École Polytechnique Fédérale de Lausanne

Formalizing GADT constraint reasoning in Scala

by Mario Bucev

Master Thesis

Approved by the Examining Committee:

Prof. Martin Odersky
Thesis Advisor

Fengyun Liu
External Expert

Aleksander Slawomir Boruch-Gruszecki
Thesis Supervisor

EPFL IC IINFCOM LAMP1
INR 319 (Bâtiment INR)
Station 14
CH-1015 Lausanne

June 25, 2021

Acknowledgments

I thank Prof. Martin Odersky for granting me the opportunity to work on this Master thesis. I am furthermore truly grateful for his course *Foundations of Software* which has sparked in me the curiosity and willingness to learn more about types and programming languages!

I would like to sincerely thank my supervisor, Aleksander Slawomir Boruch-Gruszecki. We have had weekly meetings with fruitful discussions during which he had responded to my questions and my uncertainty. He furthermore pointed out pernicious, time bomb mistakes that would have surely revealed themselves the day before the handout. Not only that, but he provided solutions for correcting them as well. Without his support, I would have very likely given up a dozen times. Sincerely thank you!

I would also like to thank Benjamin C. Pierce for his astonishing book *Types and Programming Languages*, explaining in simple terms concepts that are at first look intimidating. I am thankful to François Pottier and Didier Rémy for their excellent *The Essence of ML Type Inference* chapter in Pierce's *Advanced Types and Programming Languages* work – allowing me to view the type inference problem from a constraint point of view. I would like to add that, without Aleksander's input, I would have not stumbled across this gem during this thesis – I am delighted he introduced me to this chapter!

I would like to express my gratitude towards Dr. Michel Schinz for his course *Advanced Compiler Construction* and Prof. Viktor Kunčak for his course *Formal Verification*. These courses made me appreciate the amazing field of Computer Science even more.

I would like to thank Prof. Mathias Payer for providing a ready-to-use LaTeX thesis template. I cannot imagine how much time I would have spent trying to make one of my own before despairing.

Finally, I am extremely grateful to my family and friends for their constant and selfless support.

Lausanne, June 25, 2021 Mario Bucev

Abstract

Generalized algebraic data types (GADTs) are a powerful tool allowing to express invariants leveraging the type system.

Scala 3 considerably improves the support of GADTs with respect to its predecessor Scala 2. A unique feature of Scala 3, compared to languages integrating GADTs, is the ability to define variant GADTs.

While Scala 3 GADTs support is satisfactory, some use-cases could benefit from extending it further.

In this work, we lay out the necessary tools to help us understand and reason about the GADT inference problem. We propose an algorithm that incrementally refines the accumulated knowledge about the type variables and prove its soundness. We also show some examples where the proposed algorithm is able to infer interesting properties that the current Scala 3 compiler misses.

Contents

List of Algorithms

Chapter 1

Introduction

1.1 Motivations

Generalized algebraic data types (GADTs) are simple but extremely powerful constructs allowing to encode invariants within the type system of the host language [1, 14, 17].

Consider the following snippet representing a simply typed λ-calculus [1], adapted from [7]:

```scala
enum Expr[T] {
  case Var[T](v: T) extends Expr[T]
  case IntLit(v: Int) extends Expr[Int]
  case BoolLit(v: Boolean)
    extends Expr[Boolean]
  case Pair[A, B](fst: Expr[A],
    snd: Expr[B]) extends Expr[(A, B)]
  case If(cond: Expr[Boolean],
    tru: Expr[T],
    fls: Expr[T]) extends Expr[T]
  case Abs[A, B](fn: Expr[A] => Expr[B])
    extends Expr[A => B]
  case App[A, B](
    fn: Expr[A => B],
    arg: Expr[A]) extends Expr[B]
}
```

```scala
import Expr._

def eval[T](e: Expr[T]): T = e match {
  case Var(v) => v
  case IntLit(v) => v
  case BoolLit(v) => v
  case Pair(fst, snd) =>
    (eval(fst), eval(snd))
  case If(cond, tru, fls) =>
    if (eval(cond)) eval(tru)
    else eval(fls)
  case f: Abs[a, b] =>
    (arg: a) => eval(f.fn(Var(arg)))
  case app: App[a, b] =>
    eval(app.fn)(eval(app.arg))
}
```

Listing 1 – GADT definition in Scala 3

The `Expr` data type is a GADT: all cases but `Var` extend `Expr` with type arguments different from T. This definition allows to rule out ill-typed terms *at compile-time*, leveraging the meta-language type system. For example, the following term is well-typed:

```scala
val term1 = If(BoolLit(false), IntLit(42),
  App(Abs(e => IntLit(24)), Var(())))
```

While the one below results in a compile-time error:

```scala
// Does not compile:
// Found: IntLit; Required: Expr[Boolean]
val term2 = If(BoolLit(false), BoolLit(true),
  App(Abs(e => IntLit(24)), Var(())))
```

---

[1] Except when mentioned otherwise, all code snippets are compiled and ran under Scala 3.0.0.

The expressiveness of GADTs is truly unleashed when they are scrutinized through pattern matching. Let us have a look at the `eval` function defined above. For the `IntLit` case, we get to know that e is a subtype of `IntLit` – allowing us to derive that T is in fact equal to `Int`. Perhaps more interestingly is the `Pair` case where `fst` and `snd` are equal to `Expr`$[A]$ and `Expr`$[B]$ respectively, *for some* A and B. We therefore see that scrutinizing GADTs may not only refine the parameterized type, but also unveil existentially quantified types.

Scala supports variant GADTs as well, thus allowing an even broader range of (correct) programs. We can turn the previously defined `Expr` covariant, and appropriately annotate the cases as follows:

```scala
enum Expr[+T] {
  case Var[+T](v: T) extends Expr[T]
  case IntLit(v: Int) extends Expr[Int]
  case BoolLit(v: Boolean) extends Expr[Boolean]
  case Pair[+A, +B](fst: Expr[A],
                    snd: Expr[B]) extends Expr[(A, B)]
  case If(cond: Expr[Boolean],
          tru: Expr[T],
          fls: Expr[T]) extends Expr[T]
  case Abs[-A, +B](fn: Expr[A] => Expr[B]) extends Expr[A => B]
  case App[A, +B](fn: Expr[A => B], arg: Expr[A]) extends Expr[B]
}
```

Listing 2 – Covariant GADT in Scala 3

The `eval` function remains identical.

In the snippet below, if `Expr` had not been made covariant, the compiler would have rejected the program.

```scala
trait Bird {
  def makeSound: String
}
class Duck extends Bird {
  def makeSound: String = "quack"
}
class Sparrow extends Bird {
  def makeSound: String = "chirp"
}

val got = eval(If(
  BoolLit(false),
  Var(new Duck),
  Var(new Sparrow)
)).makeSound // "chirp"
```

Listing 3 – Leveraging the covariance of `Expr`

Though GADTs support in Scala 3 is well established, there is still some room for improvements. For instance, the following snippet does not compile even though it should:

```scala
trait Inv[X]
trait Inv2[X, Y]

// "S" for "Scrutinee"
trait S[F[_]]
// "P" for "Pattern"
trait P[Y, F[Z] >: Inv2[Z, Y]] extends S[F] {
  def y: Y
}

def patmat[Y, F[Z] <: Inv2[Z, Y]](s: S[F]): Y = s match {
  // Error: found pY;  Required Y
  // Should compile
  case p: P[pY, pF] => p.y
}
```

Listing 4 – False negative example

Intuitively, pF and F are equal; therefore `Inv2`$[Z, pY]$ must be a subtype of `Inv2`$[Z, Y]$ for all Z – leading to the conclusion that pY and Y are equal.

In this work, we establish the settings allowing to reason about the GADT inference problem. We then propose an algorithm with a departure from the current GADT inference algorithm of the Scala compiler. The core idea of the proposed algorithm is to maintain a *knowledge structure* representing the incremental accumulation of information with respect to type variables. When new information arrives, that information is *assimilated* into the structure which may result in unveiling further facts about the type variables.

1.2 Running examples

We deem it fruitful to have three examples that we refer to throughout this report. These examples serve as a motivation for the presented concepts as well as illustrating the outputs of the proposed algorithm.

The first snippet is simple: its purpose is to swiftly connect the discussions with the examples.

```scala
trait S[X, +Y]
trait P extends S[Int, String]

def patmat[X, Y](s: S[X, Y]): X = s match {
  case p: P => 42
}

val got = patmat(new P{}) // 42
```

Listing 5 – Introductory example

The second example shows a false positive snippet. The 3.0.0 compiler version performs an incorrect inference on the type variables and accepts the snippet even though it should not – leading to a `ClassCastException` [2]. In chapter 4, we show the reason why it is not correct to infer that $X =$ `String`, and present in chapter 5 the inferred results of the proposed algorithm.

---

[2] The issue is tracked in `https://github.com/lampepfl/dotty/issues/11545` and its corresponding fix in `https://github.com/lampepfl/dotty/pull/12087`.

```scala
trait Inv[X]
trait S[X]
trait P[X] extends S[Inv[X] & Inv[String]]

def patmat[X, Y](s: S[Inv[X] & Y]): X = s match {
  // Should not compile
  // Inferred: X = String
  case p: P[pX] => "Hello"
}
// ClassCastException: String cannot be cast to Integer
val got: Int = patmat[Int, Inv[String]](new P{})
```

Listing 6 – Incorrect inference leading to a crash

The third example (based on listing 4) is an intricate snippet that is rejected by the compiler. As we will see in chapter 4, it is possible to prove that pX and pY are indeed equal to Y. We then showcase in chapter 5 a run of the proposed algorithm that infers the equality between pX, pY and Y.

```scala
trait Inv[X]
trait Inv2[X, Y]

trait S[X, F[_]]
trait P[X, Y, F[Z] >: Inv2[Z, Y] & Inv[Y]] extends S[Inv[X], F] {
  def x: X
  def y: Y
}

def patmat[X, Y, F[Z] <: Inv2[Z, Y] & X](s: S[X, F]): (Y, Y) = s match {
  // Error: found (pX, pY); Required (Y, Y)
  // Should compile
  case p: P[pX, pY, pF] => (p.x, p.y)
}
```

Listing 7 – Intricate false negative example

1.3 Chapters overview

This thesis is structured as follows. In chapter 2, we lay out our assumptions on some of Scala's subtyping rules. These assumptions allow us to present a constraint language in chapter 3. This constraint language plays a central part in enabling formal reasoning about the GADT inference problem.

We then employ the tools introduced in chapters 2 and 3 to present the GADT inference problem in chapter 4. This chapter also presents the necessary conditions to soundly solve the problem.

We present an algorithm in chapter 5 tackling the stated problem by chapter 4.

In chapter 6, we discuss related works which inspired the content of this thesis.

Finally, we conclude with chapter 7 and present further improvements on our proposition.

The appendix is divided into two parts. The first contains the proofs for the core parts of the algorithm. The second includes the auxiliary definitions needed for the algorithm. We do not provide proofs for the auxiliary functions but nonetheless state the (expected) properties.

Chapter 2

Framework

In this chapter, we present an extension of the pDOT calculus [12] with nominal subtyping and higher-kinded types. We do not give proofs for the claims as it goes well beyond the scope of this work. The extension enables reasoning about some of Scala's subtyping rules we expect, such as variant GADTs and higher-kinded abstractions.

Before diving into the heart of the matter, we introduce some notations we use throughout this work.

2.1 Preamble

Definition 2.1.1 (Disjoint set)**.** Given two sets A and B, we write $A \# B$ to denote that A and B are disjoint.

