def getName(personalized: Boolean) =
  if (personalized) readString() else "Joe"
```

our system infers the following result type:

```
{ if (personalized) (_: String) else "Joe" }
```

We could have equivalently defined `getName` by omitting the `dependent` qualifier and writing its result type explicitly.

Scala requires recursive methods to have an explicit result type, and this restriction also applies to `dependent` methods. However, in the case of a `dependent` method, an explicit result type is only used as an upper bound for the actual precise result type and will only be used to type-check the method's body. At other call sites, the (precise) inferred result type is used. Bounds of dependent methods are written using a special syntax (`<: T`).

3.3.5 Approximating Side Effects

Scala's type system permits uncontrolled side effects in programs. Given the absence of an effect system, result types of methods do not convey any information about the potential use of side effects in the method body. The situation is analogous for `dependent` methods. Since we uniformly approximate all side effects, we avoid the situation where a type refers to a value that may be modified during the program execution. For instance, if `z` is a mutable integer variable, we will never introduce `z` in a singleton type. However, we can still assign singleton types to expressions containing `z`, for example, we can type `Cons(z, Nil())` as `{ Cons(_: Int, Nil()) }`.

Similarly to how we model other side effects, exceptions are approximated in types. Our type-inference algorithm uses a new error type, `Error(e)`, which we infer when raising an exception with `throw e`. Exception handlers are typed imprecisely using the default mode of type-inference. Exceptions thrown in statement positions are not reflected in singleton types, since the type of `{e1; e2}` is simply `{ e2 }`. However, exceptions thrown in tail positions (such as in remove from Section 3.2) can lead to types normalizing to `Error(e)`. In these cases, our type system can prove that the program execution will encounter exceptional behavior, and report a compilation error. This approach is conservative in that it might reject programs that recover from exceptions. Also note that this is a sanity check, rather than a guarantee of no exceptions occurring at runtime. That is, depending on which rules are used during subtyping, it is possible to succeed without entering type normalization, resulting in such errors going undetected. Despite these shortcomings, our treatment of exceptions results in a practical way to raise compile-time errors. It would be interesting to explore the addition of an effect system to our Scala extension.

3.3.6 Virtual Dispatch

Our extension does not model virtual dispatch explicitly in singleton types. Instead, the result type of a method call `t.m(...)` is always the result type of `m` in `t`'s static type. Consequently, **`dependent`** methods effectively become **`final`**, given that only a provably-equivalent implementation could be used to override it.

Special care must be taken when an imprecisely-typed method is overridden with a dependent one. In this situation, the result type of a method invocation can lose precision depending on type of the receiver. Calls to the `equals` methods are a common example of this: `equals` is defined at the top of Scala's type hierarchy as referential equality and can be overridden arbitrarily. Given a class Foo with a **`dependent`** overrides of `equals`, calls to `Foo.equals(Any)` and `Any.equals(Foo)` are not equivalent; the former precisely reflects the equality defined in `Foo` whereas the latter merely returns a `Boolean`.

3.3.7 Termination

We distinguish two important aspects of termination: the termination of type-checked programs and the termination of our type checker.

Proving termination or totality of programs is a non-goal of our system. Unlike proof assistants, Scala programs do not manipulate proof terms. Consequently, the lack of totality checks does not affect Scala's present notion of safety. Exceptions or infinite loops in the evaluation of a 'dependent' method would prevent the completion of type-checking.

The second question is termination of our type checker. Non-termination of type checking implies that the type checker can give three possible answers, "type correct", "type incorrect" or "do not know" (when type checking times out). Treating "do not know" as "type incorrect" makes the non-termination unproblematic from a soundness perspective. A similar argument is made for other dependently-typed languages with unbounded recursion, such as Dependent Haskell [Eisenberg, 2016] or Cayenne [Augustsson, 1998]. In practice, our system deals with infinite loops using a fuel mechanism. Every evaluation step consumes a unit of fuel, and an error is reported when the compiler runs out of fuel. The default fuel limit can be increased via a compiler flag to enable arbitrarily long compilation times.

3.4 Use Case

In this section, we extend the motivating example presented in Section 3.2 by building a type-safe interface for Spark datasets. We use our Scala extension to implement a simple domain-specific type checker for the SQL-like expressions used in Spark.

3.4.1 A Type-Safe Database Interface

The type-safe interface presented in this section illustrates the expressive power of our system and is implemented purely as a library. For brevity, our presentation only covers a small part of Spark's dataset interface, but the approach can be scaled to cover that interface in its entirety. The type safety of database queries is a canonical example and has been studied in many different settings [Leijen and Meijer, 1999; Kazerounian et al., 2019; Meijer et al., 2006; Chlipala, 2010].

The example built in Section 3.2 uses lists of column names to represent schemas. A straightforward improvement is to also track the type of columns as part of the schema. Instead of using column names directly, we introduce the following `Column` class with a phantom type parameter `T` for the column type, and a field `name` for the column name:

```
dependent case class Column[T](name: String) { … }
```

Table schemas become lists of `Column`-s and thereby gain precision. The definition of `join` given in Section 3.2 can be adapted to this new schema encoding to prevent joining two tables that have columns with matching names but different types.

A large proportion of the weakly-typed Spark interface is dedicated to building expressions on table columns. Such expressions can currently be built from strings, in a subset of SQL, or using a Scala DSL which is essentially untyped.

The lack of type safety for column expressions can be particularly dangerous when mixing columns of different types. The pitfall is caused by Spark's inconsistency: depending on types of columns and operations involved, programs will either crash at runtime, or, more dangerously, data will be silently converted from one type to another.

By keeping track of column types it becomes possible to enforce the well-typedness of column expressions. As an example, consider the following Spark program:

```
table.filter(table.col("a") + table.col("b") === table.col("c"))
```

We would like our interface to enforce the following safety properties:

- Columns a, b and c are part of the schema of `table`.
- Addition is well-defined on columns a and b.
- The result of adding columns a and b can be compared with column c.
- The overall column expression yields a `Boolean`, which conforms to filter's argument type.

Automatic conversions during equality checks can be prevented by restricting column equality to expressions of the same type T:

```
dependent case class Column[T](k: String):
  def ===(that: Column[T]): Column[Boolean] = Column(s"(${this.k} === ${that.k})")
```

Addition in Spark is defined between numeric types and characters. The result type of an addition depends on the operand types. For numeric types, Spark will pick the larger of the operand types according to the following ordering: Double > Long > Int > Byte. The situation is quite surprising with characters as any addition involving a Char will result in a Double.

Dependent types can be used to precisely model these conversions. We define a type function to compute the result type of additions:

```
def addRes(a: Any, b: Any) =
  (a, b) match
    case (_: Char, _: Char | Byte | Int  | Long | Double) => _: Double
    case (_: Byte, _: Byte | Int  | Long | Double)        => b
    case (_: Int,  _: Int  | Long | Double)               => b
    case (_: Long, _: Long | Double)                      => b
    case (_: Double, _: Double)                           => _: Double
    case (_: Byte | Int | Long | Double, _)               => addRes(b, a)
    case _ => throw new Error("incompatible types in addition")
type AddRes[A, B] = { addRes(_: A, _: B) }
```

Also note the use of recursion in the second-to-last case, to avoid duplicating symmetric cases. The AddRes type can be used to define a Column addition that accurately models Spark's runtime:

```
dependent case class Column[T] private (k: String):
  dependent def +[U](that: Column[U]) <: Column[_] =
    Column[AddRes[T, U]](s"(${this.k} + ${that.k})")
```

Allowing programmers to construct Column-s from string literals would defeat the purpose of a type-safe interface. Instead, programmers should extract columns from a Table's schema. For that purpose, we implement the col method on Table and annotate the Column constructor as private.

```
dependent case class Table(schema: Lst, data: spark.DataFrame):
  dependent def col(name: String) <: Column[_] =
    dependent def find(key: String, list: Lst) <: Any =
      list match
        case Cons(head: Column[_], tail) =>
          if (head.k == key) head else find(key, tail)
        case _ => throw new Error("column not found in schema")
    find(name, schema)
  dependent def filter(predicate: Column[Boolean]) <: Table =
    new Table(this.schema, this.data.filter(predicate.k))
```

The col method is implemented using a nested dependent method to find the column

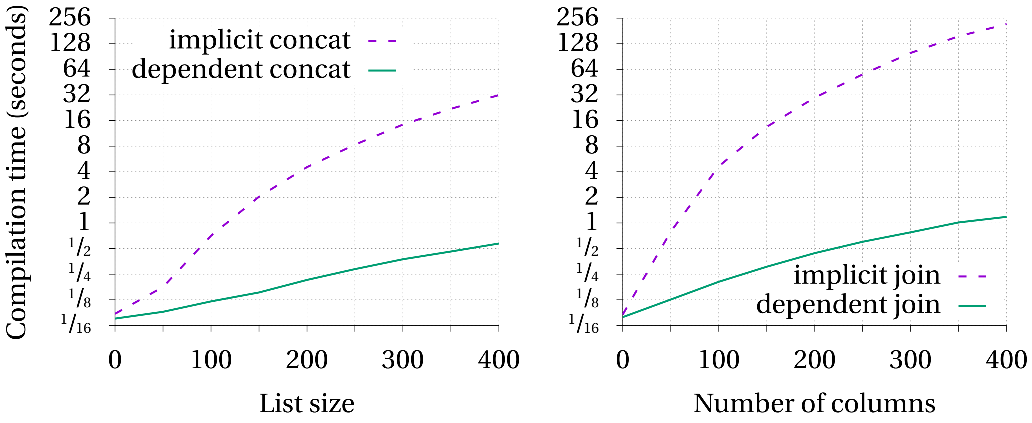

Figure 3.1: Comparing the compilation times of two implementations of list concatenation and join, logarithmic scale.

corresponding to the given name. Thanks to the dependent annotation, the type-checker is able to statically evaluate calls to `col`. Assuming the table's schema contains a column `a` of type `Int` and columns `b` and `c` of type `Long`, the compiler will be able to infer types as follows:

```
val pred =    table.col("a")     +    table.col("b")     ===    table.col("c")
// Infers: { Column[Int]("a") }   { Column[Long]("b") }   { Column[Long]("c") }
```

Given our definitions of column addition and equality, the overall `pred` expression is typed as `Column[Boolean]`. Thus, the dependently-typed interface presented in this section successfully enforces all the safety properties stated above.

3.4.2 Comparison to an Existing Technique

Programmers have managed to find clever encodings that circumvent the lack of first-class support for type-level programming in many languages. These encodings can be very cumbersome, as they often entail poor error reporting and a negative impact on compilation times [McBride, 2002], [Kiselyov et al., 2004]. In Scala, implicits are the primary mechanism by which programmers implement type-level programming [Odersky et al., 2018].

Frameless [Blanvillain et al., 2016–2022] is a Scala library that implements a type-safe interface for Spark by making heavy use of implicits. Most type-level computations in this library are performed on the heterogeneous lists provided by Shapeless [Sabin and Shapeless contributors, 2011–2022].

We compared the dependently-typed Spark interface presented in this section against the implicit-based implementation of Frameless. To do so, we isolated the implicit-based implementation of the `join` operation on table schemas, and compared its compilation time against the dependently-typed version presented in this section. To evaluate the scalability of both approaches we generated test cases with varying schema sizes and compiled each test case in isolation. A similar comparison is done for list concatenation, which constitutes a building block of `join`.

Figure 3.1 shows that, in both benchmarks, the dependently-typed implementation compiles faster than the version with implicits, and compilation time scales better with the size of the input.

In the join benchmark, we see that the implicit-based implementation exceeds 30 seconds of compilation time around the 200 columns mark, and continues to grow quadratically. This can be explained by the nature of implicit resolution, which might backtrack during its search. The compilation time of the dependently-typed implementation grows linearly and stays below one second until the 350 columns mark. We were able to observe similar trends in the concatenation benchmark. These measurements were obtained by averaging 120 compilations on a warm compiler, and have been performed on an i7-7700K Processor running Oracle JVM 1.8.0 on Linux.

3.5 Related Work

As of today, Haskell is perhaps closest to becoming dependently-typed among the general-purpose programming languages used in industry. Haskell's type families [Kiselyov et al., 2010] provide a direct way to express type-level computations. Other language extensions such as functional dependencies [Jones, 2000] and promoted datatypes [Yorgey et al., 2012] are also moving Haskell towards dependent types. Nevertheless, programming in Haskell remains significantly different from using full-spectrum dependently-typed languages. A significant difference is that Haskell imposes a strict separation between terms and types. As a result, writing dependently-typed programs in Haskell often involves code duplication between types and terms. These redundancies can be somewhat avoided using the singletons package [Eisenberg and Weirich, 2012], which uses meta-programming to automatically generate types from datatypes and function definitions.

In the context of Haskell, Eisenberg's work on Dependent Haskell [Eisenberg, 2016] is closest to ours, in that it adds first-class support for dependent types to an established language, in a backwards-compatible way. Dependent Haskell supports general recursion without termination checks, which makes it less suitable for theorem proving. While we share similar goals, our work is differentiated by the contrasting paradigms of Scala and Haskell. Like many object-oriented languages, Scala is primarily built around subtyping and does not restrict the use of side effects. Furthermore, Eisenberg's system provides control over the relevance of values and type parameters. In contrast, our system does not support any erasure annotations and simply follows Scala's canonical erasure strategy: types are systematically erased to JVM types, and terms are left untouched. Weirich et al. established a fully mechanized type safety proof for the core of Dependent Haskell [Weirich et al., 2017].

Cayenne is a Haskell-like language with dependent types introduced in 1998 by Augustsson [Augustsson, 1998]. Like Dependent Haskell, it resembles our system in its treatment of termination, and differs by being a purely functional programming language. Cayenne's treatment of erasure is similar to Scala's: types are systematically erased. Augustsson proves that Cayenne's erasure is semantics-preserving, but does not provide any other metatheoretical results.

Adding dependent types to object-oriented languages is a remarkably under-explored area of research. A notable exception is the recent work of Kazerounian et al. [2019] on adding dependent types to Ruby. Their goals are very much aligned with ours: using type-level programming to increase program safety. Given the extremely dynamic nature of Ruby, it is unsurprising that their solution greatly differs from ours. In their work, type checking happens entirely at runtime and has to be performed at every function invocation to account for possible changes in function definitions. Safety is obtained by inserting dynamic checks, similarly to gradual typing.

4 Match Types

Type-level programming is becoming more and more popular in the realm of functional programming. However, the combination of type-level programming and subtyping remains largely unexplored in practical programming languages. This chapter presents *match types*, a type-level equivalent of pattern matching. Match types integrate seamlessly into programming languages with subtyping and, despite their simplicity, offer significant additional expressiveness. We formalize the feature of match types in a calculus based on System $F_{<:}$ and prove its soundness. We demonstrate the practicality of our system by implementing match types in the Scala 3 reference compiler, thus making type-level programming readily available to a broad audience of programmers.

Attribution

This chapter is based on [Blanvillain et al., 2022], which was written in collaboration with Jonathan Brachthäuser, Maxime Kjaer, and Martin Odersky, and published in POPL'22. Section 4.5 is based on Maxime's semester project entitled "Shape-safe TensorFlow in Dotty", where he designed a strongly typed TensorFlow interface in Scala, which checks tensor shapes at compile-time in order to prevent runtime errors in machine learning models. Maxime also contributed to the Scala 3 compiler by adding support for arithmetic computations at the type level. Jonathan mechanized the soundness proof for System FM, and provided invaluable help and guidance with redaction. Martin and I collaborated on the implementation of match types in the Scala 3 compiler, which is now actively used by the Scala community.

4.1 Introduction

There is a growing interest in using *type-level computation* to increase the expressivity of type systems, express additional constraints on the type level, and thereby improve the safety of general-purpose software. What used to be an exclusive feature of dependently typed languages is slowly becoming accessible to everyday programmers. GHC Haskell has been at the forefront of making this a reality and already provides several extensions to support type-level programming. While Haskell is certainly not the only language moving towards dependent types, the trend seems to be limited to pure functional programming languages.

We believe that type-level programming is not necessarily incompatible with other programming paradigms and that the current division exists mainly due to a lack of attention from the research community. Unfortunately, most of the existing research conducted in this domain is not directly applicable to languages with *subtyping*. Although the combination of subtyping and type-level programming has been studied extensively on the theoretical side, through the means of dependently typed systems [Aspinall, 1994; Zwanenburg, 1999; Stone and Harper, 2000; Courant, 2003; Hutchins, 2010; Yang and Oliveira, 2017], the practical side remains largely unexplored.

One notable exception is the TypeScript language, which recently introduced a new feature called *conditional type*, a type-level ternary operator based on subtyping. A conditional type, written `S extends T ? Tt : Tf`, reduces to `Tt` when `S` is a subtype of `T`, to `Tf` when `S` is not a subtype of `T`, and is left unreduced when types variables do not allow to draw a conclusion. Unfortunately, the algorithm used to reduce such conditional types is both *unsound* and *incomplete*. Despite the unsoundness (discussed in Subsection 4.6.5), the addition of conditional types to TypeScript illustrates the practical need and timeliness of this feature.

This chapter presents an alternative construct for type-level programming based on subtyping, which we call *match types*. As the name suggests, match types allow programmers to express types that perform pattern matching on types:

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

The example, which we explain in detail in Section 4.2, defines the type `Elem` by matching on the type parameter `X`. We have implemented match types in the latest version of Scala, an industry-grade, production-ready compiler. Match types have received a great interest from the Scala community, and are already in active use.

In this chapter, we explore the theoretical foundations of match types through the lens of a type system which extends System $F_{<:}$ with pattern matching at the term and type level. Our formalization serves two purposes: first, it gives a clear view on *how* we integrated match types in Scala's type system and precisely describes the changes needed on the subtyping relation to make this integration possible. Second, thanks to a type safety proof based on the standard progress and preservation theorems, it gives confidence that the design of match types is sensible and our implementation is sound.

Conditional types provide *concrete evidence* that our results are valuable beyond the context of Scala. Our results are directly applicable to TypeScript's type system and provide a clear path to fixing the unsoundness introduced by conditional types. Furthermore, we hope that match types can be useful as a reference for future designs of type-level programming features for languages with subtyping.

In summary, this chapter makes the following contributions:

- We introduce programming with match types in Scala by means of an example and highlight the interaction of type-level programming and subtyping (Section 4.2).

- We formalize match types in the self-contained calculus System FM and prove it sound, providing a theoretical basis of our implementation (Section 4.3). The chapter is accompanied by a mechanization of System FM, including proofs of progress and preservation.
- We describe our implementation of match types in the Scala compiler, discuss challenges, and relate the implementation to our formalization (Section 4.4).
- We evaluate match types in a case study, presenting a type-safe version of the NumPy library (Section 4.5).
- We motivate the design of our formalization relative to prior work, we review the extensive related work on type families in Haskell, and discuss the unsoundness of conditional types in TypeScript (Section 4.6).

4.2 Overview

In this section, we offer a brief introduction of match types in Scala by inspecting the example from the previous section in more detail:

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

This example defines a type `Elem` parametrized by one type parameter `X`. The right-hand side is defined in terms of a match on the type parameter – a *match type*. A match type reduces to one of its right-hand sides, depending on the type of its scrutinee. For example, the above type reduces as follows:

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

Here we use `S =:= T` to denote type equality between the two types `S` and `T`, witnessed by mutual subtyping. To reduce a match type, the scrutinee is compared to each pattern, one after the other, using *subtyping*. For example, although `String` is a subtype of both `String` and `Any` (the top of Scala's subtyping lattice), `Elem[String]` reduces to `Char` because the corresponding case appears first.

When the scrutinee type is a `List`, the match type `Elem` is defined recursively on the element type of the list. Hence, in our example `Elem[List[Int]]` first reduces to the type `Elem[Int]`, and eventually to the type `Int`.

4.2.1 A Lightweight Form of Dependent Typing

Match types enable a lightweight form of dependent typing, since term-level pattern matching expressions can be typed accordingly at the type level as a match types. Consider the following function definition:

```
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
}
```

This definition is well-typed because the match expression in `elem`'s body has the exact same scrutinee and pattern types as `Elem[X]` (the function's return type).

Thanks to Scala's type inference, a call to the `elem` function can have a result type that *depends* on a term-level parameter. For instance, in the expression `elem(1)`, the Scala compiler infers the singleton type `X = 1` for `elem`'s type parameter. This expression thus has type `Elem[1]`, which reduces to `Int` (via `Elem`'s third case). Similarly, in `elem(x)`, the compiler infers the singleton type `X = x.type` and the expression has type `Elem[x.type]`, which might reduce further at the callsite depending on `x`'s type.

In both examples, singleton types create a dependency between a type parameter and a term, which, by transitivity, results in a lightweight form of dependent typing, that is, a dependency between a term parameter and a function's result type.

4.2.2 Disjointness

Our design of match types induces an additional constraint on match type reduction: the scrutinee type must be known to be *disjoint* with all type patterns preceding the matching case. Informally, disjointness means that two types have no shared inhabitants.

The necessity for disjointness is best illustrated with an example. Consider `Seq[Int]`, the type of integer sequences. `Elem[Seq[Int]]` does not reduce:

1. `Elem`'s first case does not apply because `Seq[Int]` is not a subtype of `String`.

2. `Elem`'s second case does not apply because `Seq[Int]` is not a subtype of `List[Int]` (lists are sequences, but not the other way around).

3. `Elem`'s third case is *not considered* because `Seq[Int]` and `List[Int]` are not disjoint.

Therefore, the reduction algorithm gets stuck on the second case and the overall type is irreducible. Without disjointness, `Elem[Seq[Int]]` would reduce to `Seq[Int]` (via `Elem`'s third case), which would be unsound. For example, the expression `elem[Seq[Int]](List(1,2,3))` would have type `Seq[Int]`, but evaluates to the integer `1`.

Subsection 4.3.2 revisits this counterexample in a formal setting. Subsection 4.4.1 discusses our implementation of disjointness in the Scala compiler.

Syntax

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

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

Evaluation

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{(E-APP1)} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{(E-APP2)} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ T_2 \longrightarrow t_1'\ T_2}\ \text{(E-TAPP)}$$

$$(\lambda x{:}T_{11}.\,t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

$$(\lambda X{<:}T_{11}.\,t_{12})\ T_2 \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

$$\frac{t_s \longrightarrow t_s'}{\begin{array}{l} t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow \\ t_s'\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \end{array}} \quad \text{(E-MATCH1)}$$

$$\frac{(C, C_n) \in \Psi \qquad \forall m < n.\ (C, C_m) \notin \Psi}{\mathit{new}\ C\ \ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow [x_n \mapsto \mathit{new}\ C]t_n} \quad \text{(E-MATCH2)}$$

$$\frac{\forall m.\ (C, C_m) \notin \Psi}{\mathit{new}\ C\ \ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow t_d} \quad \text{(E-MATCH3)}$$

$$(\lambda x{:}T.\ t)\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow t_d \quad \text{(E-MATCH4)}$$

$$(\lambda X{<:}T.\ t)\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow t_d \quad \text{(E-MATCH5)}$$

Figure 4.1: System FM syntax and evaluation rules for a given set of classes C with class inheritance Ψ and class disjointness Ξ. Highlights correspond to additions to System $F_{<:}$.

Subtyping

$$\Gamma \vdash S <: S \quad (\text{S-Refl}) \qquad\qquad \Gamma \vdash S <: \mathrm{Top} \quad (\text{S-Top})$$

$$\Gamma \vdash \{\mathit{new}\, C\} <: C \quad (\text{S-Sin}) \qquad\qquad \frac{(C_1, C_2) \in \Psi}{\Gamma \vdash C_1 <: C_2} \quad (\text{S-Psi})$$

$$\frac{\Gamma \vdash S <: U \quad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \; (\text{S-Trans}) \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \; (\text{S-Arrow})$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad (\text{S-TVar}) \qquad \frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash (\forall X <: U_1.\, S_2) <: (\forall X <: U_1.\, T_2)} \; (\text{S-All})$$

$$\frac{\Gamma \vdash T_s <: S_n \quad \forall m < n.\, \Gamma \vdash \mathrm{disj}(T_s, S_m)}{\Gamma \vdash T_s \;\; \mathit{match}\{S_i \Rightarrow T_i\} \mathit{or}\; T_d =:= T_n} \; (\text{S-Match1/2}) \qquad \frac{\forall n.\, \Gamma \vdash \mathrm{disj}(T_s, S_n)}{\Gamma \vdash T_s \;\; \mathit{match}\{S_i \Rightarrow T_i\} \mathit{or}\; T_d =:= T_d} \; (\text{S-Match3/4})$$

$$\frac{\Gamma \vdash S_s <: T_s \quad \Gamma \vdash S_d <: T_d \quad \forall n.\, \Gamma \vdash S_n <: T_n}{\begin{array}{c}\Gamma \vdash S_s \;\; \mathit{match}\{U_i \Rightarrow S_i\} \mathit{or}\; S_d <: \\ T_s \;\; \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\; T_d\end{array}} \; (\text{S-Match5})$$

Typing

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\, t_2 : T_1 \to T_2} \; (\text{T-Abs}) \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\; t_2 : T_{12}} \; (\text{T-App})$$

$$\frac{\Gamma, X <: U_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: U_1.\, t_2 : \forall X <: U_1.\, T_2} \; (\text{T-TAbs}) \qquad \frac{\Gamma \vdash t_1 : \forall X <: T_{11}.\, T_{12} \quad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\; T_2 : [X \mapsto T_2] T_{12}} \; (\text{T-TApp})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \; (\text{T-Var}) \qquad \frac{\Gamma \vdash t : S \quad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \; (\text{T-Sub}) \qquad \Gamma \vdash \mathit{new}\, C : \{\mathit{new}\, C\} \; (\text{T-Class})$$

$$\frac{\Gamma \vdash t_s : T_s \quad \Gamma, x_i : C_i \vdash t_i : T_i \quad \Gamma \vdash t_d : T_d}{\Gamma \vdash t_s \;\; \mathit{match}\{x_i : C_i \Rightarrow t_i\} \mathit{or}\; t_d : T_s \;\; \mathit{match}\{C_i \Rightarrow T_i\} \mathit{or}\; T_d} \; (\text{T-Match})$$

Disjointness

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

$$\frac{\Gamma \vdash S <: U \quad \Gamma \vdash \mathrm{disj}(U, T)}{\Gamma \vdash \mathrm{disj}(S, T)} \; (\text{D-Sub}) \qquad \begin{array}{r}\Gamma \vdash \mathrm{disj}(T_1 \to T_2, C) \; (\text{D-Arrow}) \\ \Gamma \vdash \mathrm{disj}(\forall X <: T_1.\, T_2, C) \; (\text{D-All})\end{array}$$

Figure 4.2: System FM type system for a given set of classes C with class inheritance Ψ and class disjointness Ξ. Highlights correspond to additions to System $F_{<:}$.

4.3 Formalization

In this section, we formally present System FM, an extension of System $F_{<:}$ [Cardelli et al., 1994] with pattern matching, opaque classes, and match types. Figure 4.1 defines FM's syntax and evaluation relation. Figure 4.2 defines FM's type system, composed of three relations: typing, subtyping, and type disjointness. We discuss differences to System $F_{<:}$ in the following subsections (Subsection 4.3.1 and 4.3.2). In Subsection 4.3.3, we outline a proof of type safety for System FM. In Subsection 4.3.4, we present an extension of System FM with support for binding pattern variables in type patterns.

4.3.1 Classes

System FM is parametrized by a set of classes C with class inheritance Ψ and class disjointness Ξ. The class inheritance forms a partial order on C, that is, it is reflexive, antisymmetric and transitive. The class disjointness is symmetric relation over C.

The inheritance and disjointness parameters can be understood as a representation of a hierarchy of Scala traits and classes. For example, `trait C1; class C2 extends C1` is represented in FM as $\Psi = \{(C_1, C_2)\}$; $\Xi = \{\}$. This representation also models the fact that certain types cannot possibly have common instances. For example, `class C3; class C4` is represented as $\Psi = \{\}$; $\Xi = \{(C_3, C_4), (C_4, C_3)\}$, since Scala disallows multiple class inheritance. Inheritance and disjointness must be consistent in the sense that $(A, B) \in \Xi$ implies that there is no class C such that $(C, A) \in \Psi$ and $(C, B) \in \Psi$.

Each class in C gives rise to a constructor (written *new* C), a type (written C), and a constructor singleton type (written {*new* C}). The type C denotes all values that inherit C, while the constructor singleton type {*new* C} denotes a single value: C's constructor call. Subtyping between classes is dictated by Ψ via the S-PSI rule.

This parametric approach allows us to model class inheritance as it is found in object-oriented languages, without the need for dedicated syntax for classes and data type definitions. Although our approach might appear simplistic, it can easily model advanced object-oriented features such as multiple inheritance. We discuss the encoding of Scala's types into System FM in Subsection 4.4.1.

Our type system refers to classes by names and therefore mixes structural and nominal types. Names are useful to give a direct correspondence between runtime tags and compile-time types. As we will see, runtime tags are essential to runtime type testing and play a central role in the evaluation of pattern matching.

4.3.2 Matches

System FM supports both pattern matching on the term level (*match expressions*) as well as on the type-level (*match types*). Matches, both on terms and on types, are composed of a scrutinee, a list of cases and a default expression/type. Each case consists of a *type test* and a corresponding expression/type. At the term level, a type test consists of an inheritance check against a particular class (this is also known as a typecase [Abadi et al., 1991]). At the type level, a type test corresponds to a subtyping test with a particular type. This disparity reflects the difference between runtime, where type tests are implemented using class tables, and compile time, where types are compared using the type system in its full extent. We discuss the representation of Scala types at runtime in Subsection 4.4.6.

Throughout this chapter, we use the abbreviated syntax t_s $\mathit{match}\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\}\mathit{or}\ \mathrm{t}_d$ to denote an arbitrary number of cases, that is, $\exists n \in \mathbb{N}.\ \mathrm{t}_s\ \mathit{match}\{\mathrm{x}_1 : \mathrm{C}_1 \Rightarrow \mathrm{t}_1; \ldots; \mathrm{x}_n : \mathrm{C}_n \Rightarrow \mathrm{t}_n\}\mathit{or}\ \mathrm{t}_d$.

Match expressions and match types are related by the T-MATCH typing rule. This rule operates by typing each component of a match expression to then assemble the corresponding match type.

Example A. For example, given two disjoint classes A and B, and an empty class inheritance ($\Psi = \mathrm{Id}, \Xi = \{(\mathrm{A},\mathrm{B}),(\mathrm{B},\mathrm{A})\}$); the following function:

$\mathrm{f} = \lambda \mathrm{X} <: \mathrm{Top}.\ \lambda \mathrm{x} : \mathrm{X}.\ \mathrm{x}\ \mathit{match}\{\mathrm{a} : \mathrm{A} \Rightarrow \mathrm{foo}; \mathrm{b} : \mathrm{B} \Rightarrow \mathrm{bar}\}\mathit{or}\ \mathrm{buzz}$

can be typed precisely as

$\mathrm{f}\ :\ \forall \mathrm{X} <: \mathrm{Top}.\ \mathrm{X} \rightarrow \mathrm{X}\ \mathit{match}\{\mathrm{A} \Rightarrow \mathrm{Foo}; \mathrm{B} \Rightarrow \mathrm{Bar}\}\mathit{or}\ \mathrm{Buzz}$

where foo, bar and buzz are expressions with types Foo, Bar and Buzz, respectively.

The cases of a match expression are evaluated *sequentially*: the scrutinee is checked using the type test of each case, one after the other. The overall expression reduces to the expression that corresponds to the first successful type test (E-MATCH2). When no type test succeeds, the match evaluates to its default expression (E-MATCH3/4/5). For instance, given the function f defined in Example A, the expression (f A (*new* A)) evaluates to foo and (f B (*new* B)) to bar.

Match Type Reduction

The subtyping relation contains 5 rules for match type reduction, S-MATCH1/2/3/4/5. These rules are defined in pairs using the =:= shorthand notation, where S =:= T means that S and T are in a mutual subtyping relation. More precisely, S-MATCH1/2 in Figure 4.2 corresponds to two typing rules with identical premises and symmetrical conclusion, and the same goes for S-MATCH3/4.

The typing rules for match type reduction are best explained as generalizations of the evaluation relation. Given a match type $\mathrm{M} = \mathrm{T}_s\ \mathit{match}\{\mathrm{C}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\ \mathrm{T}_d$, M reduces to T_i if and only if, for every value t_s in T_s, the term level expression $\mathrm{t}_s\ \mathit{match}\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\}\mathit{or}\ \mathrm{t}_d$ evaluates to t_i.

The S-MATCH1/2 rules correspond to the evaluation of a match expression to its *n*th case (E-MATCH2):

$$\frac{\Gamma \vdash \mathrm{T}_s <: \mathrm{S}_n \qquad \forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)}{\Gamma \vdash \mathrm{T}_s\ \ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\ \ \mathrm{T}_d =:= \mathrm{T}_n} \qquad \text{(S-MATCH1/2)}$$

The first premise ensures that the *n*th case type test will succeed for every possible value in the scrutinee type T_s. Conversely, the second premise is a disjointness judgment, which ensures that no value in the scrutinee type would result in a successful type test for cases prior to the nth case. The S-MATCH3/4 rules correspond to an evaluation to the default case (E-MATCH2), and require disjointness between the scrutinee type and each type test type:

$$\frac{\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_n)}{\Gamma \vdash \mathrm{T}_s\ \ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\ \mathrm{T}_d =:= \mathrm{T}_d} \qquad \text{(S-MATCH3/4)}$$

Disjointness between two classes can be concluded directly using the D-XI rule which uses the class disjointness Ξ. Likewise, disjointness between a constructor singleton type and a class can be concluded directly by inspecting the class inheritance Ψ (D-PSI). Function types and universal types are disjoint from classes as they are inhabited by different values (D-ARROW, D-ALL) and thus will never match. The last disjointness rule, D-SUB, states that if U and T are disjoint, then all subtypes of U are also disjoint with T.

Example B. We continue developing Example A by showing how match type reduction rules can be used to conclude that (f B (*new* B)) has type Bar. Using T-TAPP and T-APP, the expression can be typed as follows:

f B (*new* B) : B *match*{A ⇒ Foo; B ⇒ Bar} *or* Buzz

Since our example assumes an empty class inheritance and $(\mathrm{A}, \mathrm{B}) \in \Xi$, the S-MATCH1 rule gives:

$\emptyset \vdash \mathrm{B}\ \mathit{match}\{\mathrm{A} \Rightarrow \mathrm{Foo}; \mathrm{B} \Rightarrow \mathrm{Bar}\}\mathit{or}\ \mathrm{Buzz} <: \mathrm{Bar}$

Finally, using T-SUB we get (f B (*new* B)) : Bar.

Subtyping and Disjointness

One might wonder what happens if we simplify the match type reduction rules by replacing premises of the form $\Gamma \vdash \mathrm{disj}(\mathrm{T}, \mathrm{C})$ by seemingly equivalent premises of the form $(\mathrm{T}, \mathrm{C}) \notin \Psi$. Unfortunately, the resulting system would be unsound, which can be demonstrated with a counterexample. Let us assume the function f defined in Example A, adding a new class E with $\Psi = \{(\mathrm{E}, \mathrm{A}), (\mathrm{E}, \mathrm{B})\}$ and $\Xi = \{\}$. Now, consider the term (f B (*new* E)). Since we have $\emptyset \vdash \mathrm{E} <: \mathrm{B}$, this function application is well-typed and, given that $(\mathrm{E}, \mathrm{A}) \in \Psi$, evaluates to foo. The term-level and type-level reductions are inconsistent! The unsoundness arises when using $(\mathrm{B}, \mathrm{A}) \notin \Psi$ with the modified S-MATCH1 rule to wrongly conclude that (f B (*new* E)) has type Bar. This would result in an inconsistency between types (e : Bar) and evaluation ($\mathrm{e} \longrightarrow \mathrm{foo}$), and violate type soundness. In System FM, the match type obtained when typing (f B (*new* E)) does not reduce since the scrutinee type B is neither disjoint with, nor a subtype of the first pattern type test A. In this case, unreduced match type is assigned "as is". Unreduced types can appear as the result of programming error, but can also be due to the local irreducibility of a match type. For instance, the body of f is typed with an unreduced type, as shown in Example A, but that type can later become reducible depending on type variable instantiations.

4.3.3 Type Safety

We show the type safety of System FM through the usual progress and preservation theorems. This section provides an overview of the proof structure and states the involved lemmas and

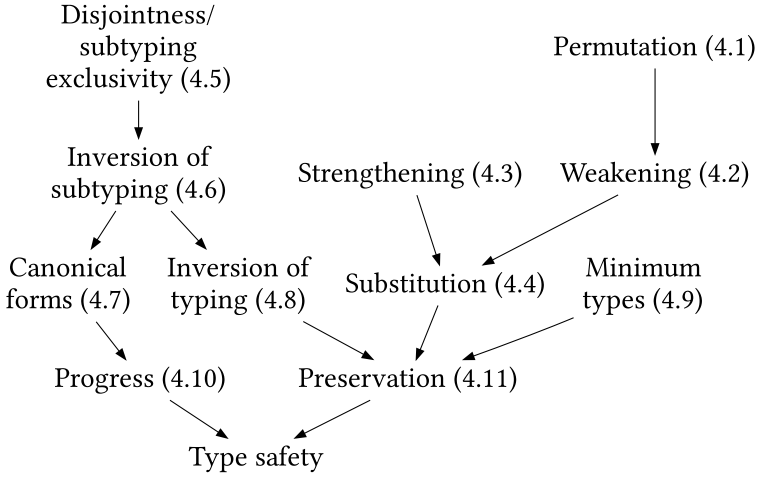

Figure 4.3: Structure of the type safety proof. Arrows represent implications between lemmas and theorems.

theorems. Detailed proofs are available in the supplementary material of this thesis, in two different versions. The first version, Blanvillain et al. [2021a], is a pen-and-paper proof where System FM is exactly as presented in Figure 4.2. The second version, Blanvillain et al. [2021b], is a mechanization of the proof in Coq, using the locally nameless representation by Aydemir et al. [2008] to model variable bindings. Our mechanization uses a simplified representation of match types with exactly one case per match. Matches with multiple cases can be expressed by nesting match types in default cases.

Figure 4.3 gives an overview of the proof structure by showing implications between the various lemmas and theorems. The basic structure resembles that of System $F_{<:}$'s standard safety proof from [Pierce, 2002]. We continue our presentation by introducing the lemmas and theorems used in our type safety proof.

Preliminary Lemmas

Our proof begins with preliminary technical lemmas: **Lemma 4.1** (Permutation), **Lemma 4.2** (Weakening), **Lemma 4.3** (Strengthening), **Lemma 4.4** (Substitution). We omit stating these lemmas as they are entirely standard yet relatively lengthy given that they span the three relations of our system: typing, subtyping, and disjointness. As usual, their proofs follow by mutual inductions on derivations.

Disjointness / Subtyping Exclusivity

The following non-standard lemma is necessary to prevent overlap between the S-MATCH1/2 and S-MATCH3/4 rules.

Lemma 4.5 (Disjointness/subtyping exclusivity).

The type disjointness and subtyping relations are mutually exclusive.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash S =:= T}\ (\text{S-M}\textsc{atch}1/2)}{\Gamma \vdash S \rightleftharpoons T} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash S =:= T}\ (\text{S-M}\textsc{atch}3/4)}{\Gamma \vdash S \rightleftharpoons T} \qquad \dfrac{\Gamma \vdash S \rightleftharpoons U \quad \Gamma \vdash U \rightleftharpoons T}{\Gamma \vdash S \rightleftharpoons T}$$

Figure 4.4: Definition of the auxiliary relation $\rightleftharpoons$, used to state inversion of subtyping.

If such overlap would be allowed, match types could reduce in several different ways, resulting in an unsound system. We prove Lemma 4.5 by contradiction. Our proof uses a mapping from System FM's types into non-empty subsets of a newly defined set $P = \{\Lambda, V\} \cup C$. Elements of P can be understood as equivalence classes for FM's types. We show that the subtyping relation in FM corresponds to a subset relation in P, and that the type disjointness relation in FM (disj) corresponds to set disjointness in P. This set-theoretical view lets us conclude the desired result directly. In our Coq mechanization, we axiomatize this lemma and delegate to the pen-and-paper proof.

Inversion of Subtyping

The following Lemma 4.6 allows us to perform inversion on the subtyping relation, which is important to show canonical forms (Lemma 4.7) and inversion of typing (Lemma 4.8). Stating the lemma requires the definition of a new relation denoted $\Gamma \vdash S \rightleftharpoons T$, defined in Figure 4.4. It represents evidence of the mutual subtyping between a match type S and a type T with the additional constraint that this evidence was exclusively constructed using pairwise applications of S-Match1/2, S-Match3/4, and S-Trans in both directions. Intuitively, $\Gamma \vdash S \rightleftharpoons T$ is handier than two independent derivations of $\Gamma \vdash S <: T$ and $\Gamma \vdash T <: S$ because it allows simultaneous induction on both subtyping directions.

Lemma 4.6 (Inversion of subtyping).

1. *If* $\Gamma \vdash S_s$ *match*$\{U_i \Rightarrow S_i\}$*or* $S_d \rightleftharpoons T$, *then either:*
   (a) $\Gamma \vdash S_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(S_s, U_m)$ *and* S_n *is a match type with* $\Gamma \vdash S_n \rightleftharpoons T$,
   (b) $\Gamma \vdash S_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(S_s, U_m)$ *and* $S_n = T$,
   (c) $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$ *and* S_d *is a match type with* $\Gamma \vdash S_d \rightleftharpoons T$,
   (d) $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$ *and* $S_d = T$.
2. *If* $\Gamma \vdash S <: X$, *or* $\Gamma \vdash S <: T$ *where* T *is a match type with* $\Gamma \vdash T \rightleftharpoons X$, *then either*
   (a) S *is a match type with* $\Gamma \vdash S \rightleftharpoons Y$, *for some* Y,
   (b) S *is a type variable.*
3. *If* $\Gamma \vdash S <: T_1 \to T_2$, *or* $\Gamma \vdash S <: T$ *where* T *is a match type with* $\Gamma \vdash T \rightleftharpoons T_1 \to T_2$, *then either*
   (a) S *is a match type with* $\Gamma \vdash S \rightleftharpoons S_1 \to S_2$, *for some* S_1, S_2 *such that* $\Gamma \vdash T_1 <: S_1$ *and* $\Gamma \vdash S_2 <: T_2$,

(b) S *is a match type with* $\Gamma \vdash S \rightleftharpoons X$, *for some* X,

(c) S *is a type variable,*

(d) S *has the form* $S_1 \to S_2$ *with* $\Gamma \vdash T_1 <: S_1$ *and* $\Gamma \vdash S_2 <: T_2$.

4. *If* $\Gamma \vdash S <: \forall X <: U_1.\, T_2$, *or* $\Gamma \vdash S <: T$ *where* T *is a match type with* $\Gamma \vdash T \rightleftharpoons \forall X <: U_1.\, T_2$, *then either*

   (a) S *is a match type with* $\Gamma \vdash S \rightleftharpoons \forall X <: U_1.\, S_2$, *for some* S_2 *such that* $\Gamma, X <: U_1 \vdash S_2 <: T_2$,

   (b) S *is a match type with* $\Gamma \vdash S \rightleftharpoons X$, *for some* X,

   (c) S *is a type variable,*

   (d) S *has the form* $\forall X <: U_1.\, S_2$ *and* $\Gamma, X <: U_1 \vdash S_2 <: T_2$.

The first point of Lemma 4.6 uses the structure of the $\rightleftharpoons$ to provide a form of inversion, which we use to prove each of the subsequent points. In comparison with the corresponding inversion lemma in $F_{<:}$'s safety proof, the statement and the proof of Lemma 4.6 are longer and more intricate. This difference is inevitable, given that match type reduction rules allow match expressions to be typed as the result of their reduction, which complexifies the inversion.

Similarly to inversion of subtyping, our canonical forms lemma is non-standard in that it uses a disjunction in its premise to account for match types.

Lemma 4.7 (Canonical forms).

1. *If* $\Gamma \vdash t : T$, *where either* T *is a type variable, or* T *is a match type with* $\Gamma \vdash T \rightleftharpoons X$, *then* t *is not a closed value.*

2. *If* v *is a closed value with* $\Gamma \vdash v : T$ *where either* $T = T_1 \to T_2$, *or* T *is a match type and* $\Gamma \vdash T \rightleftharpoons T_1 \to T_2$, *then* v *has the form* $\lambda x : S_1.\, t_2$.

3. *If* v *is a closed value with* $\Gamma \vdash v : T$ *where either* $T = \forall X <: U_1.\, T_2$, *or* T *is a match type and* $\Gamma \vdash T \rightleftharpoons \forall X <: U_1.\, T_2$, *then* v *has the form* $\lambda X <: U_1.\, t_2$.

Proof of Soundness

The remaining proof of soundness is mostly standard. Lemma 4.8 and 4.9 are simple inversions of typing rules. Lemma 4.9 is needed in the proof of preservation to recover subtyping bounds from typing judgments. The proofs proceed by routine induction on derivations.

Lemma 4.8 (Inversion of typing).

1. *If* $\Gamma \vdash \lambda x : S_1.\, s_2 : T$ *and* $\Gamma \vdash T <: U_1 \to U_2$, *then* $\Gamma \vdash U_1 <: S_1$ *and there is some* S_2 *such that* $\Gamma, x : S_1 \vdash s_2 : S_2$ *and* $\Gamma \vdash S_2 <: U_2$.

2. *If* $\Gamma \vdash \lambda X <: S_1.\, s_2 : T$ *and* $\Gamma \vdash T <: (\forall X <: U_1.\, U_2)$, *then* $U_1 = S_1$ *and there is some* S_2 *such that* $\Gamma, X <: S_1 \vdash s_2 : S_2$ *and* $\Gamma, X <: S_1 \vdash S_2 <: U_2$.

Lemma 4.9 (Minimum types).

1. *If* $\Gamma \vdash new\, \mathrm{C} : \mathrm{T}$ *then* $\Gamma \vdash \{new\, \mathrm{C}\} <: \mathrm{T}$.
2. *If* $\Gamma \vdash \lambda \mathrm{x} : \mathrm{T}_1 .\, \mathrm{t}_2 : \mathrm{T}$ *then there is some* T_2 *such that* $\Gamma \vdash \mathrm{T}_1 \to \mathrm{T}_2 <: \mathrm{T}$.
3. *If* $\Gamma \vdash \lambda \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{t}_2 : \mathrm{T}$ *then there is some* T_2 *such that* $\Gamma \vdash \forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{T}_2 <: \mathrm{T}$.

With these lemmas in hand, the proofs of progress and preservation are straightforward.

Theorem 4.10 (Progress).
If t *is a closed, well-typed term, then either* t *is a value or there is some* t' *such that* $\mathrm{t} \longrightarrow \mathrm{t}'$.

Theorem 4.11 (Preservation).
If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *and* $\mathrm{t} \longrightarrow \mathrm{t}'$ *then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

4.3.4 Type Binding Extension

In this section, we present System FMB, an extension of System FM with support for binding pattern variables in type patterns. FMB is parametrized by two sets of classes, A and B, representing non-parametric and parametric classes, respectively. A parametric class, written B T, takes exactly one type parameter[1]. We redefine C to be a syntactic object defined as C: := A|B T. The class inheritance Ψ and class disjointness Ξ remain as binary relations on C. A generic instantiation in the class hierarchy is represented as an element of Ψ, for example, `A1 extends B2[A3]` is represented as $(\mathrm{A}_1, \mathrm{B}_2\ \mathrm{A}_3) \in \Psi$. Generic inheritance is represented using multiple entries in Ψ, for example, `B1[T] extends B2[T]` is represented as $\forall \mathrm{T}.(\mathrm{B}_1\ \mathrm{T}, \mathrm{B}_2\ \mathrm{T}) \in \Psi$. This approach allows us to reuse most of FM's definitions. Indeed, our formal development treats C, Ψ, and Ξ as mathematical objects, and is compatible with FMB's new definition of classes.

In Figure 4.5, we define System FMB syntax and rules, where changes to System FM are highlighted in gray. FMB's new syntax for match expressions and match types adds a *pattern variable* to each construct. In t_s *match*$[\mathrm{X}]\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\}$*or* t_d, the pattern variable X is available to bind type parameters in C_i patterns.

The definitions of S-MATCH3/4, S-MATCH5, and T-MATCH require minor adjustments to account for the pattern variable in typing contexts. Note that pattern variables appear in contexts unconditionally, regardless of whether or not those variables are used in the corresponding patterns.

In the new subtyping rule for non-default match reduction, called BS-MATCH1/2, the first premise instantiates the pattern variable X to some type U such that the scrutinee type is a subtype of the nth pattern:

$$\frac{\Gamma, \mathrm{X} <: \mathrm{U} \vdash \mathrm{T}_s <: \mathrm{S}_n \qquad \forall m < n.\ \Gamma, \mathrm{X} <: \mathrm{Top} \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)}{\Gamma \vdash \mathrm{T}_s\ \ match\ [\mathrm{X}]\ \{\mathrm{S}_i \Rightarrow \mathrm{T}_i\} or\ \ \mathrm{T}_d =:= [\mathrm{X} \mapsto \mathrm{U}]\ \mathrm{T}_n} \qquad \text{(BS-MATCH1/2)}$$

[1] The restriction to a single type parameter is for presentation purposes. Both System FMB's type system and type-safety proof can easily be adapted to support a variable number of binding variables.

Syntax

$\colorbox{lightgray}{C ::=}$			$t ::= \ldots$	
	$\colorbox{lightgray}{A}$	*ground class*	$t\ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{\overline{x : C \Rightarrow t}\}\mathit{or}\ t$	*match expr.*
	$\colorbox{lightgray}{B T}$	*parametric class*	$T ::= \ldots$	
			$T\ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{\overline{T \Rightarrow T}\}\mathit{or}\ T$	*match type*

Evaluation

$$\frac{(C, \colorbox{lightgray}{$[X \mapsto U]$}\, C_n) \in \Psi \qquad \forall m < n.\, \colorbox{lightgray}{$\forall T.$}\ (C, \colorbox{lightgray}{$[X \mapsto T]$}\, C_m) \notin \Psi}{\mathit{new}\, C\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow \colorbox{lightgray}{$[X \mapsto U]$}\ [x_n \mapsto \mathit{new}\, C] t_n}\ \text{(BE-M\textsc{atch}2)}$$

$$\frac{\forall m.\, \colorbox{lightgray}{$\forall T.$}\ (C, \colorbox{lightgray}{$[X \mapsto T]$}\, C_m) \notin \Psi}{\mathit{new}\, C\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \longrightarrow t_d}\ \text{(BE-M\textsc{atch}3)}$$

Subtyping

$$\frac{\Gamma, \colorbox{lightgray}{$X <: U$} \vdash T_s <: S_n \qquad \forall m < n.\ \Gamma, \colorbox{lightgray}{$X <: Top$} \vdash \mathrm{disj}(T_s, S_m)}{\Gamma \vdash T_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{S_i \Rightarrow T_i\}\mathit{or}\ T_d =:= \colorbox{lightgray}{$[X \mapsto U]$}\, T_n}\ \text{(BS-M\textsc{atch}1/2)}$$

$$\frac{\forall n.\ \Gamma, \colorbox{lightgray}{$X <: Top$} \vdash \mathrm{disj}(T_s, S_n)}{\Gamma \vdash T_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{S_i \Rightarrow T_i\}\mathit{or}\ T_d =:= T_d}\ \text{(BS-M\textsc{atch}3/4)}$$

$$\frac{\Gamma \vdash S_s <: T_s \qquad \Gamma \vdash S_d <: T_d \qquad \forall n.\ \Gamma, \colorbox{lightgray}{$X <: Top$} \vdash S_n <: T_n}{\begin{array}{c}\Gamma \vdash S_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{U_i \Rightarrow S_i\}\mathit{or}\ S_d <: \\ T_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{U_i \Rightarrow T_i\}\mathit{or}\ T_d\end{array}}\ \text{(BS-M\textsc{atch}5)}$$

Typing

$$\frac{\Gamma \vdash t_s : T_s \qquad \Gamma, \colorbox{lightgray}{$X <: Top$}, x_i : C_i \vdash t_i : T_i \qquad \Gamma \vdash t_d : T_d}{\Gamma \vdash t_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d : T_s\ \ \mathit{match}\ \colorbox{lightgray}{[X]}\ \{C_i \Rightarrow T_i\}\mathit{or}\ T_d}\ \text{(BT-M\textsc{atch})}$$

Figure 4.5: System FMB syntax, evaluation and typing rules for a given set of ground classes A, set of parametric classes B, class inheritance Ψ, and class disjointness Ξ. Highlights correspond to changes made to System FM.

Here U is completely unconstrained: any instantiation of X such that $T_s <: S_n$ would be admissible. The disjointness judgments use a weaker upper bound for X than the subtyping judgment ($X <: \mathrm{Top}$ instead of $X <: U$). This is because the scrutinee type must be shown disjoint with non-matching pattern types for every possible instantiation of X. In an algorithmic system, U would be computed during type inference by constraint solving.

The new evaluation rule for non-default match reduction uses a similar mechanism: it looks for the *first case* where the pattern variable can be instantiated such that the scrutinee inherits the corresponding pattern:

$$\frac{(\mathrm{C}, [\mathrm{X}\mapsto \mathrm{U}]\, \mathrm{C}_n) \in \Psi \qquad \forall m < n.\ \forall \mathrm{T}.\ (\mathrm{C}, [\mathrm{X}\mapsto \mathrm{T}]\, \mathrm{C}_m) \notin \Psi}{\textit{new}\, \mathrm{C}\ \ \textit{match}\ [\mathrm{X}]\ \{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\}\ \textit{or}\ \mathrm{t}_d \longrightarrow [\mathrm{X}\mapsto \mathrm{U}]\ [\mathrm{x}_n \mapsto \textit{new}\, \mathrm{C}]\mathrm{t}_n} \quad (\text{BE-M}\textsc{atch}2)$$

The second premise rules out non-matching cases with a universal quantifier ranging over all types. A concrete implementation would certainly opt for a more efficient approach, for instance by implementing Ψ as a lookup table.

Example C. Consider the following class hierarchy with two ground classes: Char and String, and a single parametric class List, such that String extends List Char:

$$\begin{aligned} &\mathrm{A} = \{\mathrm{Char}, \mathrm{String}\} && \mathrm{B} = \{\mathrm{List}\} \\ &\Psi = \{(\mathrm{String}, \mathrm{List\ Char})\} \cup \mathrm{Id} && \Xi = \{\} \\ &\mathrm{f} = \lambda \mathrm{x} : \mathrm{Top}.\ \mathrm{x}\ \textit{match}[\mathrm{X}]\{\mathrm{xs} : \mathrm{List\ X} \Rightarrow \mathrm{foo}\}\,\textit{or}\ \mathrm{bar} \end{aligned}$$

The function f matches its arguments against the List X pattern, where X is a pattern variable. We examine the evaluation of two applications of f:

1. f(*new* String) matches against List X with X = Char and evaluates to $[\mathrm{X}\mapsto\mathrm{Char}][\mathrm{x}\mapsto \textit{new}\ \mathrm{String}]\mathrm{foo}$ via BE-MATCH2 (since $(\mathrm{String}, \mathrm{List\ Char}) \in \Psi$).

2. f(*new* List Top) also matches List X, this time with X = Top, and evaluates to $[\mathrm{X}\mapsto \mathrm{Top}][\mathrm{x}\mapsto \textit{new}\ \mathrm{List\ Top}]\mathrm{foo}$ via BE-MATCH2. Here $(\mathrm{List\ Top}, \mathrm{List\ Top}) \in \Psi$ follows from Ψ's reflexivity.

We established the type safety of System FMB by adapting System FM's pen-and-paper proof. The required changes are lengthy, but relatively uninteresting; it boils down to additional bookkeeping of pattern variables in contexts. The main takeaway from FMB's type safety is that the proof does not require additional constraints on type U in BS-MATCH1/2. As a result, algorithmic implementations are free to use any mechanism to come up with instantiations of pattern variables.

4.4 Implementation

Match types are implemented in Dotty, the reference compiler for Scala 3. This section explains how our implementation relates to the formalization presented in Section 4.3.

In the compiler, match-type reduction happens during subtyping, just like in System FM. In order for subtyping to remain algorithmic, match type reduction rules are never used to introduce new match types, but only to simplify the ones present in the original program. The reduction algorithm closely follows the typing rules of Section 4.3. The scrutinee type is compared with each pattern sequentially. If the scrutinee is a subtype of the first pattern type, the match type reduces. Otherwise, if the scrutinee can be shown to be disjoint with the first pattern type, the algorithm proceeds to the next pattern. If the algorithm reaches a pattern where neither subtyping nor disjointness can be concluded, the reduction is aborted and the match type remains unreduced.

4.4.1 Disjointness in Scala

Separate compilation is the biggest obstacle to concluding that two types are disjoint. Indeed, in Scala, all traits and classes are extensible by default. Because Scala programs are compiled with an open-world assumption, it is common for types to be effectively disjoint in the current compilation unit, but due to potential extensions in future compilations, the compiler must stay conservative.

Separate compilation is the reason why our formalization requires two different parameters to describe its class hierarchy. One particular instantiation of System FM can be thought of as a model of Scala's type system for a particular compilation unit, where C represents the set of classes declared *so far*. The class inheritance, Ψ, remains valid for all subsequent compilation units: new class definitions do not alter the inheritance between previously defined classes. However, the inheritance parameter (Ψ) is, on its own, not sufficient to conclude that two classes are disjoint: new class definitions can introduce new overlaps between existing classes. For this reason, our formalization uses a separate parameter to describe class disjointness (Ξ). To account for separate compilation, Ξ should only contain pairs of classes which would remain disjoint despite potential additions to the current set of classes.

Thankfully, Scala provides several ways to restrict extensibility. The *sealed* and *final* annotations on traits and classes directly restrict the extensibility of annotated types: sealed types can only be extended in the same file as its declaration, thereby providing a way to enumerate all the children of a type. Thus, disjointness of sealed traits and classes can be computed recursively by iterating over all the possible subtypes of that type. The main distinction between traits and classes is that a class can extend at most one superclass. This property allows the compiler to assert that classes are disjoint with a simple check: given two classes A and B, if neither $A <: B$ nor $B <: A$, then no class could possibly extend both A and B, and those two types are disjoint.

As an example, consider the following Scala definitions (left-hand side), and the corresponding instantiation of System FM (right-hand side):