Definition 2.1.2 (Disjoint union)**.** Given two disjoint sets A and B, we write $A \uplus B$ for their disjoint union. We mainly use this notation to assert that A and B are disjoint.

Definition 2.1.3 (Boolean set)**.** We denote the set of boolean $\mathbb{B}$, comprised of the two values `true`, `false`

Definition 2.1.4 (Function copy)**.** Given a function $f : A \to B$ and two elements a in A and b in B, we write $f[a \mapsto b]$ for the function that maps a to b and otherwise agrees with f.

Definition 2.1.5 (Partial mapping)**.** We write $f : A \rightharpoonup B$ to denote a partial mapping from A to B. We write $f(a)\downarrow$ to denote that f is defined at a (that is, $a \in \operatorname{dom}(f)$). Conversely, we write $f(a)\uparrow$ to denote that f is not defined at a. To create a copy of f "undefining" an entry a, we write $f[a \mapsto \uparrow]$.

Definition 2.1.6 (Restriction of a function)**.** Given a function $f : A \to B$, we write $f \restriction A'$ to denote the restriction of the function f to $A' \cap A$.

Definition 2.1.7 (Unordered pairs set)**.** For any set A, we write $\binom{A}{2}$ for the set of unordered pairs created from A. That is, we define $\binom{A}{2}$ as $\big\{\{a, b\} : a, b \in A,\ a \neq b\big\}$.

We borrow Kleene's strong logic of indeterminacy [5] which adds a third indeterminate truth value.

Definition 2.1.8 (Ternary set)**.** We write K_3 to denote the ternary set of the three elements `true`, `false` and `undet`. The latter stands for **undet**ermined and conveys the notion of uncertainty.

The truth functions for negation, conjunction and disjunction are given by the following tables:

$\neg p$	
`true`	`false`
`false`	`true`
`undet`	`undet`

$p \wedge q$	`false`	`true`	`undet`
`false`	`false`	`false`	`false`
`true`	`false`	`true`	`undet`
`undet`	`false`	`undet`	`undet`

$p \vee q$	`false`	`true`	`undet`
`false`	`false`	`true`	`undet`
`true`	`true`	`true`	`true`
`undet`	`undet`	`true`	`undet`

Figure 2.1 – Truth tables for K_3 logic.

2.2 Syntax of types

We first introduce the syntax of types we consider in this report and give a brief overview alongside.

$T ::=$	**Type**
$\top$	*top*
$\bot$	*bottom*
X	*type variable*
$T \mathbin{\&} T$	*intersection*
$T \mid T$	*union*
$Cls[\vec{T}]$	*concrete type con. app.*
$F[\vec{T}]$	*abstract type con. app.*
$[\vec{v}\vec{X} \lhd B] \texttt{=>>} T$	*HK abstraction*
$\{z \texttt{=>} \vec{M}\}$	*refinement*
$p.\texttt{type}$	*singleton type*
$p.Q$	*path-dependent type*
$p.Q[\vec{T}]$	*path-dependent type app.*
$T\ \{z \texttt{=>} \vec{M}\}$	*refined type (syn. sugar)*
$(\vec{x} : \vec{T}) \texttt{=>} T$	*dep. fn. (syn. sugar)*
$[\vec{X} \lhd B] \texttt{=>} (\vec{x} : \vec{T}) \texttt{=>} T$	*pol. fn. (syn. sugar)*

$x,\ y,\ z$	**Term variable**
$X,\ Y,\ Z,\ F$	**Type variable**
$a,\ f$	**Field**
$A,\ Q$	**Type member**
Cls	**Class and trait**
B	**Bounds**
$p, q ::=$	**Path**
x	*variable*
$p.a$	*field selection*
$v ::=$	**Variance**
$+$	*covariance*
$-$	*contravariance*
$\pm$	*invariance*
$M ::=$	**Refinement member**
$\texttt{type}\ Q \lhd B$	*type member*
$\texttt{val}\ f : T$	*field*
$\texttt{def}\ m[\vec{X} \lhd B](\vec{x} : \vec{T}) : T$	*method*

Figure 2.2 – Syntax for types

We only consider all well-formed types generated by the above grammar. The well-formedness of a type naturally depends on its shape. For instance, the arguments of a type application must respect the arity, variance and kind of the type constructor. We assume this property is independent of a typing environment Γ [1].

We now go over some of the rules and present some explanations and introduce additional requirements that are not captured by the grammar. We also give the desugaring assumptions of refined types, dependent and polymorphic function types.

Match types, type lambdas.

Match types and type lambdas are new constructs introduced in Scala 3. For our specific needs, we assume these can be treated as if they were higher-kinded constructs. In particular, we suppose that match types can be seen as abstract type constructors, as we do not have a dedicated treatment for the former.

Type variable.

We use the meta-variables F and G to denote a higher-kinded type variable, and X for a type variable of any kind.

Intersection and union types.

Unsurprisingly, the types involved in intersection and union types must be of simple kind.

Type constructor application.

A class or trait may have an arity of zero. Abstract and path-dependent type constructors must have an arity of at least 1.

A type constructor application is well-formed if and only if the arguments match the type constructor signature.

---

[1] Technically speaking, this assumption cannot be satisfied for a path-dependent type application such as $p.F[\vec{A}]$. We need to perform a lookup in Γ to check the kind of $p.F$. However, we deem this assumption reasonable as it eases the well-formedness analysis.

Note that we do not expect a well-formed type application to be well-typed: to determine whether the arguments satisfy the type constructor bounds, we necessarily need an environment Γ. We capture this requirement in the concept of *conformance*, which we introduce afterwards.

Refined type.
A refined type such as $T\,\{z \,\texttt{=>}\, \vec{M}\}$ is simply desugared into $T \,\&\, \{z \,\texttt{=>}\, \vec{M}\}$.

Bounds.
Bounds are constructs allowing to constraint a type variable (such as in a higher-kinded abstraction or a method) or a type member by giving it lower and upper bounds.

Bounds allow constraining multiple type variables: the bounds of a type variable may refer to another constrained type variable [2].

We can view bounds as a partial mapping from type variables to pairs of types, where the first member corresponds to the lower bound, and the second member to the upper bound. We assume that the type variables of bounds have an implicit ordering.

We write $\vec{X} \triangleleft B$ to denote that the type variables $\vec{X}$ are subject to the constraints in B. We require $\vec{X}$ to coincide with the domain of B. If the kinds between the latter and the former correspond, we can perform an appropriate α-renaming to satisfy this condition; for type variables not appearing in B, we can create a copy of B that maps these to $(\bot, \top)$.

Dependent function type.
Given a dependent function type $(\vec{x} : \vec{S}) \,\texttt{=>}\, T$, we assume its corresponding desugaring is:

$$\begin{array}{l} \texttt{FunctionN}[\vec{S}', T'] \,\&\, \{\_ \,\texttt{=>} \\ \quad \texttt{def apply}(\vec{x} : \vec{S}) : T \\ \} \end{array}$$

where $N = |\vec{S}|$. T' is the least upper approximation of T with no reference to the term $\vec{x}$. Analogously, $\vec{S}'$ are the (point-wise) greatest lower approximation of $\vec{S}$ with no reference to $\vec{x}$.

The desugaring applies to plain functions as well.

Polymorphic function type.
Similar to the assumptions about dependent function types, we assume that the corresponding desugaring of a polymorphic function $[\vec{X} \triangleleft B] \,\texttt{=>}\, (\vec{x} : \vec{S}) \,\texttt{=>}\, T$ is:

$$\begin{array}{l} \texttt{FunctionN}[\vec{S}', T'] \,\&\, \{\_ \,\texttt{=>} \\ \quad \texttt{def apply}[\vec{X} \triangleleft B](\vec{x} : \vec{S}) : T \\ \} \end{array}$$

Higher-kinded abstraction.
Given a higher-kinded abstraction $[\vec{v}\vec{X} \triangleleft B] \,\texttt{=>>}\, T$, we require $\vec{v}$ – the variance sign vector – to have the same length as $\vec{X}$. Furthermore, we require $\vec{X}$ to coincide with the domain of B. Finally, we assume that T has a simple kind. We can always remediate with a suitable η-expansion and uncurrying.

Refinements.
We first introduce a shorthand notation for refinements and then discuss the well-formedness conditions.

---

[2] There are some restrictions that we do not consider for simplicity.

Suppose we have the following refinement:

$\{z$ `=>`
  `type` $T_1 \lhd B_1$
  $\ldots$
  `type` $T_m \lhd B_m$
  `val` $f_1 : F_1$
  $\ldots$
  `val` $f_n : F_n$
  `def` $m_1[\vec{Y}_1 \lhd B_{Y,1}](\vec{x} : \vec{U}_1) : V_1$
  $\ldots$
  `def` $m_o[\vec{Y}_o \lhd B_{Y,o}](\vec{x}_o : \vec{U}_o) : V_o$
$\}$

where m, n and o can be zero. In particular, a refinement can be empty.

We can describe the refinement more compactly as follows:

$\{z$ `=>`
  `type` $\vec{T} \lhd B$
  `val` $\vec{f} : \vec{F}$
  `def` $\overrightarrow{m[\vec{Y} \lhd B_Y](\vec{x} : \vec{U})} : \vec{V}$
$\}$

For a refinement to be well-formed, we naturally require all of its members to be well-formed. Furthermore, field names must be distinct; the same goes for type members. Methods follow the same restriction: in particular, we do not allow refinements to have method overloads. This rule coincides with Scala.

We now introduce some definitions.

Definition 2.2.1 (Type variables set)**.** For each kind κ, we let $\mathcal{V}_{X_\kappa}$ be a denumerable set of type variables. We assume that these sets are disjoint and note $\mathcal{V}_X$ the set formed by the union of $\mathcal{V}_{X_\kappa}$.

We employ the notation $\bar{X}$ to denote a finite, possibly empty set of type variables. We write $\vec{X}$ for an ordering of $\bar{X}$.

We write $\mathrm{ftv}(T)$ and $\mathrm{ftv}(B)$ for the set of free type variables of a type T and a bound B respectively. $\mathrm{ftv}(B)$ is defined as $\bigcup \{\mathrm{ftv}(L_i, U_i) : (L_i, U_i) \in \mathrm{Im}(B)\} \setminus \mathrm{dom}(B)$.

Definition 2.2.2 (Term variables set)**.** We let $\mathcal{V}_x$ be a denumerable set of term variables, disjoint from $\mathcal{V}_{X_\kappa}$.

We employ a similar notation for a set of term variables $\bar{x}$ and $\vec{x}$ for an ordering of $\bar{x}$.

We write $\mathrm{ftmv}(T)$ and $\mathrm{ftmv}(B)$ for the set of free term variables of a type T and a bound B, respectively. $\mathrm{ftmv}(B)$ is defined as $\bigcup \{\mathrm{ftmv}(L_i, U_i) : (L_i, U_i) \in \mathrm{Im}(B)\}$.

Definition 2.2.3 (Classes, traits and program identifiers sets)**.** We let the set $\mathcal{S}_C$ denote the set of classes and traits symbols. We employ the meta-variable *Cls* to denote elements of that set. We assume that this set is disjoint from $\mathcal{V}_X$ and from $\mathcal{V}_x$. That is, given a symbol X, we assume it is possible to tell whether it is a type variable or a class symbol.

We similarly let the set $\mathcal{S}_a$ denote the set of program identifiers: that is, the set of symbols used to bind values as well as defining fields of refinements. We employ the meta-variables a, b, f to denote elements of that set. Analogous to $\mathcal{S}_C$, we assume it is possible to differentiate an element of $\mathcal{S}_a$ from a term variable (which binds values too).