```
sealed trait Part
final class Wheel extends Part
final class DiscBrake extends Part

trait Vehicle
class Bicycle extends Vehicle
class RoadBike extends Bicycle

class Helmet
```

$$
\begin{aligned}
C &= \{P, W, D, V, B, R, H\} \\
\Psi &= \{(W,P),(D,P),(B,V),(R,B),(R,V)\} \\
\Xi &= \{(P,V),(P,B),(P,R),(P,H), \\
&\quad (W,V),(W,B),(W,R),(W,H), \\
&\quad (D,V),(D,B),(D,R),(D,H), \\
&\quad (W,D),(B,H),(R,H)\}
\end{aligned}
$$

The classes and inheritance relation are practically isomorphic between the two representation: Scala classes have a one-to-one correspondence to their FM counterparts (abbreviated with initials) and the inheritance only contains an additional entry for R and V, obtained by transitivity (Ψ's reflexivity and Ξ's symmetry are omitted for brevity).

P is declared sealed, meaning that no additional parts can be defined outside of this compilation unit. As a result, we can enumerate all parts to conclude that none are vehicles and $(P,V) \in \Xi$. Note that this would not be the case if either W or D was declared non-final, since extending those classes would indirectly create new parts. B and H are both classes that do not inherit each other, which implies that $(B,H) \in \Xi$ given that Scala classes can extend at most one class. V and H, however, cannot be concluded disjoint in those definitions. If that turns out to be a desirable property, disjointness could easily be obtained by sealing V or finalizing H.

4.4.2 Empty Types

An important limitation of System FM compared to Scala's type system is that it does not support empty types. The bottom of Scala's subtyping lattice, called `Nothing`, provides a direct way to refer to empty sets of values. Intersection types also provide a way to construct uninhabited types given that Scala does not forbid intersecting two disjoint types. Empty types are problematic for the match type reduction algorithm as they break the fundamental assumption that two types cannot be both disjoint and subtypes (Lemma 4.5). To account for this, our implementation uses an additional *inhabitance check* on scrutinee types before attempting any reduction.

To show why empty types are problematic, we can construct an example where, in the absence of an inhabitance check, the same match type could be reduced differently in two different contexts:

```
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
}
```

In this example, the definition of `f` in `C` type-checks because `X & String` and `Int` are disjoint (since `String` and `Int` are disjoint) and `M[X & String]` reduces to `Int` (`M`'s second case applies). Class `D` refines the definition of `C` by giving concrete definition of `X`. The unsoundness manifests itself in the body of class `D`, where `X & String` is a subtype of `Int` and `M[X & String]` reduces to `String` (`M`'s first case applies). There, it is possible to call the function `f` with a string argument, which would result in a runtime error. Checking for scrutinee inhabitance prevents this class of errors. In this example, it would prevent `M[Int & String]` from reducing given that `Int & String` is not inhabited.

4.4.3 Null Values

In Scala 3, null values no longer inhabit every type: nullable types require explicit annotations of the form `A | Null` [Nieto et al., 2020]. Our implementation of subtyping and disjointness handles union types and therefore needs no particular treatment of null values. To allow easy migration from older versions, the strict treatment of nulls is still optional in Scala 3.0, enabled by the command line option `-Yexplicit-nulls`. The plan is to make strict null checking the default in the future.

4.4.4 Variance

Scala supports variance annotations on type parameters of higher-kinded types. These annotations allow programmers to specify how the subtyping of annotated parameters influences the subtyping of the higher-kinded type. For instance, `type F[+T]` defines a type `F` that is covariant in its first type parameter, meaning that `T1 <: T2` implies `F[T1] <: F[T2]`. Contravariance, written `type G[-T]`, has the opposite meaning: `T1 <: T2` implies `G[T2] <: G[T1]`.

It would appear that co- and contravariant types are always overlapping, given that, for all types `X`, `F[Nothing] <: F[X]` and `G[Any] <: G[X]` (where `Nothing` and `Any` are Scala's bottom and top types). However, in the case of covariant parameters, an exception can be made when the said type parameter corresponds to a field or a constructor parameter: the `Nothing` instantiation can be ruled out because no runtime program can produce a value of that type.

Scala tuples, for example, fall into this category. `Tuple2`, the class for pairs, is defined as follows:

```
case class Tuple2[+T1, +T2](_1: T1, _2: T2)
```

Given two instantiations of this this type, `Tuple2[Int,X]` and `Tuple2[String,X]`, although `Tuple2[Nothing,X]` is a subtype of both, there is no runtime value of type `Tuple2[Nothing,X]` (since `Nothing` is uninhabited), and, as a result, those two types are disjoint. Our disjointness algorithm implements this kind of reasoning to conclude disjointness in the presence of covariant type parameters.

4.4.5 Pattern Matching Exhaustivity

The Scala compiler checks for pattern matching exhaustivity to prevent runtime exceptions caused by missing cases. Exhaustivity checking uses static knowledge about the class hierarchy (such as the sealed and final annotations) to check that every value in the scrutinee type is covered by the pattern clauses [Liu, 2016]. Non-exhaustive patterns are compiled with an additional "catch-all" case which throws a runtime exception. System FM uses default cases as a replacement for systematic exhaustivity checks or runtime exceptions.

4.4.6 Types at Runtime

Scala's primary platform is the Java virtual machine (JVM). On the JVM, Scala is compiled using *partial erasures*, where parts of types are preserved and translated to the JVM's type systems, and other parts are erased [Schinz, 2005]. For instance, ground classes are compiled directly to JVM classes, but type parameters and type variables are erased and replaced by their bounds.

Erasure directly affects the type tests that can be performed at runtime. For example, while `case xs: List[Int] =>` is a syntactically valid pattern, it will lead to a compiler warning since the `Int` type parameter is eliminated by erasure and cannot be checked at runtime.

This restriction is reflected in our formalism by the difference between the evaluation rules for match expressions and the reduction rules for match types: evaluation is limited to inheritance checks on statically defined classes ($(\mathrm{C}, \mathrm{C}_n) \in \Psi$ in E-MATCH2/3), as opposed to the match type reduction rules which are defined using the subtyping and type disjointness relations ($\Gamma \vdash \mathrm{T}_s <: \mathrm{S}_n$ and $\Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)$ in S-MATCH1/2/3/4).

In System FMB (Subsection 4.3.4), match expressions support two sorts of parametric patterns: they can be either instantiated ($match\{\mathrm{xs} : \mathrm{List\ Int}\}$, where $\mathrm{Int} \in \mathrm{A}$), or use a binding pattern variable ($match[\mathrm{X}]\{\mathrm{xs} : \mathrm{List\ X}\}$, where X is a pattern variable). In this sense, System FMB is *more expressive* than Scala, where instantiated patterns are not available at the term level due to type erasure.

4.4.7 Non-Termination

Unlike our calculus, the Scala implementation also allows match types to be defined recursively. Recursive match types can cause subtyping checks to loop indefinitely. Our implementation does not check match types for termination, as any such check would necessarily limit expressiveness or convenience. Instead, we detect divergence during match type reduction using a fuel mechanism. The compiler is given an initial amount of fuel, which is consumed one unit at a time on every reduction step. If the compiler runs out of fuel, the reduction is aborted with a "recursion limit exceeded" error. The current implementation uses a fixed amount of initial fuel. Although this seems to be sufficient for most practical purposes, we plan on making it configurable. This mechanism is completely standard and already used in other programing languages with unbounded recursion at the type level [Eisenberg et al., 2014; Sjoberg, 2015; Eisenberg, 2016].

4.4.8 Inference

System FM's type rules enable any match expression to be typed as a match type, but the situation is different in the full Scala language. Pattern matching in Scala supports many sorts of patterns [Emir et al., 2007], most of which do not have a match type counterpart. Furthermore, typing match expressions as match types is not enabled by default in order to preserve backward compatibility. Instead, explicit type annotations must be provided.

4.4.9 Caching

Scala's type-checking algorithm makes heavy use of caching to improve its performance. Special care must be taken when caching the result of match type reduction, given that the subtyping and disjointness checks are context-dependent. Our implementation uses a context-aware cache for match types that automatically invalidates reduction results when match types are reduced in new contexts. An example where naive caching would be incorrect can be found in Subsection 4.6.4.

4.4.10 Size of the Implementation

In terms of lines of code, our match type implementation is a relatively modest addition to the Scala compiler: the overall changes amount to around 1500 lines (excluding tests and documentation).

4.5 Case Study: Shape-Safe NumPy

In this section, we present a case study to show how match types can be used to express complex type constraints, which in turn can prevent certain programming errors at compile time. To this end, we outline the type-level implementation of a library for multidimensional arrays which mimics the NumPy API [Harris et al., 2020]. The goal of our library is to provide a *shape*-safe interface for manipulating n-dimensional arrays (abbreviated ndarrays), where array shapes and indices are checked for errors at compile-time rather than at runtime. Shape and indexing errors in ndarrays is a widely acknowledged problem [Barham and Isard, 2019; Rush, 2019], and several solutions have already been proposed, notably in the form of libraries that rely on type-level programming [Chen, 2017; Huang et al., 2017–2022]. Our library uses match types to provide a shape-safe NumPy-like interface.

Scala programmers have a long history of using ad-hoc solutions for type-level programming [Sabin and Shapeless contributors, 2011–2022; Pilquist and Scodec contributors, 2013–2022; Blanvillain et al., 2016–2022]. These solutions have several downsides, such as being slow to compile and cumbersome to use. Match types aim at simplifying type-level programming by providing first-class language support. We believe that the approach presented in this case study is an improvement over the status quo because it does not use metaprogramming or any sort of convoluted encoding to express type-level operations.

4.5.1 Shape Errors in Python

In the example Python code below, the `img_batch` ndarray is a batch of 25 randomly generated RGB images of size 256×256. The code aims to compute a vector of length 25 containing the average grayscale color of each image in the batch, and then create a square 5×5 image of the average grayscale colors. However, this code contains a shape error and will throw an error at runtime.

```
import numpy as np
img_batch = np.random.normal(size=(25, 256, 256, 3))
avg_colors = np.mean(img_batch, (0, 1, 2))
avg_color_square = np.reshape(avg_colors, (5, 5))
```

The error is in the call to `np.mean`, which takes as argument a list of axes to reduce along. Unfortunately, the arguments to `np.mean` are off-by-one and result in a vector of length 3 (`img_batch`'s last dimension) instead of the intended length of 25 (`img_batch`'s first dimension); `avg_color` thus contains the average RGB color of the batch instead of the average grayscale color for each image. The reshape operation will then fail at runtime, as it cannot reshape a 3-element vector into a 25-element matrix. This error can be difficult to spot, given that NumPy's interface for reducing along multiple axes is index-based. Runtime errors like this one can be particularly frustrating when they occur late in a long-running computation.

In the remainder of this section, we show how match types can be used to prevent this class of error. After a preliminary introduction of singleton types (Subsection 4.5.2), we introduce ndarray shapes at the type level using HLists (Subsection 4.5.3). In Subsection 4.5.4, we show type-level implementations of the `np.mean` and `np.reshape` operations using match types. Finally, we show how this newly defined API can detect and report the error from our original Python example.

4.5.2 Singleton Types

Scala supports singleton types, which are types inhabited by a single value [Leontiev et al., 2014]. For instance, the singleton type `1` denotes the type containing the integer value `1`. The Scala standard library contains several predefined match types to perform arithmetic operations at the type level. For instance, `type +[A <: Int, B <: Int]` represents the addition of integer singleton types. Internally, the compiler is special-cased to implement these arithmetic operations using constant folding. Representing numbers with singleton types is desirable for practical purposes, but not absolutely necessary for this case study (for instance, Peano numerals could be used instead).

4.5.3 Type-Level Array Shape

The shape of an ndarray is a list of dimension lengths; we say that the shape $(a_1, a_2, \ldots, a_n)$ has n dimensions. For instance, a three-by-four matrix is a two-dimensional ndarray of shape $(3, 4)$. We represent the shape of an ndarray at the type level using a heterogeneous type list, or HList for short [Kiselyov et al., 2004]. We define an HList called `Shape` as an ADT with two constructors[2], `#:` and `∅`.

```
enum Shape {
  case #:[H <: Int, T <: Shape](head: H, tail: T)
  case ∅
}
```

This data type definition allows us to write lists of dimension sizes, both at the term level as `#:(3, #:(4, ∅))`, and at the type level as `#:[3, #:[4, ∅]]`. The HList type can equivalently be written with `#:` in infix notation, as `3 #: 4 #: ∅`.

To represent ndarrays, we define the `NDArray` type. This type is indexed by the ndarray element type (`T`), and by the ndarray shape (`S`), represented as an HList:

```
trait NDArray[T, S <: Shape]
```

The goal of our presentation is to define type-safe operations on `NDArrays`. Since our focus is on the type-level, we do not include value-level counterparts to `T` and `S` in the definition of `NDArray`, but this would be necessary in a complete implementation.

To construct ndarrays, we define `random_normal`, which creates an ndarray of a given shape, where all elements are random `Floats`:[3]

```
def random_normal[S <: Shape](shape: S): NDArray[Float, S] = ???
```

4.5.4 Computation on Shapes with Match Types

Encoding the types and shapes of ndarrays in the types allows us to readily provide type- and shape-safety for simple ndarray operations. For instance, the element-wise Hadamard product, written as `np.multiply(x, y)`, requires the `x` and `y` ndarrays to have the same shape and element types. This constraint does not require any match types, but can simply be expressed as:

```
def multiply[T, S <: Shape](x: NDArray[T, S], y: NDArray[T, S]): NDArray[T, S]
    = ???
```

However, we will need the additional expressiveness of match types to implement more complex constraints on array shapes, such as for reshapes (4.5.4) and reductions (4.5.4).

Reshape

An operation commonly used in NumPy is `np.reshape`, which changes the shape of an ndarray, but does not change its values. A restriction imposed by the NumPy API is that the output shape must have the same number of elements as the input shape. The number of elements of a shape is the product of the sizes of its dimensions; an ndarray of shape `2 #: 3 #: 4 #: ∅` has 2 × 3 × 4 = 24 elements. Note that an ndarray of shape `∅` is a scalar, and thus has a single element. This can be naturally expressed with a match type:

[2] In the interest of clarity, our presentation omits type bounds that are necessary to guide type inference, for example in the definition of the `Shape` data type.

[3] This snippet uses the triple question mark operator, Scala's standard notation for missing or omitted implementations.

```
type NumElements[X <: Shape] <: Int =
  X match {
    case ∅ => 1
    case head #: tail => head * NumElements[tail]
  }
```

To restrict reshaping to be applicable only on valid shapes, the type system must support encoding type equality constraints. For this, we make use of Scala's implicit parameters, and of the =:= type equality constraint that is a part of Scala's standard library.

```
def reshape[T, From <: Shape, To <: Shape](arr: NDArray[T, From], newshape: To)
    (implicit ev: NumElements[From] =:= NumElements[To]): NDArray[T, To] = ???
```

With this definition of `reshape`, the compiler will only accept a usage of `reshape` if it is able to prove that the number of elements in the old shape is the same as in the new shape. This example illustrates how match types can be used in concert with existing features like singleton types and implicit resolution to express powerful constraints.

Reduction

The NumPy API provides a variety of functions to reduce along a set of axes of an ndarray, such as `np.mean(ndarray, axes)` or `np.var(ndarray, axes)`. The `axes` parameter is a list of indices of dimensions, listing exactly those dimensions that will no longer be present in the output ndarray. The dimension indices can be unordered and repeated, and out-of-bounds indices result in an error. In the Python API, passing the `None` value instead of a list of axes reduces along all axes, meaning that the operation returns a scalar. Note that this is the opposite of passing ∅, which means that we reduce over no axes (effectively a no-op).

This behavior is more complex than the previous examples, but can still be described accurately by a match type. We use a match type called `ReduceAxes` to compute the return shape of the operation.

```
def mean[T, S <: Shape, A <: Shape](arr: NDArray[T, S], axes: A): NDArray[T,
    ReduceAxes[S, A]] = ???
```

We implement reductions along a given list of indices with logic similar to two nested loops. The outer loop, `Loop`, traverses the shape and counts the current index. The inner loops are implemented using `Contains` and `Remove`, standard operations on HLists (omitted). When `Loop` reaches the end of the list, if there are still axes to remove, these are out of bounds for the initial shape: the match type intentionally gets stuck in such cases.

```
type ReduceAxes[S <: Shape, Axes <: None | Shape] <: Shape =
  Axes match {
    case None => ∅
    case Shape => Loop[S, Axes, 0]
  }

type Loop[S <: Shape, Axes <: Shape, I <: Int] <: Shape =
  S match {
    case head #: tail => Contains[Axes, I] match {
      case true => Loop[tail, Remove[Axes, I], I + 1]
      case false => head #: Loop[tail, Axes, I + 1]
    }
    case ∅ => Axes match {
      case ∅ => ∅
      // otherwise, do not reduce further
    }
  }
```

4.5.5 Shape safety

Having defined `random_normal`, `reshape` and `mean`, we can rewrite our original Python example in Scala:

```
val img_batch = random_normal(#:(25, #:(256, #:(256, #:(3, ∅)))))
val avg_colors = mean(img_batch, #:(0, #:(1, #:(2, ∅))))
val avg_color_square = reshape(avg_colors, #:(5, #:(5, ∅)))
```

As expected, the call to `mean` returns a tensor of shape `3 #: ∅`. Therefore, the call to `reshape` does not type-check since the input shape has 3 elements instead of 25. If we fix the off by one error to reduce along the correct indices, `1 #: 2 #: 3 #: ∅`, the call to `reshape` type-checks and `avg_color` has shape `25 #: ∅`, as expected.

4.6 Related work

In this section, we provide a review of existing work and relate match types to dependently typed calculi with subtyping, intensional type analysis, type families and roles in Haskell, and conditional types in TypeScript.

4.6.1 Dependently Typed Calculi with Subtyping

There is a vast amount of literature on type systems combining subtyping with dependent types, justifying a full survey to relate it appropriately. Instead, we offer a condensed summary of our reading journey and explain what led us to decide on using System $F_{<:}$ as a foundation for our formalization.

Dependently typed calculi typically use the same language to describe terms and types. This unification is also commonly used in the presence of subtyping [Zwanenburg, 1999; Hutchins, 2010; Yang and Oliveira, 2017]. For systems with a complete term/type symmetry, this is a natural design, as it is concise and simplifies the meta-theory. Unfortunately, the lack of distinction between term and type level renders these systems impractical for our purpose, given that our research takes place in the context of an existing language with a clear term/type separation.

Singleton types provide an interesting middle ground between unified and separate syntax and have also been studied in conjunction with dependent types and subtyping [Aspinall, 1994; Stone and Harper, 2000; Courant, 2003]. Singleton types give a mechanism to refer to terms *in* types, usually by means of a set-like syntax. This mechanism is appealing because it allows type system designers to cherry-pick the term constructs that should be allowed in types. When multiple constructs are shared between terms and types, singleton types provide a clear economy of concepts. In our study of match types, a minimal use of singleton types would result in sharing a single constructor between the term and the type languages: the constructor for matches. It is unclear if the benefits in doing so would outweigh the additional complexity.

Dependent Object Types (DOT) are, to this day, the most significant effort in formalizing Scala's type system [Amin and Rompf, 2017]. DOT does not directly support any form of type-level computation. We considered using DOT as a starting point for our work, however, despite the recent effort to simplify DOT's soundness proof [Rapoport et al., 2017; Giarrusso et al., 2020], extending DOT remains too big of a challenge to concisely describe language extensions.

After several attempts at formalizing match types within existing systems, we decided to pursue a simpler route of adding new constructs to a system without dependent types. After all, the primary purpose of System FM is to serve as a medium to concisely *explain* our type-checking algorithm for match types. For this reason, we built our work on top of System $F_{<:}$, which we believe should be the simplest, most familiar calculus among the systems cited in this section.

4.6.2 Intensional Type Analysis

In their work on intensional type analysis [Harper and Morrisett, 1995], Harper and Morrisett introduce the λ_i^{ML} calculus that supports structural analysis of types. In λ_i^{ML}, types are represented as expressions that can be inspected by case analysis using a "typecase" construct, available both at the term and at the type level. Match types can be seen as an extension of intensional type analysis to work with object-oriented class hierarchies and subtyping. Whereas patterns in λ_i^{ML} are limited to a fixed set of disjoint types, match types need to deal with open class hierarchies, of which not all members are known at compile-time. This means pattern types can overlap, and we need to perform an analysis of disjointness between the scrutinee type and each pattern type. Disjointness allows for sound reduction in the presence of overlapping patterns and abstract scrutinee types, while retaining the natural sequential evaluation order of pattern matching.

4.6.3 Type Families in Haskell

Haskell's type families [Chakravarty et al., 2005; Schrijvers et al., 2008] allow programmers to define type-level functions using pattern matching. By default, type families are open, which means that a definition can spread across multiple files and compilations. This flexibility induces a substantial restriction on type family definitions: patterns must not overlap. One benefit of this restriction is that it prevents any ambiguity in the reduction of type families (patterns are pairwise disjoint), which is required given the distributed nature of definitions. Open type families are well-suited to be used in conjunction with type classes, since both constructs have open-ended definitions with non-overlapping constraints.

Closed type families (CTFs), as introduced by Eisenberg et al. [2014], allow for overlapping cases in type family definitions. Unlike the open variant, CTF reduction is performed sequentially, based on unification and apartness checks. In this regard, CTFs are closely related to match types. In fact, if we replace unification checks with subtyping and apartness checks with disjointness, their reduction algorithm is practically identical to ours, from a high-level perspective.

By default, Haskell checks for termination of recursive type families, but this check can be disabled to increase type families' expressiveness. Although the formalization presented in [Eisenberg et al., 2014] does not cover non-terminating families, the paper discusses a soundness problem caused by non-termination. The problem only occurs in the presence of repeated type bindings in patterns. In Scala, match types (and pattern matching in general) do not allow repeated bindings and are therefore not affected by this problem.

4.6.4 Roles in Haskell

Haskell's roles were introduced by Weirich et al. [2011] to fix a long-standing unsoundness caused by the interaction of open type families and the `newtype` construct. We understand roles as type annotations which specify whether a given type can safely be nominally compared to other types, or if *representational equality* (RE) should be used instead. The word representation in RE refers to the runtime representation of a type. In particular, RE dealiases `newtype` constructs.

In their introductory example, Weirich et al. show how type family reduction can lead to unsoundness in the absence of role annotations. Their presentation also includes a hypothetical translation of their example to Standard ML, which translates directly to Scala as follows:

```
trait AgeClass {
  type Age
  def addAge(a: Age, i: Int): Int
}
object AgeObject extends AgeClass {
  type Age = Int
  def addAge(a: Age, i: Int): Int = a + i
}
```

In this example, type `Age` is abstract in `AgeClass` and concrete in `AgeObject`. In the pre-role Haskell equivalent of this example, the unsoundness comes when the above definitions are combined with a type family that discriminates `Age` and `Int`. Such type family would reduce differently in `AgeClass`, where the types are different, and in `AgeObject`, where those two types are synonyms, which can easily be exploited to obtain a runtime error.

Luckily, our design of match types is not affected by this issue. The reason comes from the use of subtyping (and disjointness), which shields our implementation from incorrectly discriminating `Age` from `Int` when `Age` is abstract. Consider the following match type definition (directly translated from the Haskell example):

```
type M[X] = X match {
  case Age => Char
  case Int => Bool
}
```

Our algorithm would not reduce `M[Int]` to `Bool` in `AgeClass` as this reduction would require evidence that `Age` and `Int` are disjoint, which cannot be constructed when `Age` is an unbounded abstract type.

4.6.5 Conditional Types in TypeScript

TypeScript's conditional types, briefly mentioned in the introduction, are a type-level ternary operator based on subtyping. Conditional types can be nested into a sequence of patterns that evaluate in order, making them similar to match types.

The TypeScript language specification briefly describes the algorithm used to reduce conditional types in the presence of type variables [The TypeScript development team, 2019–2022]. Given a type `S extends T ? Tt : Tf`, the TypeScript compiler first replaces all the type parameters in `S` and `T` by `any` (the top of TypeScript's subtyping lattice). If the resulting types (after substitution) are not subtypes, the overall condition is reduced to `Tf`. Unfortunately, this algorithm is both *unsound* and *incomplete*.

The unsoundness is caused by the incorrect widening of type parameters in contravariant position. Although TypeScript does not have syntax for variance annotations, function types are covariant in their return type and contravariant in their arguments. The conditional type unification algorithm wrongly approximates `X => string` to `any => string` and unifies the former with latter, which can lead to a runtime errors.

The incompleteness comes from the fact that type parameter approximation does not account for type parameter bounds. Consider the following example:

```
type M<X> = X extends string ? A : B
function f<X extends string>: M<X> = new A
```

Here, TypeScript's reduction algorithm fails to recognize that `new A` can be typed as `M<X>`, even though `X` is clearly a subtype of `string` in `f`'s body.

Although the situation is concerning, it might not be as bad as it seems given that soundness is a non-goal of TypeScript's type system [Bierman et al., 2014]. Nevertheless, we believe that the results of this chapter are directly applicable to conditional types and could be used to improve TypeScript's type checker.

4.7 Conclusion

In this chapter, we introduced *match types*, a lightweight mechanism for type-level programming that integrates seamlessly in subtyping-based programming languages. We formalized match types in System FM, a calculus based on System $F_{<:}$, and proved it sound. Furthermore, we implemented match types in the Scala 3 compiler, making them readily available to a large audience of programmers. A key insight for sound match types is the notion of disjointness, which complements subtyping in the match type reduction algorithm. In the future, we plan to investigate inference of match types to avoid code duplication in programs that operate both at the term and the type level.

5 Performance Evaluation

In this chapter, we evaluate the performance of the various type-level programming techniques introduced in prior chapters. Specifically, we are interested in the scalability of each technique when confronted with large type-level programs, in terms of compilation times (Section 5.2) and binary sizes (Section 5.3). In Section 5.4, we discuss how the timing of match type reduction impacts compilation times.

5.1 Method

We evaluate the performance of the 3 type-level programming techniques presented in this dissertation by comparing them on a set of 4 benchmarks. Each benchmark consists of a simple type-level operation, which we implemented once using implicits (Chapter 2), once using generalized singletons (Chapter 3), and once using match types (Chapter 4).

These benchmarks are inspired by code examples used throughout this dissertation. They all follow a similar pattern of a single function call whose result type is computed at the type level. We code generated variations of each benchmark with increasing input size, from 1 to 256, in increments of 8. The source code of our benchmarks is available in the supplementary material of this dissertation[1], alongside instructions on how to reproduce our experiments.

Here is a brief description of our benchmark suite:

1. **Concat** Concatenates a heterogeneous list with itself.

2. **Remove** Removes the last element of a heterogeneous list (Section 2.2).

3. **Join** Computes the resulting schema after performing a self-join along the last column of a database table (Subsection 3.2.1).

4. **Reduce** Computes the dimension of a multidimensional array after dimensional reduction along all axes (Subsection 4.5.4).

We run the experiments presented in this chapter on an i7-7700K processor running OpenJDK 1.8.0_212 and Linux. For the implicit and match type benchmarks, we use the latest Scala 3 release (version 3.1.2 at the time of writing). For the generalized singleton bench-

[1] `https://olivierblanvillain.github.io/thesis/benchmarks.zip`

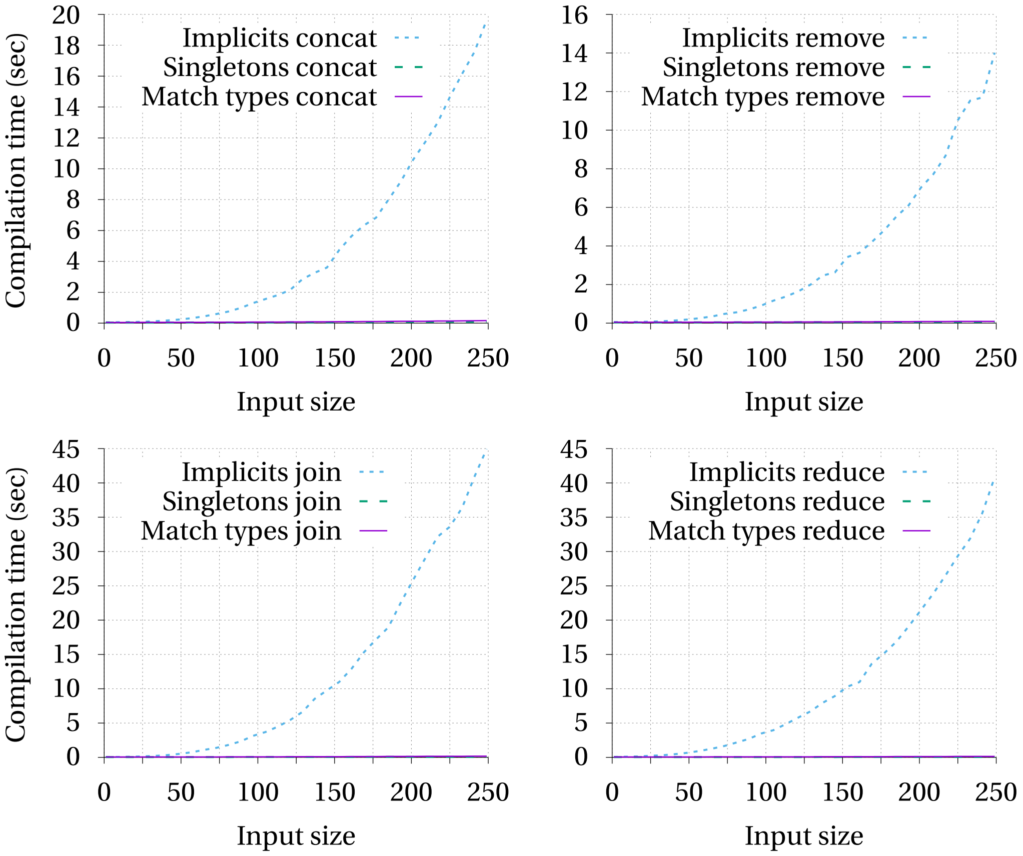

Figure 5.1: A comparison of the compilation time of implicits, generalized singletons and match types for our benchmark suite (lower is better).

marks, we use our prototype implementation, which lives in a branch of the dotty-staging repository[2].

5.2 Compilation time

Figure 5.1 compares the compilation time of implicits, generalized singletons and match types on our benchmark suite. The compilation time of implicit-based benchmarks dominates the graphs on each figures, and suggests a quadratic complexity. These results imply that implicits are ill-suited for large scale type-level computations: spending tens of seconds on a single operation is clearly out of the question for practical applications.

The same measurements are shown in Figure 5.2, but with millisecond y-axes, to focus on the comparison of non-implicit curves. Each graph follows a similar trend, with singleton-based implementations outperforming their match-type-based counterparts by about a factor of two. We speculate that this difference is due to the overhead of subtyping, which in inherent to the design of match type. Our prototype for generalized singletons uses a dedi-

[2] `git clone git@github.com:dotty-staging/dotty.git --branch add-transparent-7`

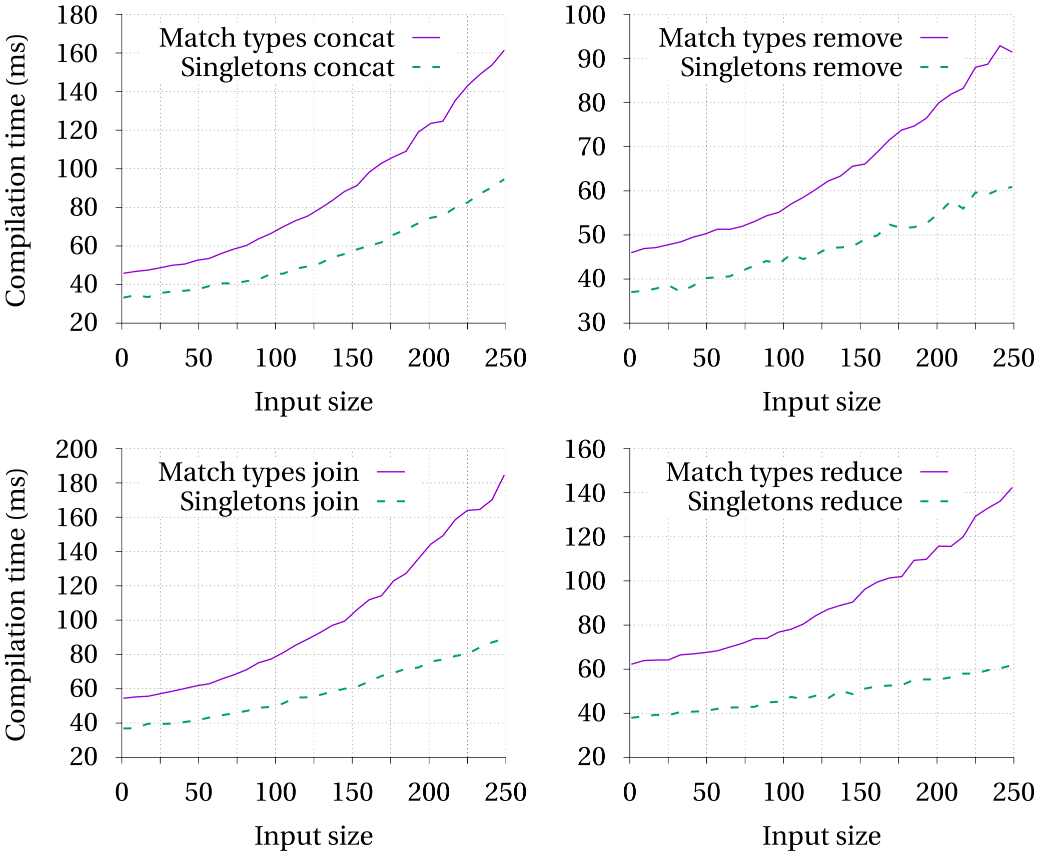

Figure 5.2: A comparison of the compilation time of generalized singletons and match types for our benchmark suite (lower is better).

cated type evaluation procedure that is specialized for the task and thus faster.

We obtained the data points of Figure 5.1 and 5.2 using Java Microbenchmark Harness (JMH) with 60 seconds of warmup, 60 seconds of measurement and a single JVM fork. As a result, the resulting measurements are precise (99.9% confidence intervals vary between ± 0.5 and ± 2 milliseconds), but subject to variations caused by the unpredictability of the JVM's just in time compiler, which explains the bumps visible in Figure 5.2.

5.3 Binary size

In Table 5.1, we show the amount of additional JVM Bytecode generated per benchmark size increment. The binary sizes of singleton- and match-type-based implementations are stable with respect to benchmark input size, which is expected given that our benchmarks consist of type-level-only operations.

The binary sizes of implicit-based implementations grow linearly with the input size, by 6 bytes per input size increment for smaller benchmarks (Concat and Remove), and by 12 and 18 bytes for the larger benchmarks (Reduce and Join, respectively). To put those numbers in

Table 5.1: Additional JVM Bytecode generated per input size increment (in bytes).

	Implicits	Singletons	Match Types
Concat	6	0	0
Remove	6	0	0
Reduce	12	0	0
Join	18	0	0

perspective, they represent an overall binary size increase of 9%, 9%, 12%, and 20%, when comparing the size-1 instantiation to the size-256 instantiation of Concat, Remove, Reduce, and Join, respectively. These increases in binary size are caused by the implicit evidences synthesized by the compiler during implicit resolution. In our benchmarks, those evidences have no runtime purpose and are simply an artifact of the mechanism in play. Other applications, such as those supported by the Shapeless library [Sabin and Shapeless contributors, 2011–2022], take advantage of implicit evidence synthesis to provide additional runtime functionalities.

5.4 The Timing of Match Type Reductions

In Chapter 4, we describe the match type reduction algorithm by adding new subtyping rules to System $F_{<:}$ (Subsection 4.3.2). The resulting system is entirely declarative: it doesn't specify *when* to apply those rules. In this section, we discuss how the timing of match type reduction impacts compilation times.

A correct type system implementation must attempt to reduce match types *as late as possible*, during subtyping. Indeed, failing to do so could lead to incorrect results in cases where a match type is reducible in the context of the subtyping query but is not reducible in any prior context.

In addition to the reductions attempted during subtyping, the compiler can try to reduce match types earlier in the pipeline, essentially at any point in time. Doing so is always correct, since the types before and after reduction are mutually subtypes of each other. Each early attempt of match type reduction is a trade-off: it can either be a small overhead, if the reduction fails, a no-op, if the reduction succeeds but the result is used exactly once, or an optimization, if the reduction succeeds and the result is used more than once. Note that reducing match types in a lazy fashion would not be a replacement for early reduction attempts, because the subtyping and disjointness checks are context-dependent (Subsection 4.4.9).

Let us consider an example that illustrates the benefits of early reduction attempts:

```
// GEQ[A, B] compiles if A is greater than B
type GEQ[A <: Int, B <: Int] = A match {
  case B => true
  case _ => GEQ[A, B + 1]
}
```

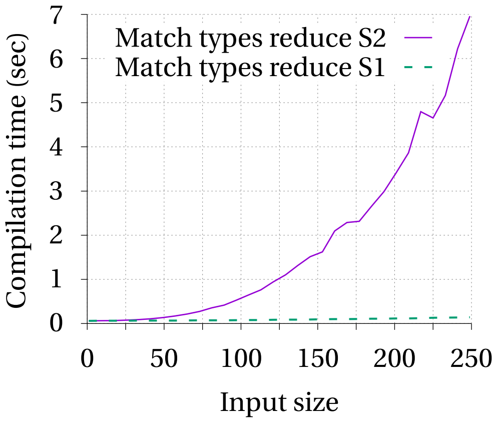

Figure 5.3: A comparison of the compilation time of the match type Reduce benchmark with two different match type reduction strategies. S1 is the reduction strategy we implemented in the Scala 3 compiler. S2 is a variation of that strategy where we disabled the reduction of match types after type parameter instantiation.

Without any early reductions, this definition will result in a quadratic number of additions. For example, here is the sequence of subtyping queries issued while compiling `GEQ[5,1]`:

```
5 <: 1? no
5 <: 1+1? no
5 <: 1+1+1? no
5 <: 1+1+1+1? no
5 <: 1+1+1+1+1? yes
```

The first instance of `1+1` comes from the right-hand-side of the first `GEQ` call (with `B=1`), but that type is used in all the subsequent subtyping queries. As a result, reducing `1+1` to `2` once and for all leads to a speed-up in compilation.

Our Scala 3 implementation reduces match types at multiple points during compilation, including right after creating types, during subtyping, and after instantiating parameters of type applications. This last point is essential to avoid the quadratic behavior of the `GEQ` example. In Figure 5.3, we show the impact of this optimization on our Reduce benchmark. In this figure, we compare the compilation times of the match type Reduce benchmark with (S1) and without (S2) the reduction of match types after type parameter instantiation.

6 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

A Type and Null Safety for Regexes

Abstract...

A.1 Introduction

The Scala standard library includes a package to manipulate regular expressions. This package is based on Java's implementation of regular expression, and thus benefits from the high performance of the JVM's regex engine. The Scala implementation innovates with its presentation: it uses the power and flexibility of Scala's pattern matching to offer a syntax that is both elegant and concise. The package's documentation starts with the following example:

```
val date = Regex("(\\d{4})-(\\d{2})-(\\d{2})")
"2004-01-20" match
  case date(y, m, d) =>
    s"$y was a good year for PLs."
```

The extractor pattern, `case date(y,m,d) =>`, replaces the need for manually indexing into the regular expression's capture groups, and shields its users from off-by-one error.

While the syntax of this example would undoubtedly make some non-Scala programmers envious, its type safety leaves much to be desired. First of all, the *number* of variable bindings in the extractor is entirely opaque to Scala's type system, and left to the discretion of the programmer. Furthermore, the values coming out of capture groups can be null (when using optional captures) and thus require an additional layer of validation. Those shortcomings might appear benign when looking at small examples, but can easily turn into bugs when dealing with a large-scale code base.

In this chapter, we propose a new design for Scala's regular expression library which provides both type-safe and null-safe capture group extraction. Our design makes extensive use of match types, Scala 3 new feature for type-level programming, to statically analyze regular expressions during type checking and generate type-specialized extractors. We build our interface to mimic Scala's original regular expression API so that Scala programmers can use it as a drop-in replacement while still enjoying additional safety.

A.2 Architecture

We present our library in terms of 3 components:

1. The type-level capture group analysis, which uses match types to inspect the user-provided regular expression and compute the corresponding tuple-based type representation (Appendix A.3).

2. The term-level capture group processing, which extracts and sanitizes the output of the regex engine, in accordance with the representation computed by the type-level component (Appendix A.4).

3. The user-level interface, which ties everything together to compile, execute, and extract the results of a regular expression, while providing type-safe and null-safe capture group extraction.

We begin our presentation with the user-level interface, which takes the form of a relatively simple Scala class and companion object:

```
object Regex:
  def apply[R <: String & Singleton](regex: R): Regex[Compile[R]] =
    new Regex(regex)

class Regex[P] private (val regex: String):
  val pattern = Pattern.compile(regex)
  def unapply(s: String): Option[P] =
    val m = pattern.matcher(s)
    if (m.matches())
      val a = Array.tabulate(m.groupCount) { i => m.group(i + 1) }
      Some(toTuple[P](regex, a))
    else
      None
```

The `apply` method is the entry point of our library. It takes a regular expression and returns an instance of `Regex`. Instead of directly accepting `String` argument, that method requires a type parameter, `R`, which will be inferred by the compiler, and allows us to get our hands on a type-level instance of the regular expression (the `Singleton` bound is necessary to guide Scala's type inference). We use that type-level string immediately to instantiate `Compile`, which is our type-level capture group analysis component. We store the output of that analysis as `Regex`'s type argument.

The implementation of the `Regex` class is straightforward. Similarly to its counterpart in the original Scala implementation, that class uses an instance of `java.util.regex.Pattern` to match the given input against the regular expression, and to extract the capture groups whenever the matching succeeds. The `Regex` class defines an `unapply` method, which the Scala compiler will invoke when using an instance of that class as a pattern matching extractor [Emir et al., 2007].

The main difference between Scala's original Regex implementation and ours lies in the signature of the unapply method, which determines the nature of extraction performed by that method. By returning an optional tuple, we tell the compiler that a successful matching consists of exactly *n* capture groups, where *n* corresponds to the tuple arity, and that the type of those groups will correspond to the tuple types. We take advantage of that flexibility to enrich the results of Java's regex engine by wrapping nullable values into options. In our implementation, this is done via the toTuple method, which is our term-level capture group processing component. As a result, our implementation never returns null values, which removes the null-checking burden from the user.

For instance, in the following example, we use a regular expression with an optional capture group to extract the integral and fractional parts of a rational number:

```
val rational = Regex("(\\d+)\\.?(\\d+)?")
"3.1415" match
  case rational(i, Some(f)) =>
    val n = i.size + f.size
    s"This number is $n digits long"
```

Here, we instantiates Regex's type parameter to P = (String, Option[String]), which we then use in unapply's return type to obtain a precise representation of the regex's capture groups.

A.3 Type-Level

The purpose of our type-level component is two-fold:

1. Identify the capture groups of the input regular expression.

2. Determine which of those capture groups are optional, that is, whether or not the regex engine can assign null values for each of those groups.

The result of that type-level computation takes the form of a Scala tuple with String and Option[String] element, depending of the optionality analysis.

We present our implementation incrementally, starting from a simple but incomplete solution, and progressively building up towards our final solution. The first version of our type-level program is limited to capture group identification. The implementation is straightforward:

```
import compiletime.ops.string.{Length, CharAt}
import compiletime.ops.int.+

type Compile[R <: String] =
  Reverse[Loop[R, 0, Length[R], EmptyTuple]]

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

The `Length` and `CharAt` types are special-cased by the compiler: when their first argument is a literal singleton string, the compiler evaluates those types via their corresponding term-level implementation. Similarly, the `+` type from `compiletime.ops.int` allows us to manipulate integer just like we would at the term level. The `EmptyTuple` and `*:` are Scala 3's new generic representation of tuples. Those types remove the need for heterogeneous lists and allow us to manipulate tuples directly.

Attentive readers might have noticed that our handling of backslash, regex's escape character, might lead to off-by-one errors. Be reassured, this is not an oversight! Regular expressions with trailing backslashes result in runtime exceptions (reported as "unexpected internal error"). Even though detecting these types of errors at compile time is out of the scope of our implementation, doing so is still desirable. If we invoke `Compile` on a regular expression with a trailing backslash, the compiler will run into a call to `charAt` with an out of bounds argument, catch the corresponding exception and report it as a compilation error.

Our first implementation of `Compile` treats every opening parenthesis as the start of a capture group, which does not do honor to the syntax of Java's regular expression. Indeed, Java also support several other special constructs which start with an opening parethesis, such as non-capturing groups, `(?:X)`, lookaheads, `(?=X)` & `(?!X)`, and lookbehinds, `(?<=X)` & `(?<!X)`, and must all be discriminated from named and unnamed-capturing groups, `(?<name>X)` & `(X)`.

```
type IsCapturing[R <: String, At <: Int] =
  CharAt[R, At] match
    case "?" => CharAt[R, At + 1] match
      case "<" => CharAt[R, At + 2] match
        case "!" => false // zero-width negative lookbehind
        case _ => true    // named-capturing group
      case _ => false     // other non-capturing special constructs
    case _ => true        // unnamed-capturing group
```

A.4 Term-Level

In this section...

A.5 Evaluation

In this section...

B Type Soundness for System FM

Lemma B.1 (Permutation).

If Γ *and* Δ *are well-formed and* Δ *is a permutation of* Γ*, then:*

1. *If* $\Gamma \vdash \mathrm{disj}(S,T)$*, then* $\Delta \vdash \mathrm{disj}(S,T)$.

2. *If* $\Gamma \vdash S <: T$*, then* $\Delta \vdash S <: T$.

3. *If* $\Gamma \vdash t : T$*, then* $\Delta \vdash t : T$.

Proof: We prove 1. and 2. simultaneously by induction on two derivations of $\Gamma \vdash \mathrm{disj}(V,W)$ and $\Gamma \vdash S <: T$. More precisely, the induction is done on the cumulative depth of both derivation tree.

1. $\Gamma \vdash \mathrm{disj}(V,W)$

   - *Case* D-XI: $\quad V = C_1 \quad W = C_2 \quad (C_1, C_2) \in \Xi$
     Using D-XI with context Δ directly leads to the desired result.
   - *Case* D-PSI: $\quad V = \{\mathit{new}\, C_1\} \quad W = C_2 \quad (C_1, C_2) \notin \Psi$
     Using D-PSI with context Δ directly leads to the desired result.
   - *Case* D-SUB: $\quad \Gamma \vdash V <: U \quad \Gamma \vdash \mathrm{disj}(U,W)$
     By the IH we get $\Delta \vdash \mathrm{disj}(U,W)$. Using the 2nd part of the lemma we obtain $\Delta \vdash V <: U$. The result follows from D-SUB.
   - *Case* D-ARROW: $\quad V = V_1 \to V_2 \quad W = C$
     Using D-ARROW with context Δ directly leads to the desired result.
   - *Case* D-ALL: $\quad V = \forall X <: V_1 .\, V_2 \quad W = C$
     Using D-ALL with context Δ directly leads to the desired result.

2. $\Gamma \vdash S <: T$.

   - *Case* S-REFL: $\quad T = S$
     Using S-REFL with context Δ directly leads to the desired result.
   - *Case* S-TRANS: $\quad \Gamma \vdash S <: U \quad \Gamma \vdash U <: T$
     The result follows directly from the IH and S-TRANS.

- *Case* S-Top: $\mathrm{T} = \mathrm{Top}$

  Using S-Top with context Δ directly leads to the desired result.

- *Case* S-Sin: $\mathrm{S} = \{\mathit{new}\,\mathrm{C}\}$ $\quad \mathrm{T} = \mathrm{C}$

  Using S-Sin with context Δ directly leads to the desired result.

- *Case* S-TVar: $\mathrm{S} = \mathrm{X} \quad \mathrm{X} <: \mathrm{T} \in \Gamma$ Since Δ is a permutation of Γ, $\mathrm{X} <: \mathrm{T} \in \Delta$, and the result follows from S-TVar.

- *Case* S-Arrow: $\mathrm{S} = \mathrm{S}_1 \to \mathrm{S}_2 \quad \mathrm{T} = \mathrm{T}_1 \to \mathrm{T}_2$
  $\Gamma \vdash \mathrm{T}_1 <: \mathrm{S}_1 \quad \Gamma \vdash \mathrm{S}_2 <: \mathrm{T}_2$

  The result follows directly from the IH and S-Arrow.

- *Case* S-All: $\mathrm{S} = \forall \mathrm{X} <: \mathrm{U}_1.\, \mathrm{S}_2 \quad \mathrm{T} = \forall \mathrm{X} <: \mathrm{U}_1.\, \mathrm{T}_2 \quad \Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$

  If Δ is a permutation of Γ, then $\Delta, \mathrm{X} <: \mathrm{U}_1$ is a permutation of $\Gamma, \mathrm{X} <: \mathrm{U}_1$. Therefore, we can use the IH to get $\Delta, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$. The result follows from S-All.

- *Case* S-Psi: $\mathrm{S} = \mathrm{C}_1 \quad \mathrm{T} = \mathrm{C}_2 \quad (\mathrm{C}_1, \mathrm{C}_2) \in \Psi$

  Using S-Psi with context Δ directly leads to the desired result.

- *Case* S-Match1/2: $\mathrm{T}_1 = \mathrm{T}_n \quad \mathrm{T}_2 = \mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\,\mathit{or}\,\mathrm{T}_d$
  $\Gamma \vdash \mathrm{T}_s <: \mathrm{S}_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)$

  By the 1st part of the lemma we get $\forall m < n.\ \Delta \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)$. From the IH we obtain $\Delta \vdash \mathrm{T}_s <: \mathrm{S}_n$. The result follows from S-Match1/2.

- *Case* S-Match3/4: $\mathrm{S} = \mathrm{T}_d \quad \mathrm{T} = \mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\,\mathit{or}\,\mathrm{T}_d \quad \forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_n)$

  The result follows from the 1st part of the lemma and S-Match3/4.

- *Case* S-Match5: $\mathrm{S} = \mathrm{S}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{S}_i\}\,\mathit{or}\,\mathrm{S}_d \quad \mathrm{T} = \mathrm{T}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{T}_i\}\,\mathit{or}\,\mathrm{T}_d$
  $\Gamma \vdash \mathrm{S}_s <: \mathrm{T}_s \quad \forall n.\ \Gamma \vdash \mathrm{S}_n <: \mathrm{T}_n \quad \Gamma \vdash \mathrm{S}_d <: \mathrm{T}_d$

  The result follows directly from the IH and S-Match5.

3. By induction on a derivation of $\Gamma \vdash \mathrm{t} : \mathrm{T}$

   - *Case* T-Var: $\mathrm{t} = \mathrm{x} \quad \mathrm{x} : \mathrm{T} \in \Gamma$

     Since Δ is a permutation of Γ, $\mathrm{x} : \mathrm{T} \in \Delta$, and the result follows from T-Var.

   - *Case* T-Abs: $\mathrm{t} = \lambda \mathrm{x} : \mathrm{T}_1 \mathrm{t}_2 \quad \mathrm{T} = \mathrm{T}_1 \to \mathrm{T}_2 \quad \Gamma, \mathrm{x} : \mathrm{T}_1 \vdash \mathrm{t}_2 : \mathrm{T}_2$

     If Δ is a permutation of Γ, then $\Delta, \mathrm{x} : \mathrm{T}_1$ is a permutation of $\Gamma, \mathrm{x} : \mathrm{T}_1$. Therefore, we can use the IH to get $\Delta, \mathrm{x} : \mathrm{T}_1 \vdash \mathrm{t}_2 : \mathrm{T}_2$. The result follows from T-Abs.

   - *Case* T-App: $\mathrm{t} = \mathrm{t}_1 \mathrm{t}_2 \quad \mathrm{T} = \mathrm{T}_{12} \quad \Gamma \vdash \mathrm{t}_1 : \mathrm{T}_{11} \to \mathrm{T}_{12} \quad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_{11}$

     The result follows directly from the IH and T-App.

   - *Case* T-TAbs: $\mathrm{t} = \lambda \mathrm{X} <: \mathrm{U}_1.\, \mathrm{t}_2 \quad \mathrm{T} = \forall \mathrm{X} <: \mathrm{U}_1.\, \mathrm{T}_2 \quad \Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{t}_2 : \mathrm{T}_2$

     If Δ is a permutation of Γ, then $\Delta, \mathrm{X} <: \mathrm{U}_1$ is a permutation of $\Gamma, \mathrm{X} <: \mathrm{U}_1$. Therefore, we can use the IH to get $\Delta, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{t}_2 : \mathrm{T}_2$. The result follows from T-TAbs.

   - *Case* T-TApp: $\mathrm{t} = \mathrm{t}_1 \mathrm{T}_2 \quad \mathrm{T} = [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{T}_{12}$
     $\Gamma \vdash \mathrm{t}_1 : (\forall \mathrm{X} <: \mathrm{U}_1.\, \mathrm{T}_{12}) \quad \Gamma \vdash \mathrm{T}_2 <: \mathrm{U}_1$

     By the IH we get $\Delta \vdash \mathrm{t}_1 : (\forall \mathrm{X} <: \mathrm{U}_1.\, \mathrm{T}_{12})$. Using the 2nd part of the lemma we obtain $\Delta \vdash \mathrm{T}_2 <: \mathrm{U}_1$. The result follows from T-TApp.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  By the IH we get $\Delta \vdash t : S$. Using the 2nd part of the lemma we obtain $\Delta \vdash S <: T$. The result follows from T-SUB.

- *Case* T-CLASS: $t = \textit{new}\, C \quad T = \{\textit{new}\, C\}$

  Using T-CLASS with context Δ directly leads to the desired result.

- *Case* T-MATCH: $t = t_s\ \textit{match}\{x_i : C_i \Rightarrow t_i\}\,\textit{or}\ t_d \quad T = T_s\ \textit{match}\{C_i \Rightarrow T_i\}\,\textit{or}\ T_d$
  $\Gamma \vdash t_s : T_s \quad \Gamma, x_i : C_i \vdash t_i : T_i \quad \Gamma \vdash t_d : T_d$

  If Δ is a permutation of Γ, then $\Delta, x_i : C_i$ is a permutation of $\Gamma, x_i : C_i$. Therefore the result follows directly from the IH and T-MATCH.

□

Lemma B.2 (Weakening).

1. *If* $\Gamma \vdash \text{disj}(S,T)$ *and* $\Gamma, X <: U$ *is well formed, then* $\Gamma, X <: U \vdash \text{disj}(S,T)$.
2. *If* $\Gamma \vdash S <: T$ *and* $\Gamma, X <: U$ *is well formed, then* $\Gamma, X <: U \vdash S <: T$.
3. *If* $\Gamma \vdash S <: T$ *and* $\Gamma, x : U$ *is well formed, then* $\Gamma, x : U \vdash S <: T$.
4. *If* $\Gamma \vdash t : T$ *and* $\Gamma, x : U$ *is well formed, then* $\Gamma, x : U \vdash t : T$.
5. *If* $\Gamma \vdash t : T$ *and* $\Gamma, X <: U$ *is well formed, then* $\Gamma, X <: U \vdash t : T$.

Proof: We prove 1. and 2. simultaneously by induction on two derivations of $\Gamma \vdash \text{disj}(V,W)$ and $\Gamma \vdash S <: T$. More precisely, the induction is done on the cumulative depth of both derivation tree.

1. $\Gamma \vdash \text{disj}(V,W)$

   - *Case* D-XI: $V = C_1 \quad W = C_2 \quad (C_1, C_2) \in \Xi$

     Using D-XI with context $\Gamma, X <: U$ directly leads to the desired result.

   - *Case* D-PSI: $V = \{\textit{new}\, C_1\} \quad W = C_2 \quad (C_1, C_2) \notin \Psi$

     Using D-PSI with context $\Gamma, X <: U$ directly leads to the desired result.

   - *Case* D-SUB: $\Gamma \vdash V <: U \quad \Gamma \vdash \text{disj}(U,W)$

     By the IH we get $\Gamma \vdash \text{disj}(U,W)$. Using the 2nd part of the lemma we obtain $\Gamma, X <: U \vdash V <: U$. The result follows from D-SUB.

   - *Case* D-ARROW: $V = V_1 \rightarrow V_2 \quad W = C$

     Using D-ARROW with context $\Gamma, X <: U$ directly leads to the desired result.

   - *Case* D-ALL: $V = \forall X <: V_1 .\, V_2 \quad W = C$

     Using D-ALL with context $\Gamma, X <: U$ directly leads to the desired result.

2. $\Gamma \vdash S <: T$.

- *Case* S-REFL: $\mathrm{T = S}$

  Using S-REFL with context $\Gamma, \mathrm{X <: U}$ directly leads to the desired result.

- *Case* S-TRANS: $\Gamma \vdash \mathrm{S <: U} \quad \Gamma \vdash \mathrm{U <: T}$

  The result follows from the IH and S-TRANS.

- *Case* S-TOP: $\mathrm{T = Top}$

  Using S-TOP with context $\Gamma, \mathrm{X <: U}$ directly leads to the desired result.

- *Case* S-SIN: $\mathrm{S} = \{\mathit{new}\,\mathrm{C}\} \quad \mathrm{T = C}$

  Using S-SIN with context $\Gamma, \mathrm{X <: U}$ directly leads to the desired result.

- *Case* S-TVAR: $\mathrm{S = Y} \quad \mathrm{Y <: T} \in \Gamma$

  If $\mathrm{Y <: T} \in \Gamma$, then $\mathrm{Y <: T} \in \Gamma, \mathrm{X <: U}$ and the result follows from S-TVAR.

- *Case* S-ARROW: $\mathrm{S = S_1 \to S_2} \quad \mathrm{T = T_1 \to T_2}$
  $\Gamma \vdash \mathrm{T_1 <: S_1} \quad \Gamma \vdash \mathrm{S_2 <: T_2}$

  The result follows from the IH and S-ARROW.

- *Case* S-ALL: $\mathrm{S} = \forall \mathrm{Y <: U_1 .\, S_2} \quad \mathrm{T} = \forall \mathrm{Y <: U_1 .\, T_2} \quad \Gamma, \mathrm{Y <: U_1} \vdash \mathrm{S_2 <: T_2}$

  Using the IH with context $\Gamma_{IH} = \Gamma, \mathrm{Y <: U_1}$, we get $\Gamma, \mathrm{Y <: U_1}, \mathrm{X <: U} \vdash \mathrm{S_2 <: T_2}$. From Lemma B.1, $\Gamma, \mathrm{X <: U}, \mathrm{Y <: U_1} \vdash \mathrm{S_2 <: T_2}$ The result follows from S-ALL.

- *Case* S-PSI: $\mathrm{S = C_1} \quad \mathrm{T = C_2} \quad (\mathrm{C_1, C_2}) \in \Psi$

  Using S-PSI with context $\Gamma, \mathrm{X <: U}$ directly leads to the desired result.

- *Case* S-MATCH1/2: $\mathrm{T_1 = T_n} \quad \mathrm{T_2 = T_s}\ \mathit{match}\{\mathrm{S_i \Rightarrow T_i}\}\mathit{or}\,\mathrm{T_d}$
  $\Gamma \vdash \mathrm{T_s <: S_n} \quad \forall m < n.\, \Gamma \vdash \mathrm{disj}(\mathrm{T_s, S_m})$

  From the IH we get $\Gamma, \mathrm{X <: U} \vdash \mathrm{T_s <: S_n}$. Using the 1st part of the lemma we obtain $\forall m < n.\ \Gamma, \mathrm{X <: U} \vdash \mathrm{disj}(\mathrm{T_s, S_m})$. The result follows from S-MATCH1/2.

- *Case* S-MATCH3/4: $\mathrm{T_1 = T_d} \quad \mathrm{T_2 = T_s}\ \mathit{match}\{\mathrm{S_i \Rightarrow T_i}\}\mathit{or}\,\mathrm{T_d} \quad \forall n.\, \Gamma \vdash \mathrm{disj}(\mathrm{T_s, S_n})$

  Using the 1st part of the lemma we get $\forall n.\ \Gamma, \mathrm{X <: U} \vdash \mathrm{disj}(\mathrm{T_s, S_n})$. The result follows from S-MATCH3/4.

- *Case* S-MATCH5: $\mathrm{S = S_s}\ \mathit{match}\{\mathrm{S_i \Rightarrow T_i}\}\mathit{or}\,\mathrm{T_d} \quad \mathrm{T = T_s}\ \mathit{match}\{\mathrm{S_i \Rightarrow U_i}\}\mathit{or}\,\mathrm{U_d}$
  $\Gamma \vdash \mathrm{S_s <: T_s} \quad \forall n.\, \Gamma \vdash \mathrm{T_n <: U_n} \quad \Gamma \vdash \mathrm{T_d <: U_d}$

  We use the IH on each premise and the result follows directly from S-MATCH5.

3. By inspection of the subtyping rules, it is clear that typing assumbtions play no role in subtyping derivations.

4. By induction on a derivation of $\Gamma \vdash \mathrm{t : T}$.

- *Case* T-VAR: $\mathrm{t = y} \quad \mathrm{y : T} \in \Gamma$

  If $\mathrm{y : T} \in \Gamma$ then $\mathrm{y : T} \in \Gamma, \mathrm{x : U}$ and the result follows from T-VAR.

- *Case* T-ABS: $\mathrm{t} = \lambda \mathrm{y : T_1 t_2} \quad \mathrm{T = T_1 \to T_2} \quad \Gamma, \mathrm{y : T_1} \vdash \mathrm{t_2 : T_2}$

  Using the IH with $\Gamma_{IH} = \Gamma, \mathrm{y : T_1}$ we get $\Gamma, \mathrm{y : T_1}, \mathrm{x : U} \vdash \mathrm{t_2 : T_2}$. From Lemma B.1, $\Gamma, \mathrm{x : U}, \mathrm{y : T_1} \vdash \mathrm{t_2 : T_2}$. The result follows from T-ABS.

- *Case* T-APP: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{12}$

  We use the IH on each premise and the result follows from T-APP.

- *Case* T-TABS: $t = \lambda X <: T_1 .\, t_2 \quad T = \forall X <: T_1 .\, T_2 \quad \Gamma, X <: T_1 \vdash t_2 : T_2$

  Using the IH with $\Gamma_{IH} = \Gamma, X <: T_1$ we get $\Gamma, X <: T_1, x : U \vdash t_2 : T_2$. From Lemma B.1, $\Gamma, x : U, X <: T_1 \vdash t_2 : T_2$ The result follows from T-TABS.

- *Case* T-TAPP: $t = t_1 T_2 \quad T = [X \mapsto T_2] T_{12}$
  $\Gamma \vdash t_1 : (\forall X <: T_{11} .\, T_{12}) \quad \Gamma \vdash T_2 <: T_{11}$

  Using the IH on the left premise we get $\Gamma, x : U \vdash t_1 : (\forall X <: T_{11} .\, T_{12})$. Using the 3rd part of the lemma on the right premise we obtain $\Gamma, x : U \vdash T_2 <: T_{11}$. The result follows from T-TAPP.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  Using the IH on the left premise we get $\Gamma, x : U \vdash t : S$. Using the 3rd part of the lemma on the right premise we obtain $\Gamma, x : U \vdash S <: T$. The result follows from T-SUB.

- *Case* T-CLASS: $t = \mathit{new}\, C \quad T = \{\mathit{new}\, C\}$

  Using T-CLASS with context $\Gamma, x : U$ directly leads to the desired result.

- *Case* T-MATCH: $t = t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\ \mathit{or}\ t_d \quad T = T_s\ \mathit{match}\{C_i \Rightarrow T_i\}\ \mathit{or}\ T_d$
  $\Gamma \vdash t_s : T_s \quad \Gamma, x_i : C_i \vdash t_i : T_i \quad \Gamma \vdash t_d : T_d$

  We use the IH on each premise and the result follows directly from Lemma B.1 and T-MATCH.

5. By induction on a derivation of $\Gamma \vdash t : T$.

- *Case* T-VAR: $t = x \quad x : T \in \Gamma$

  If $y : T \in \Gamma$ then $y : T \in \Gamma, X <: U$ and the result follows from T-VAR.

- *Case* T-ABS: $t = \lambda x : T_1 t_2 \quad T = T_1 \to T_2 \quad \Gamma, x : T_1 \vdash t_2 : T_2$

  Using the IH with $\Gamma_{IH} = \Gamma, x : T_1$ we get $\Gamma, x : T_1, X <: U \vdash t_2 : T_2$. From Lemma B.1, $\Gamma, X <: U, x : T_1 \vdash t_2 : T_2$. The result follows from T-ABS.

- *Case* T-APP: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{12}$

  We use the IH on each premise and the result follows from T-APP.

- *Case* T-TABS: $t = \lambda Y <: T_1 .\, t_2 \quad T = \forall Y <: T_1 .\, T_2 \quad \Gamma, Y <: T_1 \vdash t_2 : T_2$

  Using the IH with $\Gamma_{IH} = \Gamma, Y <: T_1$ we get $\Gamma, Y <: T_1, X <: U \vdash t_2 : T_2$. From Lemma B.1, $\Gamma, X <: U, Y <: T_1, \vdash t_2 : T_2$. The result follows from T-TABS.

- *Case* T-TAPP: $t = t_1 T_2 \quad T = [Y \mapsto T_2] T_{12}$
  $\Gamma \vdash t_1 : (\forall Y <: T_{11} .\, T_{12}) \quad \Gamma \vdash T_2 <: T_{11}$

  Using the IH on the left premise we get $\Gamma, X <: U \vdash t_1 : (\forall X <: T_{11} .\, T_{12})$. Using the 2nd part of the lemma on the right premise we obtain $\Gamma, X <: U \vdash T_2 <: T_{11}$. The result follows from T-TAPP.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  Using the IH on the left premise we get $\Gamma, S <: U \vdash t : S$. Using the 2nd part of the lemma on the right premise we obtain $\Gamma, X <: U \vdash S <: T$. The result follows from T-SUB.

- *Case* T-CLASS: $t = \textit{new}\, C \quad T = \{\textit{new}\, C\}$

  Using T-CLASS with context $\Gamma, X <: U$ directly leads to the desired result.

- *Case* T-MATCH: $t = t_s\ \textit{match}\{x_i : C_i \Rightarrow t_i\}\textit{or}\ t_d \quad T = T_s\ \textit{match}\{C_i \Rightarrow T_i\}\textit{or}\ T_d$
  $\Gamma \vdash t_s : T_s \quad \Gamma, x_i : C_i \vdash t_i : T_i \quad \Gamma \vdash t_d : T_d$

  We use the IH on each premise and the result follows directly from Lemma B.1 and T-MATCH.

□

Lemma B.3 (Strengthening).
If $\Gamma, x : T, \Delta \vdash S <: T$*, then* $\Gamma, \Delta \vdash S <: T$.

Proof: By inspection of the subtyping rules, it is clear that typing assumbtions play no role in subtyping derivations.

□

Lemma B.4 (Substitution).

1. *If* $\Gamma, X <: Q, \Delta \vdash \text{disj}(S, T)$ *and* $\Gamma \vdash P <: Q$*, then* $\Gamma, [X \mapsto P]\Delta \vdash \text{disj}([X \mapsto P]S, [X \mapsto P]T)$.
2. *If* $\Gamma, X <: Q, \Delta \vdash S <: T$ *and* $\Gamma \vdash P <: Q$*, then* $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$.
3. *If* $\Gamma, X <: Q, \Delta \vdash t : T$ *and* $\Gamma \vdash P <: Q$*, then* $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]T$.
4. *If* $\Gamma, x : Q, \Delta \vdash t : T$ *and* $\Gamma \vdash q : Q$*, then* $\Gamma, \Delta \vdash [x \mapsto q]t : T$.

Proof: We prove 1. and 2. simultaneously by induction on two derivations of $\Gamma, X <: Q, \Delta \vdash \text{disj}(V, W)$ and $\Gamma, X <: Q, \Delta \vdash S <: T$. More precisely, the induction is done on the cumulative depth of both derivation tree.

1. $\Gamma, X <: Q, \Delta \vdash \text{disj}(V, W)$

   - *Case* D-XI: $V = C_1 \quad W = C_2 \quad (C_1, C_2) \in \Xi$

     Since $[X \mapsto P]C_1 = C_1$ and $[X \mapsto P]C_2 = C_2$, we can use D-XI with context $\Gamma, [X \mapsto P]\Delta$ to obtain the desired result.

   - *Case* D-PSI: $V = \{\textit{new}\, C_1\} \quad W = C_2 \quad (C_1, C_2) \notin \Psi$

     Since $[X \mapsto P]\{\textit{new}\, C_1\} = \{\textit{new}\, C_1\}$ and $[X \mapsto P]C_2 = C_2$, we can use D-PSI with context $\Gamma, [X \mapsto P]\Delta$ to obtain the desired result.

   - *Case* D-SUB: $\Gamma, X <: Q, \Delta \vdash V <: U \quad \Gamma, X <: Q, \Delta \vdash \text{disj}(U, W)$

     By the IH we get $\Gamma, [X \mapsto P]\Delta \vdash \text{disj}([X \mapsto P]U, [X \mapsto P]W)$. Using the 2nd part of the lemma we obtain $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]V <: [X \mapsto P]U$. The result follows from D-SUB.

- *Case* D-ARROW: $V = V_1 \to V_2 \quad W = C$

  Since $[X \mapsto P](V_1 \to V_2) = [X \mapsto P]V_1 \to [X \mapsto P]V_2$ and $[X \mapsto P]C = C$, we can use D-ARROW with context $\Gamma, [X \mapsto P]\Delta$ to obtain the desired result.

- *Case* D-ALL: $V = \forall Y{<:}V_1.\, V_2 \quad W = C$

  Since $[X \mapsto P](\forall Y{<:}V_1.\, V_2) = \forall Y{<:}[X \mapsto P]V_1.\ [X \mapsto P]V_2$ and $[X \mapsto P]C = C$, we can use D-ALL with context $\Gamma, [X \mapsto P]\Delta$ to obtain the desired result.

2. $\Gamma, X{<:}Q, \Delta \vdash S <: T$.

- *Case* S-REFL: $T = S$

  The result follows directly from S-REFL.

- *Case* S-TRANS: $\Gamma, X{<:}Q, \Delta \vdash S <: U \quad \Gamma, X{<:}Q, \Delta \vdash U <: T$

  The result follows directly from the IH and S-TRANS.

- *Case* S-TOP: $T = \mathrm{Top}$

  $[X \mapsto P]\mathrm{Top} = \mathrm{Top}$ and the result follows from S-TOP.

- *Case* S-SIN: $S = \{\mathit{new}\ C\} \quad T = C$

  $[X \mapsto P]\{\mathit{new}\ C\} = \{\mathit{new}\ C\}$, $[X \mapsto P]C = C$, and the result follows from S-SIN.

- *Case* S-TVAR: $S = Y \quad Y{<:}T \in (\Gamma, X{<:}Q, \Delta)$

  By context well-formedness, $Y{<:}T \in (\Gamma, X{<:}Q, \Delta)$ can be decomposed into 3 subcases:

  - *Subcase* $Y{<:}T \in \Gamma$:

    By context well-formedness X does not appear in Γ consequently is also absent from Y and T. Hence $Y = [X \mapsto P]Y$, $T = [X \mapsto P]T$ and $[X \mapsto P]Y{<:}[X \mapsto P]T \in (\Gamma, [X \mapsto P]\Delta)$. The result follows from S-TVAR.

  - *Subcase* $Y{<:}T \in \Delta$:

    We have $[X \mapsto P]Y{<:}[X \mapsto P]T \in [X \mapsto P]\Delta$ and $[X \mapsto P]Y{<:}[X \mapsto P]T \in (\Gamma, [X \mapsto P]\Delta)$, and the result follows from S-TVAR.

  - *Subcase* $Y{<:}T = X{<:}Q$ (*i.e.* $Y = X$ and $T = Q$):

    By context well-formedness, X doesn't appear in Q, and $[X \mapsto P]T = [X \mapsto P]Q = Q$. Also $[X \mapsto P]S = [X \mapsto P]X = P$ and $[X \mapsto P]T = Q$. As a result, $\Gamma \vdash P <: Q$ implies $\Gamma \vdash [X \mapsto P]S <: [X \mapsto P]T$. Using Lemma B.2 we get $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$, as required.

- *Case* S-ARROW: $S = S_1 \to S_2 \quad T = T_1 \to T_2$

  $\Gamma, X{<:}Q, \Delta \vdash T_1 <: S_1 \quad \Gamma, X{<:}Q, \Delta \vdash S_2 <: T_2$

  $[X \mapsto P](S_1 \to S_2) = [X \mapsto P]S_1 \to [X \mapsto P]S_2$, $[X \mapsto P](T_1 \to T_2) = [X \mapsto P]T_1 \to [X \mapsto P]T_2$. The result follows from the IH and S-ARROW.

- *Case* S-ALL: $S = \forall Y{<:}U_1.\, S_2 \quad T = \forall Y{<:}U_1.\, T_2 \quad \Gamma, X{<:}Q, \Delta, Y{<:}U_1 \vdash S_2 <: T_2$

  We instantiate the IH with $\Delta_{IH} = (\Delta, Y{<:}U_1)$ to obtain $\Gamma, [X \mapsto P]\Delta, Y{<:}[X \mapsto P]U_1 \vdash [X \mapsto P]S_2 <: [X \mapsto P]T_2$. Using S-ALL, we get $\Gamma, [X \mapsto P]\Delta \vdash (\forall Y{<:}[X \mapsto P]U_1.\ [X \mapsto P]S_2) <: (\forall Y{<:}[X \mapsto P]U_1.\ [X \mapsto P]T_2)$, that is, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](\forall Y{<:}U_1.\, S_2) <: [X \mapsto P](\forall Y <: U_1.\, T_2)$, as required.

- *Case* S-Psi: $S = C_1 \quad T = C_2 \quad (C_1, C_2) \in \Psi$

  $[X \mapsto P]C_1 = C_1$ and $[X \mapsto P]C_2 = C_2$. The result follows from S-Psi.

- *Case* S-Match1/2: $T_1 = T_n \quad T_2 = T_s\ \mathit{match}\{S_i \Rightarrow T_i\}\mathit{or}\,T_d$
  $\Gamma, X{<:}Q, \Delta \vdash T_s <: S_n \quad \forall m < n.\ \Gamma, X{<:}Q, \Delta \vdash \mathrm{disj}(T_s, S_m)$

  Using the 1st part of the lemma we get $\forall m < n.\ \Gamma, [X \mapsto P]\Delta \vdash \mathrm{disj}([X \mapsto P]T_s, [X \mapsto P]S_m)$. By the IH we get $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]T_s <: S_n$. $[X \mapsto P]T = [X \mapsto P]T_s\ \mathit{match}\{ [X \mapsto P]S_i \Rightarrow [X \mapsto P]T_i\}\mathit{or}\ [X \mapsto P]T_d$, and the result follows from S-Match1/2.

- *Case* S-Match3/4: $S = T_d \quad T = T_s\ \mathit{match}\{S_i \Rightarrow T_i\}\mathit{or}\,T_d$
  $\forall n.\ \Gamma, X{<:}Q, \Delta \vdash \mathrm{disj}(T_s, S_n)$

  By the 1st part of the lemma we get $\forall n.\ \Gamma, [X \mapsto P]\Delta \vdash \mathrm{disj}([X \mapsto P]T_s, [X \mapsto P]S_n)$ and the result follows from S-Match3/4.

- *Case* S-Match5: $S = S_s\ \mathit{match}\{U_i \Rightarrow S_i\}\mathit{or}\,S_d \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\mathit{or}\,T_d$
  $\Gamma, X{<:}Q, \Delta \vdash S_s <: T_s \quad \forall n.\ \Gamma, X{<:}Q, \Delta \vdash S_n <: T_n$
  $\Gamma, X{<:}Q, \Delta \vdash S_d <: T_d$

  The result follows directly from the IH.

3. By induction on a derivation of $\Gamma, X{<:}Q, \Delta \vdash t : T$.

- *Case* T-Var: $t = x \quad x{:}T \in \Gamma, X{<:}Q, \Delta$

  $[X \mapsto P]x = x$. T-Var's premise can be divided in two subcases:

  - *Subcase* $x{:}T \in \Gamma$:
    Context well-formedness impiles that there is no occurence of X in T and $[X \mapsto P]T = T$. Also, $x{:}T \in \Gamma, [X \mapsto P]\Delta$ and the result follows from T-Var.
  - *Subcase* $x{:}T \in \Delta$:
    Context well-formedness impiles that there is a unique occurence of $x{:}T$ in Δ, that is, there exists Δ_1, Δ_2 such that $\Delta = \Delta_1, x{:}T, \Delta_2$, $x \notin \Delta_1$ and $x \notin \Delta_2$. As a result, $[X \mapsto P]\Delta = [X \mapsto P]\Delta_1, x{:}[X \mapsto P]T, [X \mapsto P]\Delta_2$ and $x{:}[X \mapsto P]T \in \Gamma, [X \mapsto P]\Delta$. The result follows from T-Var.

- *Case* T-Abs: $t = \lambda x{:}T_1 t_2 \quad T = T_1 \to T_2 \quad \Gamma, X{<:}Q, \Delta, x{:}T_1 \vdash t_2 : T_2$

  $[X \mapsto P](\lambda x{:}T_1 t_2) = \lambda x{:}[X \mapsto P]T_1 [X \mapsto P]t_2$ and $[X \mapsto P](T_1 \to T_2) = [X \mapsto P]T_1 \to [X \mapsto P]T_2$. We instantiate the IH with $\Delta_{IH} = (\Delta, x{:}T_1)$ to obtain
  $\Gamma, [X \mapsto P]\Delta, x{:}[X \mapsto P]T_1 \vdash [X \mapsto P]t_2 : [X \mapsto P]T_2$. The result follows from T-Abs.

- *Case* T-App: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma, X{<:}Q, \Delta \vdash t_1 : T_{11} \to T_{12} \quad \Gamma, X{<:}Q, \Delta \vdash t_2 : T_{11}$

  $[X \mapsto P](t_1 t_2) = [X \mapsto P]t_1 [X \mapsto P]t_2$, $[X \mapsto P]T_{11} \to T_{12} = [X \mapsto P]T_{11} \to [X \mapsto P]T_{12}$. The result follows directly from the IH and T-App.

- *Case* T-TAbs: $t = \lambda Y{<:}T_1.\ t_2 \quad T = \forall Y{<:}T_1.\ T_2 \quad \Gamma, X{<:}Q, \Delta, Y{<:}T_1 \vdash t_2 : T_2$

  $[X \mapsto P](\lambda Y{<:}T_1.\ t_2) = \lambda Y{<:}[X \mapsto P]T_1.\ [X \mapsto P]t_2$ and $[X \mapsto P](\forall Y{<:}T_1.\ T_2) = \forall Y{<:}[X \mapsto P]T_1.\ [X \mapsto P]T_2$. We instantiate the IH with $\Delta_{IH} = (\Delta, Y{<:}T_1)$ to obtain $\Gamma, [X \mapsto P]\Delta, Y{<:}[X \mapsto P]T_1 \vdash [X \mapsto P]t_2 : [X \mapsto P]T_2$. The result follows from T-TAbs.

- *Case* T-TAPP: $t = t_1 T_2 \quad T = [Y \mapsto T_2]T_{12}$
  $\Gamma, X{<:}Q, \Delta \vdash t_1 : (\forall Y{<:}T_{11}.\, T_{12}) \quad \Gamma, X{<:}Q, \Delta \vdash T_2 <: T_{11}$

  By the IH, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 : [X \mapsto P](\forall Y{<:}T_{11}.\, T_{12})$, that is, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 : \forall Y{<:}[X \mapsto P]T_{11}.\, [X \mapsto P]T_{12}$. Using the second part of the lemma, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]T_2 <: [X \mapsto P]T_{11}$. By T-TAPP we get, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1\, [X \mapsto P]T_2 : [Y \mapsto [X \mapsto P]T_2][X \mapsto P]T_{12}$, that is, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](t_1 T_2) : [X \mapsto P]([Y \mapsto T_2]T_{12})$, as required.

- *Case* T-SUB: $\Gamma, X{<:}Q, \Delta \vdash t : S \quad \Gamma, X{<:}Q, \Delta \vdash S <: T$

  By the IH, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]S$. Using the second part of the lemma we get, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$. The result follows from T-SUB.

- *Case* T-CLASS: $t = \mathit{new}\, C \quad T = \{\mathit{new}\, C\}$

  $[X \mapsto P]\, \mathit{new}\, C = \mathit{new}\, C$ and using T-CLASS with context $\Gamma, [X \mapsto P]\Delta$ directly leads to the desired result.

- *Case* T-MATCH: $t = t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \quad T = T_s\ \mathit{match}\{C_i \Rightarrow T_i\}\mathit{or}\ T_d$
  $\Gamma, X{<:}Q, \Delta \vdash t_s : T_s \quad \Gamma, X{<:}Q, \Delta, x_i : C_i \vdash t_i : T_i$
  $\Gamma, X{<:}Q, \Delta \vdash t_d : T_d$

  $[X \mapsto P](t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d) = [X \mapsto P]t_s\ \mathit{match}\{x_i : C_i \Rightarrow [X \mapsto P]t_i\}\mathit{or}\ [X \mapsto P]t_d$.
  $[X \mapsto P](T_s\ \mathit{match}\{C_i \Rightarrow T_i\}\mathit{or}\ T_d) = [X \mapsto P]T_s\ \mathit{match}\{C_i \Rightarrow [X \mapsto P]T_i\}\mathit{or}\ [X \mapsto P]T_d$.
  The result follows directly from the IH and T-APP.