Definition 2.2.4 (Ground types sets, types sets)**.** We refer to closed, well-formed types formed from the grammar 2.2 as *ground types*. We employ the meta-variable $\mathcal{T}^{\text{cl}}$ to denote subsets of the set of all ground types. We similarly employ the meta-variable $\mathcal{T}$ to denote subsets of the set of well-formed types that are not necessarily closed.

Definition 2.2.5 (Ground paths sets, paths sets)**.** A *ground path* (or closed path) is a path formed from the program identifiers (the set $\mathcal{S}_a$); that is, it is an element of the set $\{(a_1, ..., a_n) : n \geq 1, a_i \in \mathcal{S}_a\}$. We use the meta-variable $\mathcal{P}^{\text{cl}}$ to denote subsets of the set of all ground paths.

A (not necessarily closed) path is either a closed path or a path whose prefix is a term variable. That is, it is an element of the set $\{(x, a_1, ..., a_n) : n \geq 0, x \in \mathcal{V}_x,\ a_i \in \mathcal{S}_a\} \cup \{(a_1, ..., a_n) : n \geq 1, a_i \in \mathcal{S}_a\}$. We similarly use the meta-variable $\mathcal{P}$ to denote subsets of the set of all paths.

We employ the meta-variables p and q to denote paths, whether closed or not.

Definition 2.2.6 (Ground bounds sets, bounds sets)**.** A *ground bound* is a closed, well-formed bound formed from a given set of ground types. We use the meta-variable $\mathcal{B}^{\text{cl}}$ to denote subsets of the set of all ground paths. Similarly, we use the meta-variable $\mathcal{B}$ to denote subsets of the set of all bounds, whether closed or not.

Finally, we employ a standard definition of type substitution [9, 10].

Definition 2.2.7 (Type substitution)**.** A *type substitution* σ is a possibly partial, kind-preserving mapping of type variables to types. Given a type T, $\sigma(T)$ is the type obtained by recursively substituting the free type variables within T to their mapped types in σ, augmented with the identity for type variables not contained in the domain of σ.

We write $\sigma[X \mapsto T]$ to denote the assignment mapping the type variable X to T and otherwise agrees with σ.

We define the bound substitution $\sigma(B)$ accordingly where the type variables in $\text{dom}(B)$ appear bound within B.

Example 2.2.1. Let $\sigma = [X \mapsto \texttt{Int}, F[X] \mapsto \texttt{List[X]}, Y \mapsto \texttt{String}]$.
Then $\sigma([X <: \texttt{Option}[X]] \texttt{=>} F[X \,\&\, Y]) = [X <: \texttt{Option}[X]] \texttt{=>} \texttt{List[X \& String]}$.

Definition 2.2.8 (Term substitution)**.** A *term substitution* ρ is a (possibly partial) mapping of term variables to paths. Given a path p with a term prefix x, $\rho(p)$ is the path obtained by substituting x in p to the mapped path in ρ, augmented with the identity for term variables not contained in the domain of ρ. Closed paths are idempotent under ρ.

We define $\rho[x \mapsto p]$ analogously.

Term substitution within types and bounds are defined accordingly and solely operates on free term variables.

Example 2.2.2. Let $\rho = [x \mapsto a.b, y \mapsto c.d]$.
Then $\rho(\{x \texttt{=>} \texttt{type T} <: F[x.T \,\&\, y.u.v.S]\}) = \{x \texttt{=>} \texttt{type T} <: F[x.T \,\&\, c.d.u.v.S]\}$

Before concluding this section, we introduce two definitions that are essentially syntactic shorthands.

Definition 2.2.9 (Bounds satisfaction)**.** Let $B : \mathcal{B}$ be a bound constraint whose ordered domain is $\vec{X}$. For any $\vec{A}$ in $\mathcal{T}^{|\vec{X}|}$ (with the same kind as $\vec{X}$ and whose free type variables are distinct from $\bar{X}$), we say that A *satisfies* B *under* Γ and write $\Gamma \vdash \vec{A} \lhd B$ if and only if the following holds:

$$\begin{aligned} \forall X_i \in \vec{X}.\ \Gamma \vdash [\vec{X} \mapsto \vec{A}]L_i <: A_i \wedge \\ \Gamma \vdash A_i <: [\vec{X} \mapsto \vec{A}]U_i \end{aligned}$$

where L_i and U_i are the associated lower and upper bound of X_i.

Definition 2.2.10 (Vectors of types subtyping)**.** Given two vector of types $\vec{S}$ and $\vec{T}$ of same length and kind, we define $\Gamma \vdash \vec{S} <: \vec{T}$ as $\bigwedge_i^{|\vec{S}|} \Gamma \vdash S_i <: T_i$. If the length of the vectors is zero, we define $\Gamma \vdash \vec{S} <: \vec{T}$ as `true`.

Given a variance sign v, we define $\Gamma \vdash S <:^v T$ as $\Gamma \vdash S <: T$ if $v = +$, as $\Gamma \vdash T <: S$ if $v = -$, and as $\Gamma \vdash S <: T \wedge \Gamma \vdash S <: T$ if $v = \pm$,

Finally, we define $\Gamma \vdash \vec{S} <:^{\vec{v}} \vec{T}$ by simply combining the two notations.

2.3 Subtyping assumptions

We assume the existence of an extension to the pDOT calculus [12] encompassing nominal subtyping, type variable subtyping assumptions and higher-kinded abstraction. Because pDOT does not possess some constructs such as higher-kinded types, we choose Scala's syntax over pDOT's for presenting the rules.

We expect the (extended) typing environment Γ to have the capability of recording subtyping relationships between traits and classes. For instance, if we have the following traits:

$$\begin{aligned}&\texttt{Tr1}[+A]\\&\texttt{Tr2}[+A]\ \texttt{extends Tr1}[\texttt{Option}[A]]\\&\texttt{Tr3}[+A,+B]\ \texttt{extends Tr2}[\texttt{List}[A]]\ \texttt{with Tr1}[B]\end{aligned}$$

then Γ would record that $\texttt{Tr2}[+A]$ extends $\texttt{Tr1}[+A]$ through the mapping $\boldsymbol{\sigma}(A) = \texttt{Option}[A]$ (and similarly for the relationship between `Tr3` and `Tr2`). We assume that Γ records that $\texttt{Tr3}[+A,+B]$ extends $\texttt{Tr1}[+A]$ *twice* through the mappings $\boldsymbol{\sigma}_1(A,B) = \texttt{Option}[\texttt{List}[A]]$ and $\boldsymbol{\sigma}_2(A,B) = B$.

Furthermore, we assume that Γ is extent to allow type variable subtyping assumptions such as $X >: L <: H$, specifying that X is bounded between L and H.

For simplicity, we do not distinguish classes from traits.

The figure 2.3 establishes the subtyping rules we expect from the extension. We do not present the typing rule and leave them unspecified. For simplicity, we assume prior α-renaming of bound type and term variables of the considered types to avoid any collision with the variables of Γ.

We now discuss some of the rules. The changes or additions with respect to pDOT subtyping rules are highlighted in gray.

***Rules* (Top) *and* (Bot).**

We assume the existence of $\top_\kappa$ and $\bot_\kappa$ for all kinds κ. We omit the κ subscript when the kind can be inferred from the context.

***Rule* (Met-<:-Met).**

The rule for method subtyping is the adaptation of pDOT's (All-<:-All) to allow a direct representation of type parameters. We have adopted an uncurried version of method subtyping. While the rule presentation is more cumbersome than its curried variant, the former is more appropriate for our use case.

We require $\vec{S}$ to not forward-reference the term variables $\vec{x}$ (the second premise does not enforce such forbidden construction). A more formal enforcement of this requirement would be to replace the second premise of the rule by $\Gamma, \vec{Y} \lhd B_1, x_1 : S_{1,1}, ..., x_{i-1} : S_{1,i-1} \vdash S_{2,i} <: S_{1,i}$, thus turning any forward-reference ill-formed. This alternative has not been retained due to its unwieldiness.

We point out that type parameters can be encoded in pDOT using *type tags* [12]. We have however favored the assumption of a direct representation for convenience.

***Rule* (Cls-<:-Cls).**

The $\Gamma \vdash \vec{S} \lhd B$ antecedent ensures that $\vec{S}$ satisfies the bounds B of the class (or trait) *Cls*. The vector $\vec{v}$ refers to the variance signs of *Cls*.

Because $\vec{S}$ is a subtype of $\vec{T}$ with respect to the signs of $\vec{v}$, $\vec{T}$ must satisfy the bounds of *Cls* as well.

***Rules* (Cls$_1$-<:Cls$_2$) and (Cls$_2$-<:Cls$_2$).**

B_1 and B_2 refer to the bounds of Cls_1 and Cls_2 respectively.

The rule (Cls$_1$-<:Cls$_2$) allows to "upcast" Cls_1 to Cls_2 while (Cls$_2$-<:Cls$_2$) is in essence the opposite of (Cls$_1$-<:Cls$_2$).

As an illustration, we consider the three traits introduced in the previous example. If $\Gamma \vdash \texttt{Tr3}[S,T] <: \texttt{Tr1}[U]$, the rule (Cls$_1$-<:Cls$_2$) states we necessarily have $\Gamma \vdash \texttt{Tr1}[\texttt{Option}[\texttt{List}[S]]] \,\&\, \texttt{Tr1}[T] <: \texttt{Tr1}[U]$. In other word, we must have $\texttt{Tr1}[\texttt{Option}[\texttt{List}[S]]] <: \texttt{Tr1}[U]$ or $\texttt{Tr1}[T] <: \texttt{Tr1}[U]$ therefore reducing the range of types that U can possibly take.

Rule **(Path-&).**

The intuition behind this rule is to enforce equality between arguments appearing in invariant positions within an intersection of traits or classes. For example, if we have $p : \texttt{Inv}[T]\ \&\ \texttt{Inv}[S]$ where `Inv` is an invariant trait, then T and S must be equal; otherwise, we could not have instantiated it.

This rule introduces unsoundness in presence of (implicit) `null` as well as explicit casts. We discuss in more detail these two issues in 5.4.

Rule **(Refn-<:-Refn).**

R_1 and R_2 refer to the following refinements:

$$\begin{array}{ll}
R_1 = \{z\ \texttt{=>} & R_2 = \{z\ \texttt{=>} \\
\quad \texttt{type}\ \vec{T} \lhd B_1 & \quad \texttt{type}\ \vec{T} \lhd B_2 \\
\quad \texttt{type}\ \vec{T'} \lhd B' & \\
\quad \texttt{val}\ \vec{f} : \vec{F}_1 & \quad \texttt{val}\ \vec{f} : \vec{F}_2 \\
\quad \texttt{val}\ \vec{f'} : \vec{F'} & \\
\quad \texttt{def}\ \overrightarrow{m[\vec{Y} \lhd B_{Y,1}](\vec{x} : \vec{U}_1)} : \vec{V}_1 & \quad \texttt{def}\ \overrightarrow{m[\vec{Y} \lhd B_{Y,2}](\vec{x} : \vec{U}_2)} : \vec{V}_2 \\
\quad \texttt{def}\ \overrightarrow{m'[\vec{Y'} \lhd B'_Y](\vec{x'} : \vec{U'})} : \vec{V'} & \\
\} & \}
\end{array}$$

We point out that this rule subsumes pDOT's (Fld-<:-Fld) and (Typ-<:-Typ) (explaining their absence).