4. By induction on a derivation of $\Gamma, x : Q, \Delta \vdash t : T$.

- *Case* T-VAR: $t = y \quad y : T \in \Gamma, x : Q, \Delta$

  By context well-formedness, the premise can be decomposed into 3 subcases:

  - *Subcase* $y : T \in \Gamma$:
    $[x \mapsto q]y = y$ and $y : T \in \Gamma, \Delta$. The result follows from T-VAR.
  - *Subcase* $y : T \in \Delta$:
    Ditto.
  - *Subcase* $y : T = x : Q$ (*i.e.* $y = x$ and $T = Q$):
    Using $\Gamma \vdash q : Q$ and Lemma B.2 we get $\Gamma, \Delta \vdash q : Q$. Since $[x \mapsto q]y = q$ and $T = Q$, $\Gamma, \Delta \vdash [x \mapsto q]y : T$, as required.

- *Case* T-ABS: $t = \lambda y : T_1 t_2 \quad T = T_1 \to T_2 \quad \Gamma, x : Q, \Delta, y : T_1 \vdash t_2 : T_2$

  We instantiate the IH with $\Delta_{IH} = (\Delta, y : T_1)$ to get $\Gamma, \Delta, y : T_1 \vdash [x \mapsto q]t_2 : T_2$. $[x \mapsto q](\lambda y : T_1 t_2) = \lambda y : T_1 [x \mapsto q]t_2$ and the result follows from T-ABS.

- *Case* T-APP: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma, x : Q, \Delta \vdash t_1 : T_{11} \to T_{12} \quad \Gamma, x : Q, \Delta \vdash t_2 : T_{11}$

  $[x \mapsto q](t_1 t_2) = ([x \mapsto q]t_1)([x \mapsto q]t_2)$ and the result follows from the IH and T-APP.

- *Case* T-TABS: $t = \lambda X{<:}U_1.\, t_2 \quad T = \forall X{<:}U_1.\, T_2 \quad \Gamma, x : Q, \Delta, X{<:}U_1 \vdash t_2 : T_2$

  We instantiate the IH with $\Delta_{IH} = (\Delta, X{<:}U_1)$ to get $\Gamma, \Delta, X{<:}U_1 \vdash [x \mapsto q]t_2 : T_2$. $[x \mapsto q](\lambda X{<:}U_1.\, t_2) = \lambda X{<:}U_1.\, [x \mapsto q]t_2$ and the result follows from T-TABS.

- *Case* T-TAPP: $t = t_1 T_2 \quad T = [X \mapsto T_2]T_{12}$
  $\Gamma, x{:}Q, \Delta \vdash t_1 : (\forall X <: U_1.\ T_{12}) \quad \Gamma, x{:}Q, \Delta \vdash T_2 <: U_1$

  By Lemma B.3, $\Gamma, \Delta \vdash T_2 <: U_1$. From the IH we get $\Gamma, \Delta \vdash [x \mapsto q]t_1 : (\forall X <: U_1.\ T_{12})$. $[x \mapsto q](t_1 T_2) = [x \mapsto q]t_1 T_2$ and the result follows from T-TAPP.

- *Case* T-SUB: $\Gamma, x{:}Q, \Delta \vdash t : S \quad \Gamma, x{:}Q, \Delta \vdash S <: T$

  By Lemma B.3, $\Gamma, \Delta \vdash S <: T$. The result follows from the IH and T-SUB.

- *Case* T-CLASS: $t = \mathit{new}\, C \quad T = \{\mathit{new}\, C\}$

  Since $[x \mapsto q]\,\mathit{new}\, C = \mathit{new}\, C$, using T-CLASS with context Γ, Δ directly leads to the desired result.

- *Case* T-MATCH: $t = t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\, \mathit{or}\ t_d \quad T = T_s\ \mathit{match}\{C_i \Rightarrow T_i\}\, \mathit{or}\ T_d$
  $\Gamma, x{:}Q, \Delta \vdash t_s : T_s \quad \Gamma, x{:}Q, \Delta, x_i : C_i \vdash t_i : T_i \quad \Gamma, x{:}Q, \Delta \vdash t_d : T_d$

  $[x \mapsto q](t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\, \mathit{or}\ t_d) = [x \mapsto q]t_s\ \mathit{match}\{x_i : C_i \Rightarrow [x \mapsto q]t_i\}\, \mathit{or}\ [x \mapsto q]t_d$, and the result follows from the IH and T-MATCH.

□

Lemma B.5 (Disjointness/subtyping exclusivity).
The type disjointness and subtyping relations are mutually exclusive.

Proof: We prove mutual exclusivity of type disjointness and subtyping by first defining $[\![\cdot]\!]_\Gamma$, a mapping from System FM types (in a given context Γ) into non-empty subsets of a newly defined set P. We then show that the subtyping relation in FM corresponds to a subset relation in P, and that the type disjointness relation in FM (disj) corresponds to set disjointness relation in P.

This set-theoretical view of subtyping and disjointness renders the proof trivial. Indeed, suppose there exist two types S and T with $\Gamma \vdash S <: T$ and $\Gamma \vdash \mathrm{disj}(S, T)$. $[\![S]\!]_\Gamma$ and $[\![T]\!]_\Gamma$ are two non-empty sets which are both intersecting and disjoint, a contradiction.

We first define P as $P = \{\Lambda, V\} \cup C$. Elements of P can be understood as equivalence classes for System FM values: Λ corresponds to all abstraction values, V corresponds to type abstraction value, and elements of C correspond to their respective constructors. The definition of $[\![\cdot]\!]_\Gamma$ makes this correspondence apparent.

We define $[\![\cdot]\!]_\Gamma$, a mapping of System FM types into subsets of P in a given context Γ:

$$\begin{aligned}
[\![\mathrm{Top}]\!]_\Gamma &= P \\
[\![X]\!]_\Gamma &= [\![T]\!]_\Gamma \quad \text{where } X <: T \in \Gamma \\
[\![T_1 \to T_2]\!]_\Gamma &= \{\Lambda\} \\
[\![\forall X <: U_1.\ T_2]\!]_\Gamma &= \{V\} \\
[\![\{\mathit{new}\, C_1\}]\!]_\Gamma &= \{C_1\} \\
[\![C_1]\!]_\Gamma &= \{c \in C \mid (c, C_1) \in \Psi\}
\end{aligned}$$

$$[\![\mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d]\!]_\Gamma = \begin{cases} [\![\mathrm{T}_n]\!]_\Gamma & \text{if } [\![\mathrm{T}_s]\!]_\Gamma \subset [\![\mathrm{S}_n]\!]_\Gamma \\ & \text{and } \forall m < n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{S}_m]\!]_\Gamma = \{\} \\ [\![\mathrm{T}_d]\!]_\Gamma & \text{if } \forall m.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{S}_m]\!]_\Gamma = \{\} \\ \mathrm{P} & \text{otherwise} \end{cases}$$

We show that the subtyping relation in FM corresponds to a subset relation in P, and that the type disjointness relation in FM (disj) corresponds to set disjointness in P. In other words, we prove the follows statements:

1. $\Gamma \vdash \mathrm{S} <: \mathrm{T}$ implies $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{T}]\!]_\Gamma$,

2. $\Gamma \vdash \mathrm{disj}(\mathrm{S}, \mathrm{T})$ implies $[\![\mathrm{S}]\!]_\Gamma \cap [\![\mathrm{T}]\!]_\Gamma = \{\}$,

Both statements are proved simultaneously by induction on derivations of $\Gamma \vdash \mathrm{disj}(\mathrm{V}, \mathrm{W})$ and $\Gamma \vdash \mathrm{S} <: \mathrm{T}$. More precisely, the induction is done on the cumulative depth of both derivation tree.

1. ($\Gamma \vdash \mathrm{S} <: \mathrm{T}$ implies $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{T}]\!]_\Gamma$)

   - *Case* S-REFL: $\mathrm{T} = \mathrm{S}$

     $[\![\mathrm{S}]\!]_\Gamma = [\![\mathrm{T}]\!]_\Gamma$ and the result is immediate.

   - *Case* S-TRANS: $\Gamma \vdash \mathrm{S} <: \mathrm{U} \quad \Gamma \vdash \mathrm{U} <: \mathrm{T}$

     By the IH, $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{U}]\!]_\Gamma$ and $[\![\mathrm{U}]\!]_\Gamma \subset [\![\mathrm{T}]\!]_\Gamma$. From subset transitivity, $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{T}]\!]_\Gamma$, as required.

   - *Case* S-TOP: $\mathrm{T} = \mathrm{Top}$

     $[\![\mathrm{Top}]\!]_\Gamma = \mathrm{P}$ coincides with the codomain of $[\![\cdot]\!]_\Gamma$. Therefore, for any type T, $[\![\mathrm{T}]\!]_\Gamma \subset [\![\mathrm{Top}]\!]_\Gamma$, as required.

   - *Case* S-SIN: $\mathrm{S} = \{\mathit{new}\,\mathrm{C}_1\} \quad \mathrm{T} = \mathrm{C}_1$

     $[\![\{\mathit{new}\,\mathrm{C}_1\}]\!]_\Gamma = \{\mathrm{C}_1\}$ and $[\![\mathrm{C}_1]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_1) \in \Psi\}$. The result follows by reflexivity of Ψ.

   - *Case* S-TVAR: $\mathrm{S} = \mathrm{X} \quad \mathrm{X} <: \mathrm{T} \in \Gamma$

     By definition $[\![\mathrm{X}]\!]_\Gamma = [\![\mathrm{T}]\!]_\Gamma$, and the result is immediate.

   - *Case* S-ARROW: $\mathrm{S} = \mathrm{S}_1 \to \mathrm{S}_2 \quad \mathrm{T} = \mathrm{T}_1 \to \mathrm{T}_2$
     $\Gamma \vdash \mathrm{T}_1 <: \mathrm{S}_1 \quad \Gamma \vdash \mathrm{S}_2 <: \mathrm{T}_2$

     $[\![\mathrm{S}_1 \to \mathrm{S}_2]\!]_\Gamma = [\![\mathrm{T}_1 \to \mathrm{T}_2]\!]_\Gamma = \{\Lambda\}$ and the result is immediate.

   - *Case* S-ALL: $\mathrm{S} = \forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{S}_2 \quad \mathrm{T} = \forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{T}_2 \quad \Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$

     $[\![\forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{S}_2]\!]_\Gamma = [\![\forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{T}_2]\!]_\Gamma = \{\mathrm{V}\}$ and the result is immediate.

   - *Case* S-PSI: $\mathrm{S} = \mathrm{C}_1 \quad \mathrm{T} = \mathrm{C}_2 \quad (\mathrm{C}_1, \mathrm{C}_2) \in \Psi$

     $[\![\mathrm{C}_1]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_1) \in \Psi\}$ and $[\![\mathrm{C}_2]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_2) \in \Psi\}$. By transitivity of Ψ, $[\![\mathrm{C}_1]\!]_\Gamma \subset [\![\mathrm{C}_2]\!]_\Gamma$, as required.

- *Case* S-MATCH1/2: $\mathrm{T}_1 = \mathrm{T}_n \quad \mathrm{T}_2 = \mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d$
  $\Gamma \vdash \mathrm{T}_s <: \mathrm{S}_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_m)$

  Using the IH we get $[\![\mathrm{T}_s]\!]_\Gamma \subset [\![\mathrm{S}_n]\!]_\Gamma$, $\forall m < n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{S}_m]\!]_\Gamma = \{\}$ and $[\![\mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d]\!]_\Gamma = [\![\mathrm{T}_n]\!]_\Gamma$. This last equality implies both $[\![\mathrm{T}_1]\!]_\Gamma \subset [\![\mathrm{T}_2]\!]_\Gamma$ and $[\![\mathrm{T}_2]\!]_\Gamma \subset [\![\mathrm{T}_1]\!]_\Gamma$, as required.

- *Case* S-MATCH3/4: $\mathrm{T}_1 = \mathrm{T}_d \quad \mathrm{T}_2 = \mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d$
  $\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{S}_n)$

  By the IH $\forall n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{S}_n]\!]_\Gamma = \{\}$. By definition, $[\![\mathrm{T}_s\ \mathit{match}\{\mathrm{S}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d]\!]_\Gamma = [\![\mathrm{T}_d]\!]_\Gamma$. This equality implies both $[\![\mathrm{T}_1]\!]_\Gamma \subset [\![\mathrm{T}_2]\!]_\Gamma$ and $[\![\mathrm{T}_2]\!]_\Gamma \subset [\![\mathrm{T}_1]\!]_\Gamma$, the desired result for S-MATCH3 and S-MATCH4 respectively.

- *Case* S-MATCH5: $\mathrm{S} = \mathrm{S}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{S}_i\}\mathit{or}\,\mathrm{S}_d \quad \mathrm{T} = \mathrm{T}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d$
  $\Gamma \vdash \mathrm{S}_s <: \mathrm{T}_s \quad \forall n.\ \Gamma \vdash \mathrm{S}_n <: \mathrm{T}_n \quad \Gamma \vdash \mathrm{S}_d <: \mathrm{T}_d$

  By case analysis on $[\![\mathrm{T}]\!]_\Gamma$:

  (a) $[\![\mathrm{T}]\!]_\Gamma = [\![\mathrm{T}_n]\!]_\Gamma$ and $[\![\mathrm{T}_s]\!]_\Gamma \subset [\![\mathrm{U}_n]\!]_\Gamma$ and $\forall m < n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{U}_m]\!]_\Gamma = \{\}$:
  The IH gives $[\![\mathrm{S}_s]\!]_\Gamma \subset [\![\mathrm{T}_s]\!]_\Gamma$. Using $\forall m < n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{U}_m]\!]_\Gamma = \{\}$, we get $\forall m < n.\ [\![\mathrm{S}_s]\!]_\Gamma \cap [\![\mathrm{U}_m]\!]_\Gamma = \{\}$. Using $[\![\mathrm{T}_s]\!]_\Gamma \subset [\![\mathrm{U}_n]\!]_\Gamma$, we get $[\![\mathrm{S}_s]\!]_\Gamma \subset [\![\mathrm{U}_n]\!]_\Gamma$ (by subset transitivity). Therefore, $[\![\mathrm{S}]\!]_\Gamma = [\![\mathrm{S}_n]\!]_\Gamma$, and the result follows from the IH.

  (b) $[\![\mathrm{T}]\!]_\Gamma = [\![\mathrm{T}_d]\!]_\Gamma$ and $\forall n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{U}_n]\!]_\Gamma = \{\}$:
  The IH gives $[\![\mathrm{S}_s]\!]_\Gamma \subset [\![\mathrm{T}_s]\!]_\Gamma$. Using $\forall n.\ [\![\mathrm{T}_s]\!]_\Gamma \cap [\![\mathrm{U}_n]\!]_\Gamma = \{\}$ we get $\forall n.\ [\![\mathrm{S}_s]\!]_\Gamma \cap [\![\mathrm{U}_n]\!]_\Gamma = \{\}$. Therefore, $[\![\mathrm{S}]\!]_\Gamma = [\![\mathrm{S}_d]\!]_\Gamma$, and the result follows from the IH.

  (c) $[\![\mathrm{T}]\!]_\Gamma = \mathrm{P}$:
  P is the codomain of $[\![\cdot]\!]_\Gamma$, therefore $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{T}]\!]_\Gamma$, as required.

2. ($\Gamma \vdash \mathrm{disj}(\mathrm{S}, \mathrm{T})$ implies $[\![\mathrm{S}]\!]_\Gamma \cap [\![\mathrm{T}]\!]_\Gamma = \{\}$)

- *Case* D-XI: $\mathrm{S} = \mathrm{C}_1 \quad \mathrm{T} = \mathrm{C}_2 \quad (\mathrm{C}_1, \mathrm{C}_2) \in \Xi$

  $[\![\mathrm{C}_1]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_1) \in \Psi\}$ and $[\![\mathrm{C}_2]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_2) \in \Psi\}$. By definition of Ξ, there is no class c such that both $(\mathrm{c}, \mathrm{C}_1) \in \Psi$ and $(\mathrm{c}, \mathrm{C}_2) \in \Psi$, and $[\![\mathrm{C}_1]\!]_\Gamma \cap [\![\mathrm{C}_2]\!]_\Gamma = \{\}$.

- *Case* D-PSI: $\mathrm{S} = \{\mathit{new}\,\mathrm{C}_1\} \quad \mathrm{T} = \mathrm{C}_2 \quad (\mathrm{C}_1, \mathrm{C}_2) \notin \Psi$

  $[\![\{\mathit{new}\,\mathrm{C}_1\}]\!]_\Gamma = \mathrm{C}_1$ and $[\![\mathrm{C}_2]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_2) \in \Psi\}$. Since $(\mathrm{C}_1, \mathrm{C}_2) \notin \Psi$, $\mathrm{C}_1 \notin [\![\mathrm{C}_2]\!]_\Gamma$ and $[\![\mathrm{C}_1]\!]_\Gamma \cap [\![\mathrm{C}_2]\!]_\Gamma = \{\}$.

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

  By the IH, $[\![\mathrm{S}]\!]_\Gamma \subset [\![\mathrm{U}]\!]_\Gamma$ and $[\![\mathrm{U}]\!]_\Gamma \cap [\![\mathrm{T}]\!]_\Gamma = \{\}$. $[\![\mathrm{S}]\!]_\Gamma \cap [\![\mathrm{T}]\!]_\Gamma = \{\}$ follows directly.

- *Case* D-ARROW: $\mathrm{S} = \mathrm{S}_1 \to \mathrm{S}_2 \quad \mathrm{T} = \mathrm{C}_1$

  $[\![\mathrm{C}_1]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_1) \in \Psi\}$, $[\![\mathrm{S}_1 \to \mathrm{S}_2]\!]_\Gamma = \{\Lambda\}$ and $[\![\mathrm{C}_1]\!]_\Gamma \cap [\![\mathrm{S}_1 \to \mathrm{S}_2]\!]_\Gamma = \{\}$, as required.

- *Case* D-ALL: $\mathrm{S} = \forall \mathrm{X} <: \mathrm{S}_1.\ \mathrm{S}_2 \quad \mathrm{T} = \mathrm{C}_1$

  $[\![\mathrm{C}_1]\!]_\Gamma = \{\mathrm{c} \in \mathrm{C} \mid (\mathrm{c}, \mathrm{C}_1) \in \Psi\}$, $[\![\forall \mathrm{X} <: \mathrm{S}_1.\ \mathrm{S}_2]\!]_\Gamma = \{\mathrm{S}\}$ and $[\![\mathrm{C}_1]\!]_\Gamma \cap [\![\forall \mathrm{X} <: \mathrm{S}_1.\ \mathrm{S}_2]\!]_\Gamma = \{\}$, as required.

□

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathrm{S} =:= \mathrm{T}}\;(\textsc{S-Match1/2})}{\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \mathrm{S} =:= \mathrm{T}}\;(\textsc{S-Match3/4})}{\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}} \qquad \dfrac{\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{U} \quad \Gamma \vdash \mathrm{U} \rightleftharpoons \mathrm{T}}{\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}}$$

Figure B.1: Definition of the auxiliary relation $\rightleftharpoons$, used to state inversion of subtyping.

Definition. $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}$ *(defined in Figure 4.4) represents evidence of the mutual subtyping between a match type* S *and a type* T *with the additional constraint that this evidence was exclusively constructed using pairwise applications of* S-MATCH1/2, S-MATCH3/4, *and* S-TRANS *in both directions.*

Lemma B.6 (Inversion of subtyping).

1. *If* $\Gamma \vdash \mathrm{S}_s$ *match*$\{\mathrm{U}_i \Rightarrow \mathrm{S}_i\}$*or* $\mathrm{S}_d \rightleftharpoons \mathrm{T}$*, then either:*
   - *(a)* $\Gamma \vdash \mathrm{S}_s <: \mathrm{U}_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{S}_s, \mathrm{U}_m)$ *and* S_n *is a match type with* $\Gamma \vdash \mathrm{S}_n \rightleftharpoons \mathrm{T}$,
   - *(b)* $\Gamma \vdash \mathrm{S}_s <: \mathrm{U}_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{S}_s, \mathrm{U}_m)$ *and* $\mathrm{S}_n = \mathrm{T}$,
   - *(c)* $\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{S}_s, \mathrm{U}_n)$ *and* S_d *is a match type with* $\Gamma \vdash \mathrm{S}_d \rightleftharpoons \mathrm{T}$,
   - *(d)* $\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{S}_s, \mathrm{U}_n)$ *and* $\mathrm{S}_d = \mathrm{T}$.
2. *If* $\Gamma \vdash \mathrm{S} <: \mathrm{X}$*, or* $\Gamma \vdash \mathrm{S} <: \mathrm{T}$ *where* T *is a match type with* $\Gamma \vdash \mathrm{T} \rightleftharpoons \mathrm{X}$*, then either*
   - *(a)* S *is a match type with* $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{Y}$*, for some* Y,
   - *(b)* S *is a type variable.*
3. *If* $\Gamma \vdash \mathrm{S} <: \mathrm{T}_1 \to \mathrm{T}_2$*, or* $\Gamma \vdash \mathrm{S} <: \mathrm{T}$ *where* T *is a match type with* $\Gamma \vdash \mathrm{T} \rightleftharpoons \mathrm{T}_1 \to \mathrm{T}_2$*, then either*
   - *(a)* S *is a match type with* $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{S}_1 \to \mathrm{S}_2$*, for some* S_1, S_2 *such that* $\Gamma \vdash \mathrm{T}_1 <: \mathrm{S}_1$ *and* $\Gamma \vdash \mathrm{S}_2 <: \mathrm{T}_2$,
   - *(b)* S *is a match type with* $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{X}$*, for some* X,
   - *(c)* S *is a type variable,*
   - *(d)* S *has the form* $\mathrm{S}_1 \to \mathrm{S}_2$ *with* $\Gamma \vdash \mathrm{T}_1 <: \mathrm{S}_1$ *and* $\Gamma \vdash \mathrm{S}_2 <: \mathrm{T}_2$.
4. *If* $\Gamma \vdash \mathrm{S} <: \forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{T}_2$*, or* $\Gamma \vdash \mathrm{S} <: \mathrm{T}$ *where* T *is a match type with* $\Gamma \vdash \mathrm{T} \rightleftharpoons \forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{T}_2$*, then either*
   - *(a)* S *is a match type with* $\Gamma \vdash \mathrm{S} \rightleftharpoons \forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{S}_2$*, for some* S_2 *such that* $\Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$,
   - *(b)* S *is a match type with* $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{X}$*, for some* X,
   - *(c)* S *is a type variable,*
   - *(d)* S *has the form* $\forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{S}_2$ *and* $\Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$.

Proof:

1. By induction on a derivation on $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}$.

- *Case* S-MATCH1/2: $S = S_s\ \mathit{match}\{U_i \Rightarrow S_i\}\mathit{or}\, S_d \quad T = S_n$
  $\Gamma \vdash S_s <: U_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(S_s, U_m)$

  $S_n = T$ and the result follows directly from the premises.

- *Case* S-MATCH3/4: $S = S_s\ \mathit{match}\{U_i \Rightarrow S_i\}\mathit{or}\, S_d \quad T = S_d \quad \forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$

  $S_d = T$ and the result follows directly from the premises.

- *Case* S-TRANS: $\Gamma \vdash S \rightleftharpoons U \quad \Gamma \vdash U \rightleftharpoons T$

  By definition of the $\rightleftharpoons$ relation, $\Gamma \vdash U \rightleftharpoons T$ implies that U is a match type. Using the IH on the left premise we get 4 subcases:

  (a) $\Gamma \vdash S_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(S_s, U_m)$ and S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons U$:
  Using S-TRANS we get $\Gamma \vdash S_n \rightleftharpoons T$, as requiered.

  (b) $\Gamma \vdash S_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(S_s, U_m)$ and $S_n = U$:
  The result is immediate.

  (c) $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$ and S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons U$:
  Using S-TRANS we get $\Gamma \vdash S_d \rightleftharpoons T$, as requiered.

  (d) $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$ and $S_d = U$:
  The result is immediate.

2. By induction on a derivation of $\Gamma \vdash S <: T$.

- *Case* S-REFL: $S = T$

  If T is a type variable then so is S and the result is immediate. If T is a match type with $\Gamma \vdash T \rightleftharpoons X$, S is a match type and $\Gamma \vdash S \rightleftharpoons Y$, as required.

- *Case* S-TRANS: $\Gamma \vdash S <: U \quad \Gamma \vdash U <: T$

  If T is a type variable we use the IH on the right premise to get that U is either a match type with $\Gamma \vdash U \rightleftharpoons X$ or a type variable. In either case, the result follows from using the IH on the left premise.

  If T is a match type with $\Gamma \vdash T \rightleftharpoons X$ we can also use the the IH on the right premise to get to the same result.

- *Case* S-TVAR: $S = Y \quad Y <: X \in \Gamma$

  S is a type variable and the result is immediate.

- *Case* S-MATCH1: $S = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\mathit{or}\, T_d \quad T = T_n$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  If $T = X$, we use S-MATCH2 to obtain $\Gamma \vdash S \rightleftharpoons X$, as required. If T is a match type with $\Gamma \vdash T \rightleftharpoons X$, we use S-MATCH2 to get $\Gamma \vdash S \rightleftharpoons T$, and S-TRANS to obtain $\Gamma \vdash S \rightleftharpoons X$, as required.

- *Case* S-MATCH2: $S = T_n \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\mathit{or}\, T_d$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  In this case the first premise of the statement ($\Gamma \vdash S <: X$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons X$, we get 4 subcases:

(a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons X$:
$S = T_n$ is a match type and $\Gamma \vdash S \rightleftharpoons X$, as required.

(b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_n = X$:
$S = T_n = X$ is a type variable and the result is immediate.

(c) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons X$:
This case cannot occur since $\Gamma \vdash \text{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

(d) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_d = X$:
This case cannot occur since $\Gamma \vdash \text{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

- *Case* S-MATCH3: $S = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\ \mathit{or}\ T_d \quad T = T_d$
  $\forall n.\ \Gamma \vdash \text{disj}(T_s, U_n)$

  If $T = X$, we use S-MATCH4 to obtain $\Gamma \vdash S \rightleftharpoons X$, as required. If T is a match type with $\Gamma \vdash T \rightleftharpoons X$, we use S-MATCH4 to get $\Gamma \vdash S \rightleftharpoons T$, and S-TRANS to obtain $\Gamma \vdash S \rightleftharpoons X$, as required.

- *Case* S-MATCH4: $S = T_d \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\ \mathit{or}\ T_d$
  $\forall n.\ \Gamma \vdash \text{disj}(T_s, U_n)$

  In this case the first premise of the statement ($\Gamma \vdash S <: X$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons X$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons X$:
  This case cannot occur since $\Gamma \vdash \text{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

  (b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_n = X$:
  This case cannot occur since $\Gamma \vdash \text{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

  (c) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons X$:
  S is a match type and $\Gamma \vdash S \rightleftharpoons X$, as required.

  (d) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_d = X$:
  $S = T_d = X$ is a type variable and the result is immediate.

- *Case* S-MATCH5: $S = S_s\ \mathit{match}\{U_i \Rightarrow S_i\}\ \mathit{or}\ S_d \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\}\ \mathit{or}\ T_d$
  $\Gamma \vdash S_s <: T_s \quad \forall m.\ \Gamma \vdash S_m <: T_m \quad \Gamma \vdash S_d <: T_d$

  In this case the first premise of the statement ($\Gamma \vdash S <: X$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons X$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons X$:
  Using D-SUB on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we get $\forall m < n.\ \Gamma \vdash \text{disj}(S_s, U_m)$. From S-TRANS we obtain $\Gamma \vdash S_s <: U_n$. Using S-MATCH1/2 we get $\Gamma \vdash S \rightleftharpoons S_n$. Since $\Gamma \vdash S_n <: T_n$ and $\Gamma \vdash T_n \rightleftharpoons X$, we use the IH to get that either S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons Y$, or S_n is a type variable. If S_n is a type variable, we already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required. If $\Gamma \vdash S_n \rightleftharpoons Y$, then using S-TRANS with $\Gamma \vdash S \rightleftharpoons S_n$ gives $\Gamma \vdash S \rightleftharpoons Y$, as required.