Finally, we assume the four following axioms. The first two state the strengthening and weakening of the judgement for term variables.

Axiom 2.3.1 (Strengthening for term variables in subtyping derivations). If $\Gamma, x : T \vdash S_1 <: S_2$ and $x \notin \mathrm{ftmv}(S_1, S_2)$, then $\Gamma \vdash S_1 <: S_2$

Axiom 2.3.2 (Weakening for term variables in subtyping derivations). If $\Gamma \vdash S_1 <: S_2$ and $x \notin \mathrm{ftmv}(S_1, S_2)$, then $\Gamma, x : T \vdash S_1 <: S_2$

The last two allow explicit substitutions to be turned into a typing environment extension and vice-versa. δ is employed to denote subtyping and path typing.

Axiom 2.3.3 (Extensibility of Γ for type variables). $\Gamma, X >: L <: H \vdash \delta$ if and only if $\Gamma \vdash [X \mapsto A]L <: A$ and $\Gamma \vdash A <: [X \mapsto A]H$ imply $\Gamma \vdash [X \mapsto A]\delta$ for all $A \in \mathcal{T}^{\mathrm{cl}}$ with the same kind as X.

Axiom 2.3.4 (Extensibility of Γ for term variables). $\Gamma, x : T \vdash \delta$ if and only if $\Gamma \vdash p : T$ implies $\Gamma \vdash [x \mapsto p]\delta$ for all $p \in \mathcal{P}^{\mathrm{cl}}$.

Lemma 2.3.1. Let $\vec{x}$ and $\vec{T}$ be term variables and types of same length such that $\vec{T}$ does not forward-reference $\vec{x}$. Then $\Gamma, \vec{x} : \vec{T} \vdash \delta$ if and only if $\Gamma \vdash \vec{p} : [\vec{x} \mapsto \vec{p}]\vec{T}$ implies $\Gamma \vdash [\vec{x} \mapsto \vec{p}]\delta$ for all $\vec{p} \in (\mathcal{P}^{\mathrm{cl}})^{|\vec{x}|}$.

Proof. Straightforward induction on the size of $\vec{x}$ with δ held abstract, and application of axiom 2.3.4.

□

$$\Gamma \vdash T <: \top \qquad (\textsc{Top})$$

$$\Gamma \vdash \bot <: T \qquad (\textsc{Bot})$$

$$\Gamma \vdash T <: T \qquad (\textsc{Refl})$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \qquad (\textsc{Trans})$$

$$\Gamma \vdash T \mathbin{\&} U <: T \qquad (\textsc{And}_1\text{-}{<:})$$

$$\Gamma \vdash T \mathbin{\&} U <: U \qquad (\textsc{And}_2\text{-}{<:})$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \mathbin{\&} U} \qquad ({<:}\text{-}\textsc{And})$$

$$\frac{\Gamma \vdash p : \{\texttt{type}\ A >: S <: T\}}{\Gamma \vdash S <: p.A} \qquad ({<:}\text{-}\textsc{Sel})$$

$$\frac{\Gamma \vdash p : \{\texttt{type}\ A >: S <: T\}}{\Gamma \vdash p.A <: T} \qquad (\textsc{Sel}\text{-}{<:})$$

$$\frac{\Gamma \vdash p : q\texttt{.type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: [p \mapsto q]T} \qquad (\textsc{Sngl}_{pq}\text{-}{<:})$$

$$\frac{\Gamma \vdash p : q\texttt{.type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: [q \mapsto p]T} \qquad (\textsc{Sngl}_{qp}\text{-}{<:})$$

$$\frac{X >: L <: H \in \Gamma}{\Gamma \vdash L <: X} \qquad ({<:}\text{-}\textsc{TyVar})$$

$$\frac{X >: L <: H \in \Gamma}{\Gamma \vdash X <: H} \qquad (\textsc{TyVar}\text{-}{<:})$$

$$\Gamma \vdash T <: T \mid U \qquad ({<:}\text{-}\textsc{Or}_1)$$

$$\Gamma \vdash U <: T \mid U \qquad ({<:}\text{-}\textsc{Or}_2)$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash S <: T}{\Gamma \vdash S \mid T <: U} \qquad (\textsc{Or}\text{-}{<:})$$

$$\frac{\Gamma, \vec{X} \lhd B_1 \vdash \vec{X} \lhd B_2}{\Gamma \vdash B_1 <: B_2} \qquad (\textsc{Bnd}\text{-}{<:}\text{-}\textsc{Bnd})$$

$$\frac{\begin{array}{c}\Gamma, \vec{X} \lhd B_1 \vdash \vec{X} \lhd B_2 \\ \Gamma, \vec{X} \lhd B_1 \vdash S_1 <: S_2\end{array}}{\begin{array}{c}\Gamma \vdash [\vec{v}\vec{X} \lhd B_1] \texttt{=>>} S_1 <: \\ [\vec{v}\vec{X} \lhd B_2] \texttt{=>>} S_2\end{array}} \qquad (\textsc{Hk}\text{-}{<:}\text{-}\textsc{Hk})$$

$$\frac{\begin{array}{c}\Gamma, \vec{Y} \lhd B_1 \vdash \vec{Y} \lhd B_2 \\ \Gamma, \vec{Y} \lhd B_1, \vec{x} : \vec{S}_1 \vdash \vec{S}_2 <: \vec{S}_1 \\ \Gamma, \vec{Y} \lhd B_1, \vec{x} : \vec{S}_1 \vdash T_1 <: T_2\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{def}\ \ m[\vec{Y} \lhd B_1](\vec{x} : \vec{S}_1) : T_1 <: \\ \texttt{def}\ \ m[\vec{Y} \lhd B_2](\vec{x} : \vec{S}_2) : T_2\end{array}} \qquad (\textsc{Met}\text{-}{<:}\text{-}\textsc{Met})$$

$$\frac{\Gamma \vdash \vec{S} <:^{\vec{v}} \vec{T} \qquad \Gamma \vdash \vec{S} \lhd B}{\Gamma \vdash \mathit{Cls}[\vec{S}] <: \mathit{Cls}[\vec{T}]} \qquad (\textsc{Cls}\text{-}{<:}\text{-}\textsc{Cls})$$

$$\frac{\begin{array}{c}\mathit{Cls}_1 \text{ extends } \mathit{Cls}_2 \text{ through } \vec{\boldsymbol{\sigma}} \\ \Gamma \vdash \vec{T} \lhd B_1\end{array}}{\Gamma \vdash \mathit{Cls}_1[\vec{T}] <: \mathop{\&}_i^N \mathit{Cls}_2[\boldsymbol{\sigma}_i(\vec{T})]} \qquad (\textsc{Cls}_1\text{-}{<:}\textsc{Cls}_2)$$

$$\frac{\begin{array}{c}\mathit{Cls}_1 \text{ extends } \mathit{Cls}_2 \text{ through } \vec{\boldsymbol{\sigma}} \\ \Gamma \vdash \vec{S} \lhd B_1 \qquad \Gamma \vdash \vec{T} \lhd B_2 \\ \Gamma \vdash \mathit{Cls}_1[\vec{S}] <: \mathit{Cls}_2[\vec{T}]\end{array}}{\Gamma \vdash \mathop{\&}_i^N \mathit{Cls}_2[\boldsymbol{\sigma}_i(\vec{T})] <: \mathit{Cls}_2[\vec{S}]} \qquad (\textsc{Cls}_2\text{-}{<:}\textsc{Cls}_2)$$

$$\frac{\begin{array}{c}\Gamma \vdash \vec{S} \lhd B \qquad \Gamma \vdash \vec{T} \lhd B \\ \Gamma \vdash p : \mathit{Cls}[\vec{S}] \mathbin{\&} \mathit{Cls}[\vec{T}] \\ v_i = \pm\end{array}}{\Gamma \vdash S_i <: T_i \qquad \Gamma \vdash T_i <: S_i} \qquad (\textsc{Path}\text{-}\&)$$

$$\frac{\begin{array}{c}\Gamma, z : R_1 \vdash B_1 <: B_2 \\ \Gamma, z : R_1 \vdash \vec{F}_1 <: \vec{F}_2 \\ \Gamma, z : R_1 \vdash \\ \texttt{def}\ \ \overrightarrow{m[\vec{Y} \lhd B_{Y,1}](\vec{x} : \vec{U}_1)} : \vec{V}_1 <: \\ \texttt{def}\ \ \overrightarrow{m[\vec{Y} \lhd B_{Y,2}](\vec{x} : \vec{U}_2)} : \vec{V}_2\end{array}}{\Gamma \vdash R_1 <: R_2} \qquad (\textsc{Refn}\text{-}{<:}\text{-}\textsc{Refn})$$

Figure 2.3 – Presumed subtyping rules based on pDOT.

2.4 Conformance

In the last section, we have seen that some of the rules (such as (Cls-<:-Cls)) require the arguments of an applied trait or class to satisfy the bounds of that trait or class.

As an example, suppose that the class `MyClass`$[X <: $ `Int`$]$ is defined in some environment Γ. In particular, we point out its requirement towards its type parameter which must be a subtype of `Int`. Then, assuming that `Int` and `String` are defined and unrelated in the considered Γ, we say that the type `MyClass`$[$`String`$]$ is *non-conforming under* Γ because `String` does not satisfy the bounds requirement of `MyClass`. On the other hand, `MyClass[Int]` is naturally *conforming under* Γ.

This example motivates the concept of conformance.

Definition 2.4.1 (Conformance of a type under a Γ)**.** Given a typing environment Γ and a type T, we say that T is *conforming under* Γ if and only if $\mathrm{conf}_T(\Gamma, T)$ holds. Otherwise, we say that T is *non-conforming under* Γ.

The predicate $\mathrm{conf}_T(\Gamma, T)$ is defined in figure 2.4. For simplicity, we assume proper α-renaming of bound variables to avoid any collision when extending Γ.

$$
\mathrm{conf}_T(\Gamma, T) = \begin{cases}
\Gamma \vdash \vec{S} \lhd B \wedge \mathrm{conf}_T(\Gamma, \vec{S}) & \begin{array}{l}\text{if } T = \mathit{TyCon}[\vec{S}] \text{ where } B \text{ are the bounds}\\ \text{of } \mathit{TyCon}\end{array} \\
\mathrm{conf}_T(\Gamma, S) \wedge \mathrm{conf}_T(\Gamma, U) & \text{if } T = S \mathbin{\&} U \text{ or } T = S \mid U \\
\mathrm{conf}_T(\Gamma, B) \wedge \mathrm{conf}_T((\Gamma, \vec{X} \lhd B), S) & \text{if } T = [\vec{v}\vec{X} \lhd B] \mathbin{\mathtt{=>>}} S \\
\begin{array}{l}\Gamma \vdash \vec{S} \lhd B_L \wedge \Gamma \vdash \vec{S} \lhd B_U \wedge \\ \quad \mathrm{conf}_T(\Gamma, \vec{S})\end{array} & \begin{array}{l}\text{if } T = p.F[\vec{S}] \text{ where } B_L, B_U \text{ are the}\\ \text{lower and upper bounds of } p.F\end{array} \\
\begin{array}{l}\bigwedge_i \mathrm{conf}_T((\Gamma, X \lhd B), L_i) \wedge \\ \quad \bigwedge_i \mathrm{conf}_T((\Gamma, X \lhd B), U_i)\end{array} & \text{if } T = B = \bigcup \{X_i \mapsto (L_i, U_i)\} \\
\begin{array}{l}\mathrm{conf}_T((\Gamma, z : T), B) \wedge \\ \quad \mathrm{conf}_T((\Gamma, z : T), \vec{F}) \wedge \\ \quad \mathrm{conf}_T((\Gamma, z : T), \mathtt{def}\ \overrightarrow{m[\vec{Y} \lhd B_Y](\vec{x} : \vec{U})} : \vec{V})\end{array} & \begin{array}{l}\text{if } T = \{z \mathrel{\mathtt{=>}} \\ \quad \mathtt{type}\ \vec{T} \lhd B \\ \quad \mathtt{val}\ \vec{f} : \vec{F} \\ \quad \mathtt{def}\ \overrightarrow{m[\vec{Y} \lhd B_Y](\vec{x} : \vec{U})} : \vec{V} \\ \}\end{array} \\
\begin{array}{l}\mathrm{conf}_T(\Gamma, B_Y) \wedge \\ \quad \mathrm{conf}_T((\Gamma, \vec{Y} \lhd B_Y, \vec{x} : \vec{U}), \vec{U}) \wedge \\ \quad \mathrm{conf}_T((\Gamma, \vec{Y} \lhd B_Y, \vec{x} : \vec{U}), V)\end{array} & \text{if } T = m[\vec{Y} \lhd B_Y](\vec{x} : \vec{U}) : V \\
\mathtt{true} & \text{otherwise}
\end{cases}
$$

Figure 2.4 – Conformance of a type T under an environment Γ

We naturally extend the definition of conformance to a set of types.

Chapter 3

A constraint language

We devote this chapter to the definition of a constraint language $\mathscr{C}(\mathcal{T}^{\mathrm{cl}}, \mathcal{P}^{\mathrm{cl}}, \mathcal{V}_X, \mathcal{V}_x, \Gamma)$ parameterized by sets of ground types $\mathcal{T}^{\mathrm{cl}}$, of ground paths $\mathcal{P}^{\mathrm{cl}}$, of type variables $\mathcal{V}_X$, of term variables $\mathcal{V}_x$, and a typing environment Γ.

For the remainder of the work, we consider $\mathcal{T}^{\mathrm{cl}}$, $\mathcal{P}^{\mathrm{cl}}$, $\mathcal{V}_X$, $\mathcal{V}_x$ and Γ fixed. We require $\mathcal{T}^{\mathrm{cl}}$ to be conforming under Γ and all paths in $\mathcal{P}^{\mathrm{cl}}$ to be well-typed under Γ. The sets $\mathcal{T}$, $\mathcal{B}$ and $\mathcal{B}^{\mathrm{cl}}$ are accordingly generated from $\mathcal{V}_X$ and $\mathcal{T}^{\mathrm{cl}}$. $\mathcal{P}$ is generated from $\mathcal{V}_x$ and $\mathcal{P}^{\mathrm{cl}}$.

We first define the syntax. We then give the logical interpretation of the language. Finally, we establish some important results to help us in our journey.

This chapter is largely inspired by the constraint language introduced by François Pottier and Didier Rémy employed to view the inference problem in Damas and Milner's type system (DM) as a constraint solving problem [11].

3.1 Syntax

We now present the syntax of constraints. We then give some requirements that are not captured by the grammar.

$C ::=$	**Constraint**
$C_{\#}$	*core constraint*
$B \preceq B$	*bounds subtyping*
$C \curlywedge C$	*conjunction*
$\exists X.\, C$	*existential type quantification*
$\exists x.\, C$	*existential term quantification*
$T \asymp T$	*equality (syn. sugar)*
$T \preceq^{v} T$	*variance subtyping (syn. sugar)*
B	*bounds constraint (syn. sugar)*
$\vec{A} \lhd B$	*bounds satisfaction (syn. sugar)*

$C_{\#} ::=$	**Core constraint**
`true`	*truth*
`false`	*contradiction*
$T \preceq T$	*subtyping*
$p : T$	*path typing*

Figure 3.1 – Syntax for constraints

We require types tied in a subtyping constraint, such as $S \preceq T$, to be of same kind. Additionally the type T in a path typing $p : T$ must have a simple kind. Furthermore, two bounds B_1 and B_2 tied in a subtyping constraint must have the same domain with the same ordering and kind. If the length and the kind of the domain match, it is possible to create a copy of B_1 and B_2 with an α-renamed and reordered domain to satisfy this requirement. Finally, a bounds satisfaction constraint such as $\vec{A} \lhd B$ must have its constrained types $\vec{A}$ correspond in length and kind to the domain of B.

The desugaring for the four last constructs is rather straightforward:

- An equality constraint $S \asymp T$ desugars into $S \preceq T \curlywedge T \preceq S$.
- A variance subtyping $S \preceq^{v} T$ desugars into $S \preceq T$ if $v = +$, $T \preceq S$ if $v = -$, and $S \preceq T \curlywedge T \preceq S$ if $v = \pm$.
- A bounds constraint B desugars into:

$$\bigcurlywedge \{L_i \preceq X_i \curlywedge X_i \preceq U_i : (X_i, (L_i, U_i)) \in B\}$$

- A bounds satisfaction constraint $\vec{A} \lhd B$ desugars into:

$$\bigcurlywedge \{[\vec{X} \mapsto \vec{A}]L_i \preceq A_i \curlywedge A_i \preceq [\vec{X} \mapsto \vec{A}]U_i : (X_i, (L_i, U_i)) \in B\}$$

We denote $\mathcal{C}$ the set of all well-formed constraints.

The constraints being rather self-explanatory, we move on to the next section and introduce their interpretation.

3.2 Meaning of constraints

The meaning of a constraint is dependent on the closed types and closed paths assigned to the free type and term variables of the considered constraint. The notion of associating each type and term variable is captured in the concepts of *ground type assignment* and *ground path assignment*, which we define next.

Definition 3.2.1 (Ground type assignment)**.** A *ground type assignment* (or just *type assignment*) ϕ is a total, kind-preserving mapping from $\mathcal{V}_X$ into $\mathcal{T}^{\text{cl}}$. We write $\phi[X \mapsto T]$ to denote the assignment mapping the type variable X to T and otherwise agrees with ϕ. Ground type assignments are naturally extended into type substitutions, respecting bound variables scope. For convenience, we abuse notation and write $\phi(T)$ to denote the substitution of T under the mapping given by ϕ.

Remark. A type substitution resulting from a ground type assignment may yield a non-conforming type. See example 3.2.3.

Example 3.2.1 (Type assignment, simple kind)**.** Consider the ground type assignment $\phi' = \phi[X \mapsto \texttt{Int}, Y \mapsto \texttt{String}]$, for some unspecified ϕ. Then, $\phi'(\texttt{MyClass}[X, [+X] \texttt{=>>} X \ \& \ Y, Z])$ is equal to $\texttt{MyClass}[\texttt{Int}, [+X] \texttt{=>>} X \ \& \ \texttt{String}, \phi(Z)]$.

Example 3.2.2 (Type assignment, higher-kinded)**.** Consider the assignment $\phi' = \phi[F[X, Y] \mapsto \texttt{SomeClass}[X \ \& \ Y]]$. Then, $\phi'(F[X, \texttt{Int}] \ \& \ X)$ is equal to $\texttt{SomeClass}[\phi(X) \ \& \ \texttt{Int}] \ \& \ \phi(X)$.

Example 3.2.3 (Type assignment, non-conforming)**.** Let the assignment $\phi' = \phi[X \mapsto \texttt{Int}]$ and the class $\texttt{MyClass}[Y <: \texttt{String}]$. Then $\phi'(\texttt{MyClass}[X]) = \texttt{MyClass}[\texttt{Int}]$ is non-conforming.

This example shows that the well-formedness preservation property of an assignment ϕ is insufficient to guarantee meaningful types.

We define ground path assignments similarly.

Definition 3.2.2 (Ground path assignment)**.** A *ground path assignment* (or just *path assignment*) γ is a total mapping from $\mathcal{V}_x$ into $\mathcal{P}^{\text{cl}}$. Furthermore, ground path assignments are extended into path substitution – taking into account scoped term variables.

Remark. Akin to ground type assignment, a term substitution resulting from a ground path assignment may yield an ill-formed path, as illustrated by the next example.

Example 3.2.4 (Path assignment)**.** Let the path assignment $\gamma' = \gamma[x \mapsto p.a, y \mapsto p.b]$ and suppose that $\Gamma \vdash p : \{\texttt{val } a : \{\texttt{val } b : \texttt{Int}\}\}$. Then, $\gamma'(x) = p.a$ and $\gamma'(x.b) = p.a.b$ are both well-typed paths. On the other hand, $\gamma'(y) = p.b$ is ill-typed.

As for type assignments ϕ, we need to require a bit more from γ to guarantee meaningful path and types.

Now that we have separately defined type and path assignment, we are interested in combining them together.

Definition 3.2.3 (Type and term variable substitution)**.** We write $(\phi, \gamma)T$ for the simultaneous substitution of T by the substitution created by combining ϕ and γ.

We analogously write $(\phi, \gamma)\Gamma$ to denote the substitution of all type and term variables. The type variable assumptions $X >: L <: U$ are removed.

Fortunately, we do not have to worry about simultaneous substitution since it is possible to swap ϕ and γ, as stated by the following lemma.

Lemma 3.2.1 (Commutativity of ϕ and γ). For any assignments ϕ,γ and type $T \in \mathcal{T}$, $(\phi, \gamma)(T)$, $\phi(\gamma(T))$ and $\gamma(\phi(T))$ are equal.

Similarly, $(\phi, \gamma)\Gamma$, $\phi(\gamma(\Gamma))$ and $\gamma(\phi(\Gamma))$ are equal.

Proof. It is sufficient to prove that $\phi(\gamma(T))$ and $\gamma(\phi(T))$ are equal. The same reasoning applies to $\phi(\gamma(\Gamma))$ and $\gamma(\phi(\Gamma))$.

To do so, we can examine the domain and codomain of ϕ and γ and deduce that these are disjoint. The domain of ϕ is $\mathcal{V}_X$ while the codomain of γ is $\mathcal{P}^{\text{cl}}$. These are obviously disjoint. On the other hand, the codomain of ϕ is $\mathcal{T}^{\text{cl}}$ and the domain of γ is $\mathcal{V}_x$. Path-dependent types are elements of $\mathcal{T}^{\text{cl}}$, but since the paths are closed, no term variable can appear in a prefix position. As such, the codomain of ϕ and the domain of γ are disjoint.

□

We introduce the notion for *self-conformance* of the typing environment Γ under some given assignments ϕ, γ. The idea boils down to ensuring that Γ does not introduce typing assumptions through path-dependent types or type variables once these are substituted through ϕ, γ. Self-conformance is, in essence, a weaker variant of *inertness* of Γ in pDOT which requires all types in Γ to be the precise types of values [12].

Definition 3.2.4 (Self-conformance of Γ under assignments ϕ, γ)**.** Given the assignments ϕ, γ, we say that Γ is *self-conform under* ϕ, γ if and only if $\text{conf}_\Gamma(\Gamma, \phi, \gamma)$ holds.