(b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_n = X$:
Using D-SUB on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we get $\forall m < n.\ \Gamma \vdash \text{disj}(S_s, U_m)$. From S-TRANS we obtain $\Gamma \vdash S_s <: U_n$. Using S-MATCH1/2 we get $\Gamma \vdash S \rightleftharpoons S_n$. Since $\Gamma \vdash S_n <: X$, we use the IH to get that either S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons Y$, or S_n is a type variable. If S_n is a type variable, we already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required. If $\Gamma \vdash S_n \rightleftharpoons Y$, then using S-TRANS with $\Gamma \vdash S \rightleftharpoons S_n$ gives $\Gamma \vdash S \rightleftharpoons Y$, as required.

(c) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons X$:
Using D-SUB on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we obtain $\forall m.\ \Gamma \vdash \text{disj}(S_s, U_m)$. Using S-MATCH3/4 we obtain $\Gamma \vdash S \rightleftharpoons S_d$. Since $\Gamma \vdash S_d <: T_d$ and $\Gamma \vdash T_d \rightleftharpoons X$, we use the IH to get that either S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons Y$, or S_d is a type variable. If S_d is a type variable, we already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required. If $\Gamma \vdash S_d \rightleftharpoons Y$, then using S-TRANS with $\Gamma \vdash S \rightleftharpoons S_d$ gives $\Gamma \vdash S \rightleftharpoons Y$, as required.

(d) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_d = X$:
Using D-SUB on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we obtain $\forall m.\ \Gamma \vdash \text{disj}(S_s, U_m)$. Using S-MATCH3/4 we obtain $\Gamma \vdash S <: S_d$. Since $\Gamma \vdash S_d <: T_d$ and $\Gamma \vdash T_d \rightleftharpoons X$, we use the IH to get that either S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons Y$, or S_d is a type variable. If S_d is a type variable, we already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required. If $\Gamma \vdash S_d \rightleftharpoons Y$, then using S-TRANS with $\Gamma \vdash S \rightleftharpoons S_d$ gives $\Gamma \vdash S \rightleftharpoons Y$, as required.

- *Case* S-TOP, S-SIN, S-ARROW, S-ALL, S-PSI:
In those cases, T is neither a type variable nor a match type and the result is immediate.

3. By induction on a derivation of $\Gamma \vdash S <: T$.

- *Case* S-REFL: $T = S$
If T is a match type with $\Gamma \vdash T \rightleftharpoons T_1 \to T_2$ and the result follows directly from S-REFL. If $T = T_1 \to T_2$, the result also follows directly from S-REFL.

- *Case* S-TRANS: $\Gamma \vdash S <: U \quad \Gamma \vdash U <: T$
Using the IH on the right premise we get 4 subcases:

(a) U is a match type with $\Gamma \vdash U \rightleftharpoons U_1 \to U_2$, for some U_1, U_2 such that $\Gamma \vdash T_1 <: U_1$ and $\Gamma \vdash U_2 <: T_2$:
The result follows directly from using the IH on the left premise ($\Gamma \vdash T_1 <: S_1$ is obtained using S-TRANS with $\Gamma \vdash T_1 <: U_1$ and $\Gamma \vdash U_1 <: S_1$, $\Gamma \vdash S_2 <: T_2$ is obtained analogously).

(b) U is a match type with $\Gamma \vdash U \rightleftharpoons X$, for some X:
Using the 2nd part of the lemma on the left premise leads to the desired result.

(c) U is a type variable:
The result follows from using the 2nd part of the lemma on the left premise.

(d) U has the form $U_1 \rightarrow U_2$ with $\Gamma \vdash T_1 <: U_1$ and $\Gamma \vdash U_2 <: T_2$:
The result follows directly from using the IH on the left premise ($\Gamma \vdash T_1 <: S_1$ is obtained using S-TRANS with $\Gamma \vdash T_1 <: U_1$ and $\Gamma \vdash U_1 <: S_1$, $\Gamma \vdash S_2 <: T_2$ is obtained analogously).

- *Case* S-MATCH1: $S = T_s\ \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\ T_d \quad T = T_n$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  If $T = T_1 \rightarrow T_2$, we use S-MATCH2 to obtain $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, as required. If T is a match type with $\Gamma \vdash T \rightleftharpoons T_1 \rightarrow T_2$, we use S-MATCH2 to get $\Gamma \vdash S \rightleftharpoons T$, and S-TRANS to obtain $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, and the result follows from S-REFL.

- *Case* S-MATCH2: $S = T_n \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\ T_d$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  In this case the first premise of the statement ($\Gamma \vdash S <: T_1 \rightarrow T_2$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons T_1 \rightarrow T_2$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons T_1 \rightarrow T_2$:
  $S = T_n$ is a match type and $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, and the result follows from S-REFL.

  (b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and $T_n = T_1 \rightarrow T_2$:
  $S = T_1 \rightarrow T_2$, $S_1 = T_1$, $S_2 = T_2$, and the result follows from S-REFL.

  (c) $\forall m.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons T_1 \rightarrow T_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

  (d) $\forall m.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and $T_d = T_1 \rightarrow T_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

- *Case* S-MATCH3: $S = T_s\ \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\ T_d \quad T = T_d$
  $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$

  If $T = T_1 \rightarrow T_2$, we use S-MATCH2 to obtain $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, and the result follows from S-REFL. If T is a match type with $\Gamma \vdash T \rightleftharpoons T_1 \rightarrow T_2$, we use S-MATCH2 to get $\Gamma \vdash S \rightleftharpoons T$, and S-TRANS to obtain $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, and the result follows from S-REFL.

- *Case* S-MATCH4: $S = T_d \quad T = T_s\ \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\ T_d$
  $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$

  In this case the first premise of the statement ($\Gamma \vdash S <: T_1 \rightarrow T_2$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons T_1 \rightarrow T_2$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons T_1 \rightarrow T_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

(b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash disj(T_s, U_m)$ and $T_n = T_1 \rightarrow T_2$:
This case cannot occur since $\Gamma \vdash disj(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

(c) $\forall m.\ \Gamma \vdash disj(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons T_1 \rightarrow T_2$:
$S = T_d$ is a match type and $\Gamma \vdash S \rightleftharpoons T_1 \rightarrow T_2$, and the result follows from S-Refl.

(d) $\forall m.\ \Gamma \vdash disj(T_s, U_m)$ and $T_d = T_1 \rightarrow T_2$:
$S = T_1 \rightarrow T_2$, $S_1 = T_1$, $S_2 = T_2$, and the result follows from S-Refl.

- *Case* S-Match5: $S = S_s\ match\{U_i \Rightarrow S_i\}\ or\ S_d \quad T = T_s\ match\{U_i \Rightarrow T_i\}\ or\ T_d$
$\Gamma \vdash S_s <: T_s \quad \forall n.\ \Gamma \vdash S_n <: T_n \quad \Gamma \vdash S_d <: T_d$

  In this case the first premise of the statement ($\Gamma \vdash S <: T_1 \rightarrow T_2$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons T_1 \rightarrow T_2$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash disj(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons T_1 \rightarrow T_2$:
  Using D-Sub on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m < n.\ \Gamma \vdash disj(T_s, U_m)$ we obtain $\forall m < n.\ \Gamma \vdash disj(S_s, U_m)$. From S-Trans, $\Gamma \vdash S_s <: U_n$. Using S-Match1/2 we get $\Gamma \vdash S \rightleftharpoons S_n$. Since $\Gamma \vdash S_n <: T_n$ and $\Gamma \vdash T_n \rightleftharpoons T_1 \rightarrow T_2$, the IH gives 4 subsubcases:

  i. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons S_1 \rightarrow S_2$ for some S_1, S_2 such that $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
  The result follows directly from S-Trans.

  ii. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons X$:
  The result follows directly from S-Trans.

  iii. S_n is a type variable:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

  iv. S_n has the form $S_1 \rightarrow S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

  (b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash disj(T_s, U_m)$ and $T_n = T_1 \rightarrow T_2$:
  Using D-Sub on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m < n.\ \Gamma \vdash disj(T_s, U_m)$ we obtain $\forall m < n.\ \Gamma \vdash disj(S_s, U_m)$. From S-Trans, $\Gamma \vdash S_s <: U_n$. Using S-Match1/2 we get $\Gamma \vdash S \rightleftharpoons S_n$. Since $\Gamma \vdash S_n <: T_n$, the IH gives 4 subsubcases:

  i. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons S_1 \rightarrow S_2$ for some S_1, S_2 such that $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
  The result follows directly from S-Trans.

  ii. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons X$:
  The result follows directly from S-Trans.

  iii. S_n is a type variable:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

  iv. S_n has the form $S_1 \rightarrow S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

(c) $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons T_1 \to T_2$:
Using D-Sub on $\forall n.\ \Gamma \vdash S_n <: T_n$ and $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$ we obtain $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$. From S-Trans we get $\Gamma \vdash S_s <: C_d$. Using S-Match3/4 we obtain $\Gamma \vdash S \rightleftharpoons S_d$. Since $\Gamma \vdash S_d <: T_d$ and $\Gamma \vdash T_d \rightleftharpoons T_1 \to T_2$, the IH gives 4 subsubcases:

i. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons S_1 \to S_2$ for some S_1, S_2 such that $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
The result follows directly from S-Trans.

ii. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons X$:
The result follows directly from S-Trans.

iii. S_d is a type variable:
We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

iv. S_d has the form $S_1 \to S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

(d) $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$ and $T_d = T_1 \to T_2$:
Using D-Sub on $\forall n.\ \Gamma \vdash S_n <: T_n$ and $\forall n.\ \Gamma \vdash \mathrm{disj}(T_s, U_n)$ we obtain $\forall n.\ \Gamma \vdash \mathrm{disj}(S_s, U_n)$. From S-Trans we get $\Gamma \vdash S_s <: C_d$. Using S-Match3/4 we obtain $\Gamma \vdash S \rightleftharpoons S_d$. Since $\Gamma \vdash S_d <: T_d$, the IH gives 4 subsubcases:

i. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons S_1 \to S_2$ for some S_1, S_2 such that $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
The result follows directly from S-Trans.

ii. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons X$:
The result follows directly from S-Trans.

iii. S_d is a type variable:
We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

iv. S_d has the form $S_1 \to S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

- *Case* S-TVar: $S = Y \quad Y <: T \in \Gamma$

  S is a type variable and the result is immediate.

- *Case* S-Arrow: $S = S_1 \to S_2 \quad T = T_1 \to T_2$
  $\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2$

  S has the form $S_1 \to S_2$, with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$, as required.

- *Case* S-Top, S-Sin, S-All, S-Psi:

  In those cases, T is neither a type variable nor a function type and the result is immediate.

4. By induction on a derivation of $\Gamma \vdash S <: T$.

- *Case* S-Refl: $T = S$

  If T is a match type with $\Gamma \vdash T \rightleftharpoons \forall X <: U_1.\ T_2$ the result follows directly from S-Refl. If $T = \forall X <: U_1.\ T_2$, $S_2 = T_1$, and the result also follows from S-Refl.

- *Case* S-Trans: $\Gamma \vdash S <: U \quad \Gamma \vdash U <: T$

  Using the IH on the right premise we get 4 subcases:

  (a) U is a match type with $\Gamma \vdash U \rightleftharpoons U_1 \to U_2$ for some U_2 such that $\Gamma, X <: U_1 \vdash U_2 <: T_2$:
  The result follows directly from using the IH on the left premise ($\Gamma, X <: U_1 \vdash S_2 <: T_2$ is obtained using S-Trans with $\Gamma, X <: U_1 \vdash S_2 <: U_2$ and $\Gamma, X <: U_1 \vdash U_2 <: T_2$).

  (b) U is a match type with $\Gamma \vdash U \rightleftharpoons X$:
  Using the 2nd part of the lemma on the left premise leads to the desired result.

  (c) U is a type variable:
  The result follows from using the 2nd part of the lemma on the left premise.

  (d) U has the form $\forall X <: U_1 .\, U_2$ with $\Gamma, X <: U_1 \vdash U_2 <: T_2$:
  The result follows directly from using the IH on the left premise ($\Gamma, X <: U_1 \vdash S_2 <: T_2$ is obtained using S-Trans with $\Gamma, X <: U_1 \vdash S_2 <: U_2$ and $\Gamma, X <: U_1 \vdash U_2 <: T_2$).

- *Case* S-Match1: $S = T_s \; \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\, T_d \quad T = T_n$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  If $T = \forall X <: U_1 .\, T_2$, we use S-Match2 to obtain $\Gamma \vdash S \rightleftharpoons \forall X <: U_1 .\, T_2$, as required, and the result follows from S-Refl. If T is a match type with $\Gamma \vdash T \rightleftharpoons \forall X <: U_1 .\, T_2$, we use S-Match2 to get $\Gamma \vdash S \rightleftharpoons T$, and S-Trans to obtain $\Gamma \vdash S \rightleftharpoons \forall X <: U_1 .\, T_2$, and the result follows from S-Refl.

- *Case* S-Match2: $S = T_n \quad T = T_s \; \mathit{match}\{U_i \Rightarrow T_i\} \mathit{or}\, T_d$
  $\Gamma \vdash T_s <: U_n \quad \forall m < n.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$

  In this case the first premise of the statement ($\Gamma \vdash S <: (\forall X <: U_1 .\, T_2)$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash T \rightleftharpoons \forall X <: U_1 .\, T_2$, we get 4 subcases:

  (a) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and T_n is a match type with $\Gamma \vdash T_n \rightleftharpoons \forall X <: U_1 .\, T_2$:
  $S = T_n$ is a match type and $\Gamma \vdash S \rightleftharpoons \forall X <: U_1 .\, T_2$, and the result follows from S-Refl.

  (b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and $T_n = \forall X <: U_1 .\, T_2$:
  $S = \forall X <: U_1 .\, T_2$, $S_2 = T_2$, and the result follows from S-Refl.

  (c) $\forall m.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons \forall X <: U_1 .\, T_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

  (d) $\forall m.\, \Gamma \vdash \mathrm{disj}(T_s, U_m)$ and $T_d = \forall X <: U_1 .\, T_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(T_s, U_n)$ would contradict $\Gamma \vdash T_s <: U_n$, by Lemma 4.5.

- *Case* S-MATCH3: $\quad \mathrm{S} = \mathrm{T}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d \quad \mathrm{T} = \mathrm{T}_d$
  $\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_n)$

  If $\mathrm{T} = \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, we use S-MATCH2 to obtain $\Gamma \vdash \mathrm{S} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, and the result follows from S-REFL. If T is a match type with $\Gamma \vdash \mathrm{T} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, we use S-MATCH2 to get $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{T}$, and S-TRANS to obtain $\Gamma \vdash \mathrm{S} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, and the result follows from S-REFL.

- *Case* S-MATCH4: $\quad \mathrm{S} = \mathrm{T}_d \quad \mathrm{T} = \mathrm{T}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d$
  $\forall n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_n)$

  In this case the first premise of the statement ($\Gamma \vdash \mathrm{S} <: (\forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2)$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash \mathrm{T} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, we get 4 subcases:

  (a) $\Gamma \vdash \mathrm{T}_s <: \mathrm{U}_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ and T_n is a match type with $\Gamma \vdash \mathrm{T}_n \rightleftharpoons \forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{T}_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_n)$ would contradict $\Gamma \vdash \mathrm{T}_s <: \mathrm{U}_n$, by Lemma 4.5.

  (b) $\Gamma \vdash \mathrm{T}_s <: \mathrm{U}_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ and $\mathrm{T}_n = \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$:
  This case cannot occur since $\Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_n)$ would contradict $\Gamma \vdash \mathrm{T}_s <: \mathrm{U}_n$, by Lemma 4.5.

  (c) $\forall m.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ and T_d is a match type with $\Gamma \vdash \mathrm{T}_d \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$:
  $\mathrm{S} = \mathrm{T}_d$ is a match type and $\Gamma \vdash \mathrm{S} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, and the result follows from S-REFL.

  (d) $\forall m.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ and $\mathrm{T}_d = \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$:
  $\mathrm{S} = \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, $\mathrm{S}_2 = \mathrm{T}_2$, and the result follows from S-REFL.

- *Case* S-MATCH5: $\quad \mathrm{S} = \mathrm{S}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{S}_i\}\mathit{or}\,\mathrm{S}_d \quad \mathrm{T} = \mathrm{T}_s\ \mathit{match}\{\mathrm{U}_i \Rightarrow \mathrm{T}_i\}\mathit{or}\,\mathrm{T}_d$
  $\Gamma \vdash \mathrm{S}_s <: \mathrm{T}_s \quad \forall n.\ \Gamma \vdash \mathrm{S}_n <: \mathrm{T}_n \quad \Gamma \vdash \mathrm{S}_d <: \mathrm{T}_d$

  In this case the first premise of the statement ($\Gamma \vdash \mathrm{S} <: (\forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2)$) does not apply since T is a match type. Using the 1st part of the lemma on $\Gamma \vdash \mathrm{T} \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, we get 4 subcases:

  (a) $\Gamma \vdash \mathrm{T}_s <: \mathrm{U}_n$, $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ and T_n is a match type with $\Gamma \vdash \mathrm{T}_n \rightleftharpoons \forall \mathrm{X} <: \mathrm{U}_1.\ \mathrm{T}_2$:
  Using D-SUB on $\forall m.\ \Gamma \vdash \mathrm{S}_m <: \mathrm{T}_m$ and $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{T}_s, \mathrm{U}_m)$ we get $\forall m < n.\ \Gamma \vdash \mathrm{disj}(\mathrm{S}_s, \mathrm{U}_m)$. From S-TRANS we get $\Gamma \vdash \mathrm{S}_s <: \mathrm{U}_n$. Using S-MATCH1/2 we get $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{S}_n$. Since $\Gamma \vdash \mathrm{S}_n <: \mathrm{T}_n$ and $\Gamma \vdash \mathrm{T}_n \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{T}_2$, the IH gives 4 sub-subcases:

     i. S_n is a match type with $\Gamma \vdash \mathrm{S}_n \rightleftharpoons \forall \mathrm{X}{<:}\mathrm{U}_1.\ \mathrm{S}_2$ for some S_2 such that $\Gamma, \mathrm{X} <: \mathrm{U}_1 \vdash \mathrm{S}_2 <: \mathrm{T}_2$:
     The result follows directly from S-TRANS.

     ii. S_n is a match type with $\Gamma \vdash \mathrm{S}_n \rightleftharpoons \mathrm{X}$:
     The result follows directly from S-TRANS.

     iii. S_n is a type variable:
     We already proved $\Gamma \vdash \mathrm{S} \rightleftharpoons \mathrm{S}_n$, as required.

- iv. S_n has the form $\forall X <: U_1.\ S_2$ with $\Gamma, X <: U_1 \vdash S_2 <: T_2$:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

(b) $\Gamma \vdash T_s <: U_n$, $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_n = \forall X <: U_1.\ T_2$:
Using D-Sub on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m < n.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we get $\forall m < n.\ \Gamma \vdash \text{disj}(S_s, U_m)$. From S-Trans we get $\Gamma \vdash S_s <: U_n$. Using S-Match1/2 we get $\Gamma \vdash S \rightleftharpoons S_n$. Since $\Gamma \vdash S_n <: T_n$, the IH, the IH gives 4 subsubcases:

- i. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons \forall X <: U_1.\ S_2$ for some S_2 such that $\Gamma, X <: U_1 \vdash S_2 <: T_2$:
  The result follows directly from S-Trans.
- ii. S_n is a match type with $\Gamma \vdash S_n \rightleftharpoons X$:
  The result follows directly from S-Trans.
- iii. S_n is a type variable:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.
- iv. S_n has the form $\forall X <: U_1.\ S_2$ with $\Gamma, X <: U_1 \vdash S_2 <: T_2$:
  We already proved $\Gamma \vdash S \rightleftharpoons S_n$, as required.

(c) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and T_d is a match type with $\Gamma \vdash T_d \rightleftharpoons \forall X <: U_1.\ T_2$:
Using D-Sub on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we obtain $\forall m.\ \Gamma \vdash \text{disj}(S_s, U_m)$. From S-Trans we get $\Gamma \vdash S_s <: C_d$. Using S-Match3/4 we obtain $\Gamma \vdash S \rightleftharpoons S_d$. Since $\Gamma \vdash S_d <: T_d$ and $\Gamma \vdash T_d \rightleftharpoons \forall X <: U_1.\ T_2$, the IH gives 4 subsubcases:

- i. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons \forall X <: U_1.\ S_2$ for some S_2 such that $\Gamma, X <: U_1 \vdash S_2 <: T_2$::
  The result follows directly from S-Trans.
- ii. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons X$:
  The result follows directly from S-Trans.
- iii. S_d is a type variable:
  We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.
- iv. S_d has the form $\forall X <: U_1.\ S_2$ with $\Gamma, X <: U_1 \vdash S_2 <: T_2$:
  We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

(d) $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ and $T_d = \forall X <: U_1.\ T_2$:
Using D-Sub on $\forall m.\ \Gamma \vdash S_m <: T_m$ and $\forall m.\ \Gamma \vdash \text{disj}(T_s, U_m)$ we obtain $\forall m.\ \Gamma \vdash \text{disj}(S_s, U_m)$. From S-Trans we get $\Gamma \vdash S_s <: C_d$. Using S-Match3/4 we obtain $\Gamma \vdash S \rightleftharpoons S_d$. Since $\Gamma \vdash S_d <: T_d$, the IH gives 4 subsubcases:

- i. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons \forall X <: U_1.\ S_2$ for some S_2 such that $\Gamma, X <: U_1 \vdash S_2 <: T_2$::
  The result follows directly from S-Trans.
- ii. S_d is a match type with $\Gamma \vdash S_d \rightleftharpoons X$:
  The result follows directly from S-Trans.
- iii. S_d is a type variable:
  We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

iv. S_d has the form $\forall X{<:}U_1.\, S_2$ with $\Gamma, X{<:}U_1 \vdash S_2 <: T_2$:
We already proved $\Gamma \vdash S \rightleftharpoons S_d$, as required.

- *Case* S-TVAR: $S = Y \quad Y{<:}T \in \Gamma$

  S is a type variable and the result is immediate.

- *Case* S-ALL: $S = \forall X{<:}U_1.\, S_2 \quad T = \forall X{<:}U_1.\, T_2$
  $\Gamma, X{<:}U_1 \vdash S_2 <: T_2$

  S has the form $\forall X{<:}U_1.\, S_2$, with $\Gamma, X{<:}U_1 \vdash S_2 <: T_2$, as required.

- *Case* S-TOP, S-SIN, S-ARROW, S-PSI:

  In those cases, T is neither a type variable nor a universal type and the result is immediate.

□

Lemma B.7 (Canonical forms).

1. *If* $\Gamma \vdash t : T$, *where either* T *is a type variable, or* T *is a match type with* $\Gamma \vdash T \rightleftharpoons X$, *then* t *is not a closed value.*

2. *If* v *is a closed value with* $\Gamma \vdash v : T$ *where either* $T = T_1 \to T_2$, *or* T *is a match type and* $\Gamma \vdash T \rightleftharpoons T_1 \to T_2$, *then* v *has the form* $\lambda x{:}S_1.\, t_2$.

3. *If* v *is a closed value with* $\Gamma \vdash v : T$ *where either* $T = \forall X{<:}U_1.\, T_2$, *or* T *is a match type and* $\Gamma \vdash T \rightleftharpoons \forall X{<:}U_1.\, T_2$, *then* v *has the form* $\lambda X{<:}U_1.\, t_2$.

Proof:

1. By induction on a derivation of $\Gamma \vdash t : T$

   - *Case* T-VAR, T-TAPP, T-APP, T-MATCH:

     In those cases, t is not a value and the result is immediate.

   - *Case* T-ABS, T-TABS, T-CLASS:

     In those cases, T is neither a match type nor a type variable and the result is immediate.

   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

     By Lemma 4.6, either S is a match type with $\Gamma \vdash S \rightleftharpoons Y$, or S is a type variable. In both cases, we use the IH to show that t is not a closed value, as required.

2. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-VAR, T-TAPP, T-APP, T-MATCH:

     In those cases, t is not a value and the result is immediate.

   - *Case* T-ABS: $t = \lambda x{:}T_1 t_2 \quad T = T_1 \to T_2 \quad \Gamma, x{:}T_1 \vdash t_2 : T_2$

     $t = \lambda x{:}T_1 t_2$ and the result is immediate.

- *Case* T-TABS, T-CLASS:

  T is neither a function type nor a match type and the result is immediate.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  Using Lemma 4.6 we get 4 subcases:

  (a) S is a match type with $\Gamma \vdash S \rightleftharpoons S_1 \to S_2$:
  The result follows from the IH.

  (b) S is a match type with $\Gamma \vdash S \rightleftharpoons X$:
  This case cannot occure since the 1th part of the lemma would lead to a contradiction on the fact that v is a closed value.

  (c) S is a type variable:
  Similarly, this case cannot occure since the 1st part of the lemma would lead to a contradiction.

  (d) S has the form $S_1 \to S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$:
  The result follows from the IH.

3. By induction on a derivation of $\Gamma \vdash t : T$.

- *Case* T-VAR, T-TAPP, T-APP, T-MATCH:

  In those cases, t is not a value and the result is immediate.

- *Case* T-ABS, T-CLASS:

  T is neither a universal type nor a match type and the result is immediate.

- *Case* T-TABS: $t = \lambda Y <: T_1.\, t_2 \quad T = \forall Y <: T_1.\, T_2 \quad \Gamma, Y <: T_1 \vdash t_2 : T_2$

  $t = \lambda Y <: T_1.\, t_2$ and the result is immediate.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  Using Lemma 4.6 we get 4 subcases:

  (a) S is a match type with $\Gamma \vdash S \rightleftharpoons \forall X <: U_1.\, S_2$, The result follows from the IH.

  (b) S is a match type with $\Gamma \vdash S \rightleftharpoons X$, This case cannot occure since the 1th part of the lemma would lead to a contradiction on the fact that v is a closed value.

  (c) S is a type variable, Similarly, this case cannot occure since the 1st part of the lemma would lead to a contradiction.

  (d) S has the form $\forall X <: U_1.\, S_2$ and $\Gamma, X <: U_1 \vdash S_2 <: T_2$. The result follows from the IH.

□

Lemma B.8 (Inversion of typing).

1. *If* $\Gamma \vdash \lambda x : S_1.\, s_2 : T$ *and* $\Gamma \vdash T <: U_1 \to U_2$, *then* $\Gamma \vdash U_1 <: S_1$ *and there is some* S_2 *such that* $\Gamma, x : S_1 \vdash s_2 : S_2$ *and* $\Gamma \vdash S_2 <: U_2$.

2. *If* $\Gamma \vdash \lambda X <: S_1.\, s_2 : T$ *and* $\Gamma \vdash T <: (\forall X <: U_1.\, U_2)$, *then* $U_1 = S_1$ *and there is some* S_2 *such that* $\Gamma, X <: S_1 \vdash s_2 : S_2$ *and* $\Gamma, X <: S_1 \vdash S_2 <: U_2$.

Proof:

1. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-ABS: $t = \lambda x{:}S_1 s_2 \quad T = S_1 \to T_2 \quad \Gamma x{:}S_1 \vdash s_2 : T_2$

     Given $\Gamma \vdash S_1 \to T_2 <: U_1 \to U_2$, we use Lemma 4.6 to obtain $\Gamma \vdash U_1 <: S_1$ and $\Gamma \vdash T_2 <: U_2$. We pick S_2 to be T_2 to obtain the desired result.

   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

     Using S-TRANS with $\Gamma \vdash S <: T$ and $\Gamma \vdash T <: U_1 \to U_2$ we get $\Gamma \vdash S <: U_1 \to U_2$. The results follows directly from the IH.

   - *Case* T-VAR, T-APP, T-TABS, T-TAPP, T-CLASS, T-MATCH:

     In those cases, t is not of the form $\lambda x{:}S_1.\, s_2$ and the result is immediate.

2. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-TABS: $t = \lambda X{<:}S_1.\, s_2 \quad T = \forall X{<:}S_1.\, T_2 \quad \Gamma, X{<:}S_1 \vdash s_2 : T_2$

     Given $\Gamma \vdash (\forall X{<:}S_1.\, T_2) <: (\forall X{<:}U_1.\, U_2)$, we use Lemma 4.6 to obtain $U_1 = S_1$ and $\Gamma, X{<:}S_1 \vdash T_2 <: U_2$. We pick S_2 to be T_2 to obtain the desired result.

   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

     Using S-TRANS with $\Gamma \vdash S <: T$ and $\Gamma \vdash T <: (\forall X{<:}U_1.\, U_2)$ we get $\Gamma \vdash S <: (\forall X{<:}U_1.\, U_2)$. The results follows directly from the IH.

   - *Case* T-VAR, T-ABS, T-APP, T-TAPP, T-CLASS, T-MATCH:

     In those cases, t is not of the form $\lambda X{<:}S_1.\, s_2 : T$ and the result is immediate.

□

Lemma B.9 (Minimum types).

1. *If* $\Gamma \vdash$ *new* $C : T$ *then* $\Gamma \vdash \{$*new* $C\} <: T$.
2. *If* $\Gamma \vdash \lambda x{:}T_1.\, t_2 : T$ *then there is some* T_2 *such that* $\Gamma \vdash T_1 \to T_2 <: T$.
3. *If* $\Gamma \vdash \lambda X{<:}U_1.\, t_2 : T$ *then there is some* T_2 *such that* $\Gamma \vdash \forall X{<:}U_1.\, T_2 <: T$.

Proof:

1. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-VAR, T-ABS, T-APP, T-TABS, T-TAPP, T-MATCH:

     In those cases, t is not a constructor call and the result is immediate.

   - *Case* T-CLASS: $t =$ *new* $C \quad T = \{$*new* $C\}$

     The result follows directly from S-REFL.

   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

     By the IH, $\Gamma \vdash \{$*new* $C\} <: S$. The result follows from S-TRANS.

2. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-VAR, T-APP, T-TABS, T-TAPP, T-CLASS, T-MATCH:
     In those cases, t is not an abstraction and the result is immediate
   - *Case* T-ABS: $T = T_1 \to S_2 \quad \Gamma, x : T_1 \vdash t_2 : S_2$
     $T_2 = S_2$ and the result is immediate using S-REFL.
   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$
     Using the IH, there is some T_2 such that $\Gamma \vdash T_1 \to T_2 <: S$. The result follows from S-TRANS.

3. By induction on a derivation of $\Gamma \vdash t : T$.

   - *Case* T-VAR, T-ABS, T-APP, T-TAPP, T-CLASS, T-MATCH:
     In those cases, t is not a type abstraction and the result is immediate
   - *Case* T-TABS: $T = \forall X <: U_1 .\, S_2 \quad \Gamma, X <: U_1 \vdash t_2 : S_2$
     $T_2 = S_2$ and the result is immediate using S-REFL.
   - *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$
     Using the IH, there is some T_2 such that $\Gamma \vdash \forall X <: U_1 .\, T_2 <: S$. The result follows from S-TRANS.

□

Theorem B.10 (Preservation).
If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : T$.

Proof: By induction on a derivation of $\Gamma \vdash t : T$.

- *Case* T-VAR, T-ABS, T-TABS, T-CLASS:

  These cases cannot arise since we assume $t \longrightarrow t'$ but there are no evaluation rules for variables, abstractions, type abstractions and class instantiations.

- *Case* T-APP: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}$

  By definition of the evaluation relation, there are 3 subcases:

  - *Subcase* E-APP1: $t_1 \longrightarrow t_1' \quad t' = t_1' t_2$
    By the IH and the 1st premise we get $\Gamma \vdash t_1' : T_{11} \to T_{12}$. We use T-APP with that result and the 2nd premise to obtain $\Gamma \vdash t_1' t_2 : T_{12}$, as required.
  - *Subcase* E-APP2: $t_2 \longrightarrow t_2' \quad t_1 = v_1 \quad t' = v_1 t_2'$
    Analogously, the IH gives $\Gamma \vdash t_2' : T_{12}$ and the result follows from T-APP.
  - *Subcase* E-APPABS: $t_1 = \lambda x : U_{11} .\, u_{12} \quad t' = [x \mapsto t_2] u_{12}$
    By Lemma 4.8 (with $S_1 = U_{11}$, $s_2 = u_{12}$, $U_1 = T_{11}$ and $U_2 = T_{12}$), there is some S_2 such that $\Gamma, x : U_{11} \vdash u_{12} : S_2$, $\Gamma \vdash S_2 <: T_{12}$ and $\Gamma \vdash T_{11} <: U_{11}$. Using T-SUB with $\Gamma \vdash t_2 : T_{11}$ and $\Gamma \vdash T_{11} <: U_{11}$ we obtain $\Gamma \vdash t_2 : U_{11}$. By Lemma B.4 we get $\Gamma \vdash [x \mapsto t_2] u_{12} : S_2$. Using T-SUB we obtain $\Gamma \vdash [x \mapsto t_2] u_{12} : T_{12}$, as required.

- *Case* T-TAPP: $\mathrm{t} = \mathrm{t}_1 \mathrm{T}_2 \quad \mathrm{T} = [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{T}_{12}$
  $\Gamma \vdash \mathrm{t}_1 : (\forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{T}_{12}) \quad \Gamma \vdash \mathrm{T}_2 <: \mathrm{U}_1$

  The proof for case T-TAPP is analogous to the one for T-APP. By definition of the evaluation relation, there are 2 subcases:

  - *Subcase* E-TAPP: $\mathrm{t}_1 \longrightarrow \mathrm{t}_1' \quad \mathrm{t}' = \mathrm{t}_1' \mathrm{T}_2$
    By the IH and the 1st premise we get $\Gamma \vdash \mathrm{t}_1' : (\forall \mathrm{X} <: \mathrm{U}_1 .\, \mathrm{T}_{12})$. We use T-TAPP with that result and the 2nd premise to obtain $\Gamma \vdash \mathrm{t}_1' \mathrm{T}_2 : [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{T}_{12}$, as required.
  - *Subcase* E-TAPPTABS: $\mathrm{t}_1 = \lambda \mathrm{X} <: \mathrm{U}_{11} .\, \mathrm{u}_{12} \quad \mathrm{t}' = [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{u}_{12}$
    By Lemma 4.8 (with $\mathrm{S}_1 = \mathrm{U}_{11}$, $\mathrm{s}_2 = \mathrm{u}_{12}$ and $\mathrm{U}_2 = \mathrm{T}_{12}$), there is some S_2 such that $\Gamma, \mathrm{X} <: \mathrm{U}_{11} \vdash \mathrm{u}_{12} : \mathrm{S}_2$, $\mathrm{U}_1 = \mathrm{U}_{11}$, and $\Gamma, \mathrm{X} <: \mathrm{U}_{11} \vdash \mathrm{S}_2 <: \mathrm{T}_{12}$. Since $\Gamma \vdash \mathrm{T}_2 <: \mathrm{U}_{11}$, we use Lemma B.4 twice to get $\Gamma \vdash [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{u}_{12} : [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{S}_2$ and $\Gamma \vdash [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{S}_2 <: [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{T}_{12}$. By T-SUB $\Gamma \vdash [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{u}_{12} : [\mathrm{X} \mapsto \mathrm{T}_2]\mathrm{T}_{12}$, as required.

- *Case* T-SUB: $\Gamma \vdash \mathrm{t} : \mathrm{S} \quad \Gamma \vdash \mathrm{S} <: \mathrm{T}$

  By the IH, $\Gamma \vdash \mathrm{t}' : \mathrm{S}$. The result follows directly from T-SUB.

- *Case* T-MATCH: $\mathrm{t} = \mathrm{t}_s \; \mathit{match}\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\} \mathit{or}\; \mathrm{t}_d \quad \mathrm{T} = \mathrm{T}_s \; \mathit{match}\{\mathrm{C}_i \Rightarrow \mathrm{T}_i\} \mathit{or}\; \mathrm{T}_d$
  $\Gamma \vdash \mathrm{t}_s : \mathrm{T}_s \quad \Gamma, \mathrm{x}_i : \mathrm{C}_i \vdash \mathrm{t}_i : \mathrm{T}_i \quad \Gamma \vdash \mathrm{t}_d : \mathrm{T}_d$

  By definition of the evaluation relation, there are 5 subcases:

  - *Subcase* E-MATCH1: $\mathrm{t}_s \longrightarrow \mathrm{t}_s' \quad \mathrm{t}' = \mathrm{t}_s' \; \mathit{match}\{\mathrm{x}_i : \mathrm{C}_i \Rightarrow \mathrm{t}_i\} \mathit{or}\; \mathrm{t}_d$
    By the IH $\Gamma \vdash \mathrm{t}_s' : \mathrm{T}_s$. The result follows directly from T-MATCH.
  - *Subcase* E-MATCH2: $\mathrm{t}_s = \mathit{new}\, \mathrm{C} \quad (\mathrm{C}, \mathrm{C}_n) \in \Psi$
    $\forall m < n.\, (\mathrm{C}, \mathrm{C}_m) \notin \Psi \quad \mathrm{t}' = [\mathrm{x}_n \mapsto \mathit{new}\, \mathrm{C}]\mathrm{t}_n$
    By S-PSI, S-SIN and S-TRANS, $\Gamma \vdash \{\mathit{new}\, \mathrm{C}\} <: \mathrm{C}_n$. From D-PSI, we obtain $\forall m < n.\, \Gamma \vdash \mathrm{disj}(\{\mathit{new}\, \mathrm{C}\}, \mathrm{C}_m)$. By Lemma 4.9, $\Gamma \vdash \{\mathit{new}\, \mathrm{C}\} <: \mathrm{T}_s$. Let T_1 be $\{\mathit{new}\, \mathrm{C}\}\; \mathit{match}\{\mathrm{C}_i \Rightarrow \mathrm{T}_i\} \mathit{or}\; \mathrm{T}_d$. From S-MATCH1, $\Gamma \vdash \mathrm{T}_n <: \mathrm{T}_1$. Using S-MATCH5, $\Gamma \vdash \mathrm{T}_1 <: \mathrm{T}$. By S-TRANS, $\Gamma \vdash \mathrm{T}_n <: \mathrm{T}$.
    Using T-CLASS, T-SUB, S-SIN and S-PSI (with $(\mathrm{C}, \mathrm{C}_n) \in \Psi$) we get $\Gamma \vdash \mathit{new}\; \mathrm{C} : \mathrm{C}_n$. From the case premises, we have $\Gamma, \mathrm{x}_n : \mathrm{C}_n \vdash \mathrm{t}_n : \mathrm{T}_n$. By Lemma B.4, we get $\Gamma \vdash [\mathrm{x}_n \mapsto \mathit{new}\, \mathrm{C}]\mathrm{t}_n : \mathrm{T}_n$. Finally, using T-SUB we get , $\Gamma \vdash [\mathrm{x}_n \mapsto \mathit{new}\, \mathrm{C}]\mathrm{t}_n : \mathrm{T}$, as required.
  - *Subcase* E-MATCH3: $\mathrm{t}_s = \mathit{new}\, \mathrm{C} \quad \forall n.\, (\mathrm{C}, \mathrm{C}_n) \notin \Psi \quad \mathrm{t}' = \mathrm{t}_d$
    The proof for subcase E-MATCH3 is analogous to the one for E-MATCH2. From D-PSI, $\forall n.\, \Gamma \vdash \mathrm{disj}(\{\mathit{new}\, \mathrm{C}\}, \mathrm{C}_n)$. By Lemma 4.9 we get $\Gamma \vdash \{\mathit{new}\, \mathrm{C}\} <: \mathrm{T}_s$. Let T_1 be $\{\mathit{new}\, \mathrm{C}\}\; \mathit{match}\{\mathrm{C}_i \Rightarrow \mathrm{T}_i\} \mathit{or}\; \mathrm{T}_d$. From S-MATCH2, $\Gamma \vdash \mathrm{T}_d <: \mathrm{T}_1$. Using S-MATCH5, $\Gamma \vdash \mathrm{T}_1 <: \mathrm{T}$. By S-TRANS $\Gamma \vdash \mathrm{T}_d <: \mathrm{T}$. Using T-SUB, $\Gamma \vdash \mathrm{t}_d : \mathrm{T}$, as required.
  - *Subcase* E-MATCH4: $\mathrm{t}_s = \lambda \mathrm{x} : \mathrm{U}.\, \mathrm{u} \quad \mathrm{t}' = \mathrm{t}_d$
    By Lemma 4.9, there exists a V such that $\Gamma \vdash \mathrm{U} \rightarrow \mathrm{V} <: \mathrm{T}_s$. Using D-ARROW, $\forall n.\, \Gamma \vdash \mathrm{disj}(\mathrm{U} \rightarrow \mathrm{V}, \mathrm{C}_n)$. Let T_1 be $\mathrm{U} \rightarrow \mathrm{V}\; \mathit{match}\{\mathrm{C}_i \Rightarrow \mathrm{T}_i\} \mathit{or}\; \mathrm{T}_d$. From S-MATCH2, $\Gamma \vdash \mathrm{T}_d <: \mathrm{T}_1$. Using S-MATCH5, $\Gamma \vdash \mathrm{T}_1 <: \mathrm{T}$. By S-TRANS $\Gamma \vdash \mathrm{T}_d <: \mathrm{T}$. Using T-SUB, $\Gamma \vdash \mathrm{t}_d : \mathrm{T}$, as required.

- *Subcase* E-MATCH5: $t_s = \forall X<:U.\ u \quad t' = t_d$

  The proof for subcase E-MATCH5 is analogous to the one for E-MATCH4. By Lemma 4.9, there exists a V such that $\Gamma \vdash (\forall X<:U.\ V) <: T_s$. Using D-ARROW, we get $\forall n.\ \Gamma \vdash \mathrm{disj}(\forall X<:U.\ V, C_n)$. Let T_1 be $(\forall X<:U.\ V)$ $\mathit{match}\{C_i \Rightarrow T_i\}\mathit{or}\, T_d$. From S-MATCH2, $\Gamma \vdash T_d <: T_1$. Using S-MATCH5, $\Gamma \vdash T_1 <: T$. By S-TRANS $\Gamma \vdash T_d <: T$. Using T-SUB, $\Gamma \vdash t_d : T$, as required.

□

Theorem B.11 (Progress).
If t *is a closed, well-typed term, then either* t *is a value or there is some* t′ *such that* $t \longrightarrow t'$.

Proof: By induction on a derivation of $\Gamma \vdash t : T$.

- *Case* T-VAR: $t = x \quad x : T \in \Gamma$

  This case cannot occure because t is closed.

- *Case* T-ABS, T-TABS, T-CLASS:

  In those cases, t is a value and the result is immediate.

- *Case* T-APP: $t = t_1 t_2 \quad T = T_{12} \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}$

  By the IH, either t_1 is a value or t_1 can take a step (there is some t_1' such that $t_1 \longrightarrow t_1'$). If t_1 can take a step, then E-APP1 applies to t. If t_1 is a value, we use Lemma 4.7 to get that t_1 has the form $\lambda x : S_1.\ t_2$. Therefore, E-APPABS applies to t, as required.

- *Case* T-TAPP: $t = t_1 T_2 \quad T = [X \mapsto T_2]T_{12}$
  $\Gamma \vdash t_1 : (\forall X<:U_1.\ T_{12}) \quad \Gamma \vdash T_2 <: U_1$

  The proof for case T-TAPP is analogous to the one for T-APP.

  By the IH, either t_1 is a value or t_1 can take a step. If t_1 can take a step, then E-TAPP applies to t. If t_1 is a value, we use Lemma 4.7 to get that t_1 has the form $\lambda X<:T_1.\ t_2$. Therefore, E-TAPPTABS applies to t, as required.

- *Case* T-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

  The result follows directly from the IH.

- *Case* T-MATCH: $t = t_s\ \mathit{match}\{x_i : C_i \Rightarrow t_i\}\mathit{or}\ t_d \quad T = T_s\ \mathit{match}\{C_i \Rightarrow T_i\}\mathit{or}\ T_d$
  $\Gamma \vdash t_s : T_s \quad \Gamma, x_i : C_i \vdash t_i : T_i \quad \Gamma \vdash t_d : T_d$

  By the IH, either t_s is a value or t_s can take a step. If t_s can take a step, then E-MATCH1 applies to t, as required. If t_s is a value, t_s can take 3 different forms:

  - *Subcase* t_s is of the form *new* C:
    If $\forall m.\ (C, C_m) \notin \Psi$, then E-MATCH3 applies to t. Otherwise, let C_k be the first class such that $(C, C_k) \in \Psi$. By construction we know that $\forall m < k.\ (C, C_k) \notin \Psi$. Therefore E-MATCH2 applies to t, as required.

- *Subcase* t_s is of the form $\lambda\mathrm{x}{:}\mathrm{T}_1\mathrm{t}_2$:
  E-MATCH4 applies to t and the result is immediate.
- *Subcase* t_s is of the form $\forall\mathrm{X}{<:}\mathrm{U}_1.\,\mathrm{T}_2$:
  E-MATCH5 applies to t and the result is immediate.

□

Bibliography

Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1991). Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2). https://doi.org/10.1145/103135.103138.

Amin, N. and Rompf, T. (2017). Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'17, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/3009837.3009866.

Aspinall, D. (1994). Subtyping with singleton types. In *International Workshop on Computer Science Logic*, Berlin, Heidelberg. Springer, Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0022243.

Augustsson, L. (1998). Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'98, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/291251.289451.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R., and Weirich, S. (2008). Engineering formal metatheory. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'08, New York, NY, USA. ACM. https://doi.org/10.1145/1328438.1328443.

Barham, P. and Isard, M. (2019). Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS'19, New York, NY, USA. ACM. https://doi.org/10.1145/3317550.3321441.

Bertot, Y. and Castéran, P. (2004). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer. https://doi.org/10.1007/978-3-662-07964-5.

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'14, Berlin, Heidelberg. Springer-Verlag. https://doi.org/10.1007/978-3-662-44202-9_11.

Blanvillain, O., Brachthäuser, J., Kjaer, M., and Odersky, M. (2021a). Type-level programming with match types. page 70. https://infoscience.epfl.ch/record/290019.

Blanvillain, O., Brachthäuser, J., Kjaer, M., and Odersky, M. (2021b). Type-level programming with match types artifact. https://doi.org/10.5281/zenodo.5568850.

Blanvillain, O., Brachthäuser, J. I., Kjaer, M., and Odersky, M. (2022). Type-level programming with match types. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'22. ACM. https://doi.org/10.1145/3498698.

Blanvillain, O., Iliofotou, M., Chang, A., Kanterov, G., and Frameless contributors (2016–2022). Frameless. https://github.com/typelevel/frameless.

Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. (1994). An extension of system f with subtyping. *Information and computation*, 109(1-2). https://doi.org/10.1006/inco.1994.1013.

Chakravarty, M. M. T., Keller, G., and Jones, S. P. (2005). Associated type synonyms. *SIGPLAN Not.*, 40(9). https://doi.org/10.1145/1090189.1086397.

Chen, T. (2017). Typesafe abstractions for tensor operations (short paper). In *Proceedings of the ACM SIGPLAN International Symposium on Scala*, SCALA'17, New York, NY, USA. ACM. https://doi.org/10.1145/3136000.3136001.

Chlipala, A. (2010). Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/1809028.1806612.

Courant, J. (2003). Strong normalization with singleton types. *Electronic Notes in Theoretical Computer Science*, 70(1). https://doi.org/10.1016/S1571-0661(04)80490-0.

Eisenberg, R. A. (2016). *Dependent types in Haskell: Theory and practice.* PhD dissertation, University of Pennsylvania. https://arxiv.org/abs/1610.07978.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., and Weirich, S. (2014). Closed type families with overlapping equations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, POPL'14, New York, NY, USA. AMC. https://doi.org/10.1007/BFb0022243.

Eisenberg, R. A. and Weirich, S. (2012). Dependently typed programming with singletons. In *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, Haskell'12, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/2430532.2364522.

Emir, B., Odersky, M., and Williams, J. (2007). Matching objects with patterns. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'07, Berlin, Heidelberg. Springer-Verlag. https://dl.acm.org/doi/10.1007/978-3-540-73589-2_14.

Giarrusso, P. G., Stefanesco, L., Timany, A., Birkedal, L., and Krebbers, R. (2020). Scala step-by-step: Soundness for dot with step-indexed logical relations in iris. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'20, New York, NY, USA. ACM. https://doi.org/10.1145/3408996.

Harper, R. and Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'95, New York, NY, USA. ACM. https://doi.org/10.1145/199448.199475.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825). https://doi.org/10.1038/s41586-020-2649-2.

Huang, A., Stites, S., and Scholak, T. (2017–2022). HaskTorch. https://github.com/hasktorch/hasktorch.

Hutchins, D. S. (2010). Pure subtype systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'10, New York, NY, USA. ACM. https://doi.org/10.1145/1706299.1706334.

Jones, M. P. (2000). Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP'00, London, UK, UK. Springer-Verlag. https://dl.acm.org/doi/10.1007/3-540-46425-5_15.

Kazerounian, M., Guria, S. N., Vazou, N., Foster, J. S., and Van Horn, D. (2019). Type-level computations for ruby libraries. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'19, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/3314221.3314630.

Kiselyov, O., Jones, S. P., and Shan, C.-c. (2010). Fun with type functions. In *Reflections on the Work of CAR Hoare*. Springer. https://dl.acm.org/doi/10.1007/978-1-84882-912-1_14.

Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell'04, New York, NY, USA. ACM. https://doi.org/10.1145/1017472.1017488.

Kvrikava, F., Miller, H., and Vitek, J. (2019). Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'19, New York, NY, USA. ACM. https://doi.org/10.1145/3360589.

Leijen, D. and Meijer, E. (1999). Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages*, DSL'99, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/331960.331977.

Leontiev, G., Burmako, E., Zaugg, J., Moors, A., Phillips, P., Port, O., and Sabin, M. (2014). *SIP-23 - Literal-Based Singleton Types*. Scala Center. https://docs.scala-lang.org/sips/42.type.html.

Liu, F. (2016). A generic algorithm for checking exhaustivity of pattern matching (short paper). In *Proceedings of the ACM SIGPLAN Symposium on Scala*, SCALA'16, New York, NY, USA. ACM. https://doi.org/10.1145/2998392.2998401.

McBride, C. (2002). Faking it: Simulating dependent types in Haskell. *Journal of functional programming*, 12(4-5). https://dl.acm.org/doi/10.1017/S0956796802004355.

Meijer, E., Beckman, B., and Bierman, G. (2006). Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/1142473.1142552.

Nieto, A., Zhao, Y., Lhoták, O., Chang, A., and Pu, J. (2020). Scala with Explicit Nulls. In Hirschfeld, R. and Pape, T., editors, *34th European Conference on Object-Oriented Programming (ECOOP'20)*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://10.4230/LIPIcs.ECOOP.2020.25.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD dissertation, Chalmers University of Technology. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

Odersky, M., Altherr, P., Cremet, V., Dubochet, G., Emir, B., Haller, P., Micheloud, S., Mihaylov, N., Moors, A., Rytz, L., Schinz, M., Stenman, E., and Zenger, M. (2006–2022). *Scala Language Specification*. EPFL and Lightbend, Inc. https://scala-lang.org/files/archive/spec/2.13/.

Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., and Stucki, S. (2018). Simplicitly: Foundations and applications of implicit function types. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'18. ACM. https://doi.org/10.1145/3158130.

Odersky, M. and Dotty contributors (2013–2022). *Scala 3 Language Reference*. LAMP/EPFL. https://docs.scala-lang.org/scala3/reference/overview.html.

Oliveira, B. C., Moors, A., and Odersky, M. (2010). Type classes as objects and implicits. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'10, New York, NY, USA. ACM. https://doi/10.1145/1932682.1869489.

Pierce, B. C. (2002). *Types and programming languages*. MIT press. https://dl.acm.org/doi/book/10.5555/509043.

Pilquist, M. and Scodec contributors (2013–2022). Scodec. https://github.com/scodec/scodec.

Rapoport, M., Kabir, I., He, P., and Lhoták, O. (2017). A simple soundness proof for dependent object types. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'17, New York, NY, USA. ACM. https://doi.org/10.1145/3133870.

Rush, A. (2019). Tensor considered harmful. https://nlp.seas.harvard.edu/NamedTensor.

Sabin, M. and Shapeless contributors (2011–2022). Shapeless. https://github.com/milessabin/shapeless.

Schinz, M. (2005). *Compiling Scala for the Java virtual machine*. PhD dissertation, EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-3302.

Schmid, G. S., Blanvillain, O., Hamza, J., and Kuncak, V. (2020). Coming to terms with your choices: An existential take on dependent types. *CoRR*, abs/2011.07653. https://arxiv.org/abs/2011.07653.

Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. (2008). Type checking with open type functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'08, New York, NY, USA. ACM. https://doi.org/10.1145/1411204.1411215.

Shabalin, D. (2020). *Just-in-time performance without warm-up*. PhD dissertation, EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-9768.

Sjoberg, V. (2015). *A Dependently Typed Language with Nontermination*. PhD thesis, University of Pennsylvania. https://repository.upenn.edu/dissertations/AAI3709556.

Stone, C. A. and Harper, R. (2000). Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

The TypeScript development team (2019–2022). *The TypeScript Handbook*. Microsoft Corporation. https://www.typescriptlang.org/docs/handbook/intro.html.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'89, New York, NY, USA. ACM. https://doi.org/10.1145/75277.75283.

Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. (2017). A specification for dependent types in haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'17, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/3110275.

Weirich, S., Vytiniotis, D., Peyton Jones, S., and Zdancewic, S. (2011). Generative type abstraction and type-level computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'11, New York, NY, USA. ACM. https://doi.org/10.1145/1926385.1926411.

Xi, H. and Pfenning, F. (1998). Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'98, New York, NY, USA. ACM. https://dl.acm.org/doi/10.1145/277650.277732.

Yang, Y. and Oliveira, B. C. d. S. (2017). Unifying typing and subtyping. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'17, New York, NY, USA. ACM. https://doi.org/10.1145/3133871.

Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhães, J. P. (2012). Giving Haskell a promotion. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI'12. ACM. https://dl.acm.org/doi/10.1145/2103786.2103795.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11). https://dl.acm.org/doi/10.1145/2934664.

Zwanenburg, J. (1999). Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer. https://dl.acm.org/doi/10.1007/3-540-48959-2_27.

Olivier Blanvillain

Avenue Floréal 5
1006 Lausanne, Switzerland
+417 86 85 77 85
olivier.blanvillain@gmail.com

Education

2022: PhD in Computer Science at EPFL (expected)

2015: EPFL Master's degree in Computer Science

2012: EPFL Bachelor's degree in Computer Science

2008: French Baccalaureate

Experience

2016-2022: Doctoral Assistant at EPFL in the Programming Methods Laboratory under the supervision of Prof. Martin Odersky.

- Co-designed, implemented and formalized match types, a new Scala feature for type-level computations.
- Co-designed and prototyped a dependently typed extension of Scala based on singleton types.
- Supervised several master student projects and worked as a teaching assistant for undergraduate courses (CS-210 and CS-206).

2015-2016: Software Engineer at MFG Labs (14 months).

- Lead a team of 6 engineers working on an AdTech project.
- Worked with Scala, Play, PostgreSQL, Spark, Elasticsearch, AWS.

2013: Software Engineer Intern at CERN (6 months).

Publications

O. Blanvillain, J. Brachthäuser, M. Kjaer, M. Odersky. Type-Level Programming with Match Types. In *Symposium on Principles of Programming Languages*, 2022 (POPL'22).

M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, S. Stucki. Simplicitly: Foundations and Applications of Implicit Function Types. In *Symposium on Principles of Programming Languages*, 2018 (POPL'18).

O. Blanvillain, N. Kasioumis, V. Banos. BlogForever Crawler: Techniques and Algorithms to Harvest Modern Weblogs. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics*, 2014 (WIMS'14).

Personal

Born on July 9, 1990 in Geneva, dual citizenship French-Swiss.

Languages: French (mother tongue), English (C1/C2), Spanish (B2).

Hobbies: Music (piano, drums and lots of listening), board games, cycling.