The predicate $\text{conf}_\Gamma(\Gamma, \phi, \gamma)$ is defined below.

$$\text{conf}_\Gamma(\Gamma, \phi, \gamma) = \begin{cases} \texttt{true} & \text{if } \Gamma = \emptyset \\ (\phi, \gamma)\Gamma' \vdash (\phi, \gamma)L <: (\phi, \gamma)U \wedge \text{conf}_\Gamma(\Gamma', \phi, \gamma) & \text{if } \Gamma = \Gamma', X >: L <: U \\ \begin{array}{l} (\phi, \gamma)\big(\Gamma', x : \{z \texttt{=>} \texttt{type}\ \vec{T};\ \ \texttt{val}\ \ \vec{f} : \vec{F}\}\big) \vdash \\ \quad (\phi, \gamma)\big([z \mapsto x]L_i\big) <: (\phi, \gamma)\big([z \mapsto x]U_i\big) \wedge \\ \quad \text{conf}_T((\phi, \gamma)\big(\Gamma', x : \{z \texttt{=>} \texttt{type}\ \vec{T};\ \ \texttt{val}\ \ \vec{f} : \vec{F}\}\big), \\ \qquad\qquad (\phi, \gamma)\big([z \mapsto x]\vec{F}\big)) \wedge \\ \quad \text{conf}_\Gamma(\Gamma', \phi, \gamma) \end{array} & \begin{array}{l} \text{if } \Gamma = \Gamma', x : \{z \texttt{=>} \\ \quad \texttt{type}\ \vec{T} \lhd B \\ \quad \texttt{val}\ \ \vec{f} : \vec{F}, \ldots \\ \} \end{array} \\ \text{conf}_\Gamma(\Gamma', \phi, \gamma) & \text{if } \Gamma = \Gamma', \delta \end{cases}$$

Figure 3.2 – Self-conformance of Γ under ϕ, γ

In the third case, we extend Γ' with a weaker type for x where the bounds of its type members have been stripped away. The types L_i and U_i refer to the lower and upper bounds specified by B.

In the last case, δ refers to any typing information not related to term typing or type variable bounds (e.g. declaration of inheritance between traits).

Now that we have all the necessary tools, we can introduce the constraints interpretation in the form of a judgement. The rules below define the constraint satisfaction predicate $\models$.

$$\frac{\mathrm{conf}_\Gamma(\Gamma, \phi, \gamma)}{\phi, \gamma \models \texttt{true}} \quad (\text{CM-True})$$

$$\frac{\begin{array}{c}\mathrm{conf}_\Gamma(\Gamma, \phi, \gamma) \\ \mathrm{conf}_T(\Gamma, T_1) \qquad \mathrm{conf}_T(\Gamma, T_2) \\ (\phi, \gamma)\Gamma \vdash (\phi, \gamma)T_1 <: (\phi, \gamma)T_2\end{array}}{\phi, \gamma \models T_1 \preceq T_2} \quad (\text{CM-TSubtype})$$

$$\frac{\begin{array}{c}\mathrm{conf}_\Gamma(\Gamma, \phi, \gamma) \\ \mathrm{conf}_T(\Gamma, B_1) \qquad \mathrm{conf}_T(\Gamma, B_2) \\ (\phi, \gamma)\Gamma \vdash (\phi, \gamma)B_1 <: (\phi, \gamma)B_2\end{array}}{\phi, \gamma \models B_1 \preceq B_2} \quad (\text{CM-BSubtype})$$

$$\frac{\begin{array}{c}\mathrm{conf}_\Gamma(\Gamma, \phi, \gamma) \qquad \mathrm{conf}_T(\Gamma, T) \\ (\phi, \gamma)\Gamma \vdash \gamma(p) : (\phi, \gamma)T\end{array}}{\phi, \gamma \models p : T} \quad (\text{CM-PathTyped})$$

$$\frac{\phi, \gamma \models C_1 \qquad \phi, \gamma \models C_2}{\phi, \gamma \models C_1 \curlywedge C_2} \quad (\text{CM-And})$$

$$\frac{\phi[X \mapsto T], \gamma \models C}{\phi, \gamma \models \exists X.\, C} \quad (\text{CM-ExistsType})$$

$$\frac{\phi, \gamma[x \mapsto p] \models C}{\phi, \gamma \models \exists x.\, C} \quad (\text{CM-ExistsTerm})$$

Figure 3.3 – Constraints interpretation

We explain some of the rules, starting with (CM-True). This rule states that all assignments ϕ, γ are solutions to the constraint `true` as long as Γ is self-conform under them. This requirement is present for all rules involving a (direct) manipulation of Γ. As such, in the remainder of this work, we will only consider ϕ, γ for which Γ is self-conform.

(CM-TSubtype), (CM-BSubtype) and (CM-PathTyped) are the "bridges" allowing to go back and forth from the constraint world to the subtyping world. One may wonder why T_1 and T_2 (and similarly for B_1 and B_2) are checked for conformance *un-substituted*. Intuitively, the conformance requirement may be satisfied for substituted types and environment without being satisfiable if un-substituted. By requiring an un-substituted conformance, we weaken the judgement. However, it is more convenient to reason about the conformance of un-substituted types and environment Γ as we do not have to consider the assignments ϕ, γ. Similarly to the assumption of self-conformance, we will only treat types and bounds that are conform under Γ.

The rules (CM-ExistsType) and (CM-ExistsTerm) allow X and x to denote any ground types T and ground paths p satisfying C regardless of the present mapping in ϕ and γ respectively.

Proofs involving the explicit use of ϕ, γ can quickly become cumbersome. When possible, it is preferable to state logical properties of constraints in terms of *entailment* that we define next.

Definition 3.2.5 (Entailment)**.** Given two constraints C_1 and C_2, we say that C_1 *entails* C_2 – and write $C_1 \Vdash C_2$ – if and only if, for all assignments ϕ, γ satisfying C_1, ϕ, γ satisfy C_2 as well.

We deduce two intuitive lemmas.

Lemma 3.2.2 (Reflexivity and transitivity of $\Vdash$). The relation $\Vdash$ is reflexive and transitive.

Proof. Immediate. ☐

Lemma 3.2.3 (Entailment of `false`). If $C \Vdash$ `false`, then C is unsatisfiable.

Proof. Straightforward. ☐

It is natural to introduce equivalency of two constraints, which we define next.

Definition 3.2.6 (Equivalency)**.** Given two constraints C_1 and C_2, we say that C_1 and C_2 are *equivalent* – and write $C_1 \equiv C_2$ – if and only if $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$ hold.

Lemma 3.2.4 (Equivalency of $\equiv$). The relation $\equiv$ between constraints is an equivalence relation.

Proof. Trivial. □

3.3 Some constraints laws

We now present and prove some properties we deem useful for latter on, starting with an inversion lemma.

Lemma 3.3.1 (Inversion of the constraint satisfaction relation).

1. If $\phi, \gamma \models \texttt{true}$, then Γ is self-conform under ϕ, γ
2. If $\phi, \gamma \models T_1 \preceq T_2$, then $(\phi, \gamma)\Gamma \vdash (\phi, \gamma)T_1 <: (\phi, \gamma)T_2$, Γ is self-conform under ϕ, γ and T_1, T_2 are both conforming under Γ.
3. If $\phi, \gamma \models B_1 \preceq B_2$, then $(\phi, \gamma)\Gamma \vdash (\phi, \gamma)B_1 <: (\phi, \gamma)T_2$, Γ is self-conform under ϕ, γ and B_1, B_2 are both conforming under Γ.
4. If $\phi, \gamma \models p : T$, then $(\phi, \gamma)\Gamma \vdash \gamma(p) : (\phi, \gamma)T$, Γ is self-conform under ϕ, γ and T is conforming under Γ.
5. If $\phi, \gamma \models C_1 \curlywedge C_2$, then $\phi, \gamma \models C_1$ and $\phi, \gamma \models C_2$.
6. If $\phi, \gamma \models \exists x.\, C$, then $\phi, \gamma[x \mapsto p] \models C$ for some $p \in \mathcal{P}^{\text{cl}}$.
7. If $\phi, \gamma \models \exists X.\, C$, then $\phi[X \mapsto T], \gamma \models C$ for some $T \in \mathcal{T}^{\text{cl}}$ with the same kind as X.

Proof. Immediate from the definition of the constraint interpretation relation. □

The following lemma states that we are allowed "move" a ground type substitution from ϕ to the constraint in question and vice-versa.

Lemma 3.3.2. $\phi[X \mapsto A], \gamma \models C$ holds if and only if $\phi, \gamma \models [X \mapsto A]C$ holds

Proof. By structural induction on C. The assignments ϕ, γ are held abstract.
Cases* `true`, `false`*:
Trivial.

Case* $S \preceq T$*:
We have:

$$\begin{aligned}
& \phi[X \mapsto A], \gamma \models S \preceq T \\
\iff\ & (\phi, \gamma)\Gamma \vdash (\phi[X \mapsto A], \gamma)(S) <: (\phi[X \mapsto A], \gamma)(T) && \text{By (CM-TSubtype) and inversion lemma} \\
\iff\ & (\phi, \gamma)\Gamma \vdash \gamma(\phi[X \mapsto A](S)) <: \gamma(\phi[X \mapsto A](T)) && \text{By lemma 3.2.1} \\
\iff\ & (\phi, \gamma)\Gamma \vdash \gamma(\phi([X \mapsto A]S)) <: \gamma(\phi([X \mapsto A]T)) && \text{By definition of } \phi \\
\iff\ & (\phi, \gamma)\Gamma \vdash (\phi, \gamma)([X \mapsto A]S) <: (\phi, \gamma)([X \mapsto A]T) && \text{By lemma 3.2.1} \\
\iff\ & \phi, \gamma \models [X \mapsto A]S \preceq [X \mapsto A]T && \text{By (CM-TSubtype) and inversion lemma}
\end{aligned}$$

For the third derivation – where ϕ and $[X \mapsto A]$ get swapped –, the ($\Longrightarrow$) direction is possible because X does not appear free in $[X \mapsto A]S$, so it is possible to "push" $[X \mapsto A]$ into ϕ. The ($\Longleftarrow$) direction is straightforward, as we are simply creating a copy of ϕ which takes care of substituting X into A.

Case* $p : T$*:
Similar to the previous case. We instead make use of the (CM-PathTyped). Furthermore, any path p' (not necessarily closed) is idempotent under all assignment ϕ.

Case* $B_1 \preceq B_2$*:
Similar to the $S \preceq T$ case.

Case* $C_1 \curlywedge C_2$*:
Straightforward application of the IH.
Case* $\exists Y.\, C'$*,* $Y \neq X$*:
We have:

$$\begin{aligned}
& \phi[X \mapsto A], \gamma \models \exists Y.\, C' && \\
\iff\ & \phi[X \mapsto A][Y \mapsto A'], \gamma \models C' && \text{For some } A'\text{, by (CM-ExistsType) and inversion lemma} \\
\iff\ & \phi[Y \mapsto A'][X \mapsto A], \gamma \models C' && \text{By definition of } \phi \\
\iff\ & \phi[Y \mapsto A'], \gamma \models [X \mapsto A]C' && \text{By IH} \\
\iff\ & \phi, \gamma \models \exists Y.\, [X \mapsto A]C' && \text{By (CM-ExistsType) and inversion lemma} \\
\iff\ & \phi, \gamma \models [X \mapsto A](\exists Y.\, C') && \text{By property of substitution and assumption } X \neq Y
\end{aligned}$$

Case* $\exists X.\, C'$*:
We first remark that, by the property of substitution, $\phi[X \mapsto A](\exists X.\, C')$ is equal to $\phi(\exists X.\, C')$. Similarly, $\exists X.\, C'$ is idempotent under $[X \mapsto A]$.
Leveraging these two observations, we get:

$$\begin{aligned}
& \phi[X \mapsto A], \gamma \models \exists X.\, C' \\
\iff\ & \phi, \gamma \models \exists X.\, C' \\
\iff\ & \phi, \gamma \models [X \mapsto A](\exists X.\, C')
\end{aligned}$$

Case* $\exists p.\, C'$*:
Straightforward use of (CM-PathTyped), the inversion lemma, and the IH.

□

The lemma below is similar to the previous one and applies to γ instead.

Lemma 3.3.3. $\phi, \gamma[x \mapsto p] \models C$ holds if and only if $\phi, \gamma \models [x \mapsto p]C$ holds

Proof. The proof is similar to the previous one. We thus omit it. □

The lemma to follow states we can remap a type variable to any ground type, as long as that type variable does not appear free in the constraint in question.

Lemma 3.3.4. If $X \notin \mathrm{ftv}(C)$, then for all $A \in \mathcal{T}^{\mathrm{cl}}$ with the same kind as X, we have:

$$\phi, \gamma \models C \iff \phi[X \mapsto A], \gamma \models C$$

Proof. By structural induction on C. The assignments ϕ, γ are held abstract.
Cases* `true`, `false`*:
Trivial.

Case* $S \preceq T$*:
We have:

$$\begin{aligned}
& \phi, \gamma \models S \preceq T \\
\iff\ & (\phi,\gamma)\Gamma \vdash (\phi,\gamma)(S) <: (\phi,\gamma)(T) && \text{By (CM-TSubtype) and inversion lemma} \\
\iff\ & (\phi,\gamma)\Gamma \vdash \gamma(\phi(S)) <: \gamma(\phi(T)) && \text{By lemma 3.2.1} \\
\iff\ & (\phi,\gamma)\Gamma \vdash \gamma(\phi[X \mapsto A](S)) <: \gamma(\phi[X \mapsto A](T)) && \text{By assumptions that } X \notin \text{ftv}(S,T) \\
\iff\ & (\phi,\gamma)\Gamma \vdash (\phi,\gamma[X \mapsto A])(S) <: (\phi,\gamma[X \mapsto A])(T) && \text{By lemma 3.2.1} \\
\iff\ & \phi, \gamma[X \mapsto A] \models S \preceq T && \text{By (CM-TSubtype) and inversion lemma}
\end{aligned}$$

Case* $p : T$*:
Similar to the previous case.

Case* $B_1 \preceq B_2$*:
Similar to the $S \preceq T$ case.

Case* $C_1 \curlywedge C_2$*:
Straightforward application of the IH.

Case* $\exists Y.\, C'$*,* $Y \neq X$*:
We have:

$$\begin{aligned}
& \phi, \gamma \models \exists Y.\, C' \\
\iff\ & \phi[Y \mapsto A'], \gamma \models C' && \text{For some } A', \text{ by (CM-ExistsType) and inversion lemma} \\
\iff\ & \phi[Y \mapsto A'][X \mapsto A], \gamma \models C' && \text{By IH } (X \text{ does not appear in } C' \text{ by assumptions}) \\
\iff\ & \phi[X \mapsto A][Y \mapsto A'], \gamma \models C' && \text{By definition of } \phi \\
\iff\ & \phi[X \mapsto A], \gamma \models \exists Y.\, C' && \text{By (CM-ExistsType) and inversion lemma}
\end{aligned}$$

Case* $\exists X.\, C'$*:
We have $\phi(\exists X.\, C') = \phi[X \mapsto A](\exists X.\, C')$ by the property of substitution. The conclusion for this case is then immediate.

Case* $\exists p.\, C'$*:
Straightforward use of (CM-PathTyped), the inversion lemma, and the IH.

□

It is in general more convenient to work with entailment and avoid using the lower-level concepts of assignments ϕ, γ. Therefore, we give an entailment version for most lemmas (where it is applicable).

The corollary below is immediately deduced by the previous lemma and the lemma 3.3.2

Corollary. If $X \notin \text{ftv}(C_1)$, then for all $A \in \mathcal{T}^{\text{cl}}$ with the same kind as X, we have:

$$C_1 \Vdash C_2 \implies C_1 \Vdash [X \mapsto A]C_2$$

in particular, C_2 may contain X.

The lemma below is analogous to lemma 3.3.4 and applies to γ.

Lemma 3.3.5. If $x \notin \text{ftv}(C)$, then for all $p \in \mathcal{P}^{\text{cl}}$, we have:

$$\phi, \gamma \models C \iff \phi, \gamma[x \mapsto p] \models C$$

Proof. The proof is similar to the proof for lemma 3.3. ☐

Corollary. If $x \notin \mathrm{ftv}(C_1)$, then for all $p \in \mathcal{P}^{\mathrm{cl}}$, we have:

$$C_1 \Vdash C_2 \implies C_1 \Vdash [x \mapsto p]C_2$$

in particular, C_2 may contain x.

The three following lemmas are intuitive and may be implicitly used in the proofs to come.

Lemma 3.3.6 (Weakening, conclusion conjunction).

1. If $C_1 \Vdash C_2$ holds, then for any C, $C \curlywedge C_1 \Vdash C_2$ holds as well.
2. $C_1 \Vdash C_2$ and $C_1 \Vdash C_3$ hold if and only if $C_1 \Vdash C_2 \curlywedge C_3$ holds.
3. If $C_1 \Vdash C_2$ holds, then for any C, $C \curlywedge C_1 \Vdash C \curlywedge C_2$ holds as well.

Proof. Straightforward. ☐

Lemma 3.3.7. Let C_1 be a constraint and ϕ, γ any constraint satisfying C_1. If C_2 is unsatisfiable under all ϕ, γ satisfying C_1, the entailment $C_1 \curlywedge C_2 \Vdash \texttt{false}$ holds.

Proof. Straightforward. ☐

Lemma 3.3.8.

1. $T \asymp T \equiv \texttt{true}$
2. $S \asymp T \Vdash T \asymp S$
3. $S \preceq T \curlywedge T \preceq U \Vdash S \preceq U$
4. $S \asymp T \curlywedge T \asymp U \Vdash S \asymp U$

Proof. Immediate. ☐

3.4 Derived results from subtyping rules

We dedicate this section in adapting the subtyping assumptions presented in chapter 2 to the constraint world.

We start with an inversion lemma for the subtyping rules. Perhaps surprisingly, it is stated in terms of assignments ϕ, γ and only applies if Γ is self-conforming under ϕ, γ. A "naive" inversion lemma would be incorrect if we do not require self-conformance: indeed, a subtyping assumption such as $X >: L <: H$ could potentially introduce dud subtyping relationships between L and H that do not hold. Self-conformance enforces L to be a subtype of H (once substituted).

Lemma 3.4.1 (Inversion of the subtyping relation). For any assignments ϕ, γ such that Γ is self-conform under ϕ, γ, the following assertions hold. Primed symbols denote symbols applied to $(\phi, \gamma)(\cdot)$ (e.g. $\Gamma' = (\phi, \gamma)\Gamma$).

1. If $\Gamma' \vdash S' <: T' \mid U'$, then $\Gamma' \vdash S' <: T'$ or $\Gamma' \vdash S' <: U'$.
2. If $\Gamma' \vdash S' \mathbin{\&} T' <: U'$, then $\Gamma' \vdash S' <: U'$ or $\Gamma' \vdash T' <: U'$.
3. If $\Gamma' \vdash B'_1 <: B'_2$, then $\Gamma', \vec{X} \lhd B'_1 \vdash \vec{X} \lhd B'_2$
4. If $\Gamma' \vdash [\vec{v}\vec{X} \lhd B'_1] \texttt{=>>} S'_1 <: [\vec{v}\vec{X} \lhd B'_2] \texttt{=>>} S'_2$, then $\Gamma', \vec{X} \lhd B'_1 \vdash \vec{X} \lhd B'_2$ and $\Gamma', \vec{X} \lhd B'_1 \vdash S'_1 <: S'_2$

5. If $\Gamma' \vdash \texttt{def}\ \ m[\vec{Y} \lhd B'_1](\vec{x} : \vec{S}'_1) : T'_1 <: \texttt{def}\ \ m[\vec{Y} \lhd B'_2](\vec{x} : \vec{S}'_2) : T'_2$, then:
$$\begin{gathered}\Gamma', \vec{Y} \lhd B'_1 \vdash \vec{Y} \lhd B'_2 \\ \Gamma', \vec{Y} \lhd B'_1, \vec{x} : \vec{S}'_1 \vdash \vec{S}'_2 <: \vec{S}'_1 \\ \Gamma', \vec{Y} \lhd B'_1, \vec{x} : \vec{S}'_1 \vdash T'_1 <: T'_2\end{gathered}$$
6. If $\Gamma' \vdash Cls[\vec{S}'] <: Cls[\vec{T}']$, then $\Gamma' \vdash \vec{S}' <:^{\vec{v}} \vec{T}'$ and $\Gamma' \vdash \vec{S}' \lhd B'$
7. If $\Gamma' \vdash R_1 <: R_2$, then:
$$\begin{aligned}&\Gamma', z : R'_1 \vdash B'_1 <: B'_2 \\ &\Gamma', z : R'_1 \vdash \vec{F}'_1 <: \vec{F}'_2 \\ &\Gamma', z : R'_1 \vdash \texttt{def}\ \ \overrightarrow{m[\vec{Y} \lhd B'_{Y,1}](\vec{x} : \vec{U}'_1)} : \vec{V}'_1 <: \\ &\qquad\qquad\quad \texttt{def}\ \ \overrightarrow{m[\vec{Y} \lhd B'_{Y,2}](\vec{x} : \vec{U}'_2)} : \vec{V}'_2\end{aligned}$$
8. If a class Cls_1 of arity $M \geq 0$ does not extend a class Cls_2 (distinct from Cls_1) of arity $L \geq 0$, then for any $\vec{A} \in \mathcal{T}^M$ and any $\vec{B} \in \mathcal{T}^L$, $\Gamma' \nvdash Cls_1[\vec{A}'] <: Cls_2[\vec{B}']$.
9. If a member M of a refinement R_2 is not included in a refinement R_1, then $\Gamma' \nvdash R'_1 <: R'_2$.
10. For any refinement R, class Cls of arity $L \geq 0$, and types $\vec{A} \in \mathcal{T}^L$, $\Gamma' \nvdash R' <: Cls[\vec{A}']$.

Proof. Straightforward induction on the subtyping derivation. Self-conformance takes care of the assumptions introduced by (<:-TYVAR) and (TYVAR-<:) and combined with (TRANS). ☐

Lemma 3.4.2 (Bounds subtyping). Let $B_1, B_2 : \mathcal{B}$ be bound constraints with the same ordered domain $\vec{X}$.

Then, for any assignments ϕ, γ, we have:

$$\begin{gathered}\phi, \gamma \models B_1 \preceq B_2 \\ \Longleftrightarrow \\ \forall \vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_1 \implies \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_2\end{gathered}$$

where the quantified $\vec{A}$ has the same kind as $\vec{X}$.

Proof.

Direction ($\Longrightarrow$)**.**

We assume that $\vec{X} \mathbin{\#} \mathrm{ftv}(\Gamma)$. Such condition can always be satisfied with appropriate α-renaming.

From the inversion lemma of constraint meaning, we have $(\phi, \gamma)\Gamma \vdash (\phi, \gamma)B_1 <: (\phi, \gamma)B_2$. We then employ the subtyping inversion lemma to obtain $(\phi, \gamma)\Gamma, \vec{X} \lhd (\phi, \gamma)B_1 \vdash \vec{X} \lhd (\phi, \gamma)B_2$; note that $\vec{X}$ is bound in B_1 and B_2 and is therefore unaffected by ϕ.

In the following steps, we will need to replace the $\vec{X}$ within B_1 and B_2. As such, we should unwrap $\vec{X} \lhd (\phi, \gamma)B_1$ (and similarly $\vec{X} \lhd (\phi, \gamma)B_2$). The shorthand expands into $X_i >: (\phi', \gamma)L_{1,i} <: (\phi', \gamma)H_{1,i}$ for all $X_i \in \vec{X}$ with $(L_{1,i}, H_{1,i}) = B_1(X_i)$ and $\phi' = \phi[\vec{X} \mapsto \vec{X}]$; ϕ' ensures that we do not substitute the $\vec{X}$ because these are bound to B_1 and B_2. To range over all i, we shorten the syntax to $\vec{X} >: (\phi', \gamma)\vec{L}_1 <: (\phi', \gamma)\vec{H}_1$

We therefore have:

$$(\phi, \gamma)\Gamma, \vec{X} >: (\phi', \gamma)\vec{L}_1 <: (\phi', \gamma)\vec{H}_1 \vdash \vec{X} >: (\phi', \gamma)\vec{L}_2 <: (\phi', \gamma)\vec{H}_2$$

Applying axiom 2.3.3, we get for all closed $\vec{A}$:

$$\begin{gathered}(\phi, \gamma)\Gamma \vdash \vec{A} >: [\vec{X} \mapsto \vec{A}](\phi', \gamma)\vec{L}_1 <: [\vec{X} \mapsto \vec{A}](\phi', \gamma)\vec{H}_1 \\ \Longrightarrow \\ (\phi, \gamma)\Gamma \vdash \vec{A} >: [\vec{X} \mapsto \vec{A}](\phi', \gamma)\vec{L}_2 <: [\vec{X} \mapsto \vec{A}](\phi', \gamma)\vec{H}_2\end{gathered}$$

We can hoist (ϕ', γ) due to $\vec{A}$ being closed and ϕ' being the identity for $\vec{X}$. Then:

$$(\phi, \gamma)\Gamma \vdash (\phi', \gamma)(\vec{A} >: [\vec{X} \mapsto \vec{A}]\vec{L}_1 <: [\vec{X} \mapsto \vec{A}]\vec{H}_1)$$
$$\Longrightarrow$$
$$(\phi, \gamma)\Gamma \vdash (\phi', \gamma)(\vec{A} >: [\vec{X} \mapsto \vec{A}]\vec{L}_2 <: [\vec{X} \mapsto \vec{A}]\vec{H}_2)$$

Because $\bar{X} \# \mathrm{ftv}(\Gamma)$, $(\phi, \gamma)\Gamma$ and $(\phi', \gamma)\Gamma$ are equal. Then, rule (CM-TSubtype) gives us:

$$\phi', \gamma \models [\vec{X} \mapsto \vec{A}]\vec{L}_1 \preceq \vec{A} \curlywedge \vec{A} \preceq [\vec{X} \mapsto \vec{A}]\vec{H}_1$$
$$\Longrightarrow$$
$$\phi', \gamma \models [\vec{X} \mapsto \vec{A}]\vec{L}_2 \preceq \vec{A} \curlywedge \vec{A} \preceq [\vec{X} \mapsto \vec{A}]\vec{H}_2$$

Lemma 3.3.2 allows us to move $[\vec{X} \mapsto \vec{A}]$ to ϕ'.

$$\phi[\vec{X} \mapsto \vec{A}], \gamma \models \vec{L}_1 \preceq \vec{A} \curlywedge \vec{A} \preceq \vec{H}_1$$
$$\Longrightarrow$$
$$\phi[\vec{X} \mapsto \vec{A}], \gamma \models \vec{L}_2 \preceq \vec{A} \curlywedge \vec{A} \preceq \vec{H}_2$$

where we have used the fact that $\phi'[\vec{X} \mapsto \vec{A}] = \phi[\vec{X} \mapsto \vec{X}][\vec{X} \mapsto \vec{A}] = \phi[\vec{X} \mapsto \vec{A}]$

To conclude, it is sufficient to remark that B_1 – when viewed as a constraint – is a shorthand for $\vec{L}_1 \preceq \vec{A} \curlywedge \vec{A} \preceq \vec{H}_1$ (and similarly for B_2).

Direction **($\Longleftarrow$).**

The steps employed for the direction ($\Longrightarrow$) can also be taken backwards, where we replace the application of subtyping and constraint meaning rules by the respective inversion lemmas and vice-versa.

$\square$

Lemma 3.4.3 (Class subtyping). Let Cls a class of variance $\vec{v}$ and let $\vec{S}$ and $\vec{T}$ two vectors of types in $\mathcal{T}$ matching the signature of Cls.

Then, for any assignments ϕ, γ under which $\vec{S}$ and $\vec{T}$ satisfy the bounds of Cls, $\phi, \gamma \models Cls[\vec{S}] \preceq Cls[\vec{T}]$ if and only if $\phi, \gamma \models \vec{S} \preceq^{\vec{v}} \vec{T}$.

Proof. Straightforward application of rule (Cls-<:-Cls) and subtyping inversion lemma. $\square$

Corollary (Class subtyping). Under the same conditions, lemma 3.4.3 can be expressed as follows.

For any constraint C, if $C \Vdash Cls[\vec{S}] \preceq Cls[\vec{T}]$, then $C \Vdash \vec{S} \preceq^{\vec{v}} \vec{T}$.

For any constraint C entailing the bounds satisfaction of Cls by $\vec{S}$ and $\vec{T}$ – that is, if ϕ, γ satisfy C, then $(\phi, \gamma)\vec{S}$ and $(\phi, \gamma)\vec{T}$ satisfy the bounds of Cls – if $C \Vdash \vec{S} \preceq^{\vec{v}} \vec{T}$, then $C \Vdash Cls[\vec{S}] \preceq Cls[\vec{T}]$.

Lemma 3.4.4 (Extension of a class). Let a class Cls_1 of arity $L \geq 0$ extending N times a class Cls_2 of arity $M \geq 0$ through the mappings $\boldsymbol{\sigma}_i : \mathcal{T}^L \to \mathcal{T}^M$, $1 \leq i \leq N$

For any assignments ϕ, γ and any $\vec{S} \in \mathcal{T}^L$, if $(\phi, \gamma)S$ satisfies the bounds of Cls_1, then:

$$\phi, \gamma \models Cls_1[\vec{S}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})]$$

Furthermore, for any $\vec{T} \in \mathcal{T}^M$, if $\phi, \gamma \models Cls_1[\vec{S}] \preceq Cls_2[\vec{T}]$ then $\phi, \gamma \models \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})] \preceq Cls_2[\vec{T}]$

Proof. Straightforward application of rule (Cls$_1$-<:Cls$_2$). $\square$

Corollary. Under the same conditions, lemma 3.4.4 can be formulated as follows.

For any constraint C entailing the bounds satisfaction of Cls_1 by $\vec{S}$ – that is, if ϕ, γ satisfy C, then $(\phi, \gamma)\vec{S}$ satisfies the bounds of Cls_1 – we have:

$$C \Vdash Cls_1[\vec{S}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})]$$

Furthermore, for any $\vec{T} \in \mathcal{T}^M$, we have:

$$Cls_1[\vec{S}] \preceq Cls_2[\vec{T}] \Vdash \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})] \preceq Cls_2[\vec{T}]$$

Lemma 3.4.5 (Absence of subtyping irrefutability). If a class Cls_1 of arity $L \geq 0$ does not extend a class $Cls_2 \neq Cls_1$ of arity $M \geq 0$, then for any $\vec{S} \in \mathcal{T}^L$ and any $\vec{T} \in \mathcal{T}^M$ of appropriate kind, $Cls_1[\vec{S}] \preceq Cls_2[\vec{T}] \equiv \texttt{false}$.

Proof. Straightforward application of the subtyping inversion lemma. □

Lemma 3.4.6 (Higher-kinded abstractions subtyping). For any assignments ϕ, γ:

$$\phi, \gamma \models [\vec{v}\vec{X} \triangleleft B_1] \texttt{=>>} S_1 \preceq [\vec{v}\vec{X} \triangleleft B_2] \texttt{=>>} S_2$$

$$\iff$$

$$\forall \vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_1 \implies \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_2 \curlywedge S_1 \preceq S_2$$

where the quantified $\vec{A}$ has the same kind as $\vec{X}$.

Proof. Straightforward application of rules (Hk-<:-Hk) and subtyping inversion lemma, using similar steps as for the proof of lemma 3.4.2.

□

Lemma 3.4.7 (Intersection of invariant positions). Let Cls a class, and let $\vec{S}$ and $\vec{T}$ two vectors of types in $\mathcal{T}$ matching the signature of Cls.

Let p a path in $\mathcal{P}$. For any assignments ϕ, γ satisfying the constraint $p : Cls[\vec{S}] \,\&\, Cls[\vec{T}]$, the assertion $\phi, \gamma \models S_i \asymp T_i$ holds for all $i \in \{i : v_i = \pm, 1 \leq i \leq |\vec{v}|\}$, where $\vec{v}$ is the variance sign vector of Cls.

Proof. Straightforward application of rule (Path-&) The satisfaction of the bounds of Cls is granted by the assumptions that ϕ, γ satisfy $p : Cls[\vec{S}] \,\&\, Cls[\vec{T}]$.

□

Corollary. Under the same conditions, lemma 3.4 can be expressed as follows:

$$p : Cls[\vec{A}] \,\&\, Cls[\vec{B}] \Vdash A_i \asymp B_i$$

Lemma 3.4.8 (Methods subtyping). Let the two following methods with the same name, type parameters and term arguments:

$$\texttt{def}\ \ m[\vec{Y} \triangleleft B_1](\vec{x} : \vec{S}_1) : T_1$$
$$\texttt{def}\ \ m[\vec{Y} \triangleleft B_2](\vec{x} : \vec{S}_2) : T_2$$

Then, for any assignments ϕ, γ:

$$\phi, \gamma \models \texttt{def}\ \ m[\vec{Y} \triangleleft B_1](\vec{x} : \vec{S}_1) : T_1 \preceq$$
$$\texttt{def}\ \ m[\vec{Y} \triangleleft B_2](\vec{x} : \vec{S}_2) : T_2$$

$$\iff$$

$$\forall \vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{Y}|}, \vec{p} \in (\mathcal{P}^{\mathrm{cl}})^{|\vec{x}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_1 \curlywedge \vec{x} : \vec{S}_1 \implies$$
$$\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_2 \curlywedge \vec{S}_2 \preceq \vec{S}_1 \curlywedge T_1 \preceq T_2$$

where the quantified $\vec{A}$ has the same kind as $\vec{Y}$.

Proof.

Direction **$(\implies)$.**

We assume that $\mathrm{ftmv}(B_1, B_2) \# \bar{x}$. We also assume that $(\bar{Y} \cup \bar{x}) \# (\mathrm{ftv}(\Gamma) \cup \mathrm{ftmv}(\Gamma))$. Such conditions can always be satisfied with appropriate α-renaming.

By the inversion lemma of constraint meaning, we have:

$$\begin{aligned}(\phi, \gamma)\Gamma \vdash \texttt{def}\ \ & m[\vec{Y} \triangleleft (\phi', \gamma')B_1](\vec{x} : (\phi', \gamma')\vec{S}_1) : (\phi', \gamma')T_1 <: \\ \texttt{def}\ \ & m[\vec{Y} \triangleleft (\phi', \gamma')B_2](\vec{x} : (\phi', \gamma')\vec{S}_2) : (\phi', \gamma')T_2\end{aligned}$$

with $\phi' = \phi[\vec{Y} \mapsto \vec{Y}]$ and $\gamma' = \gamma[\vec{x} \mapsto \vec{x}]$. We observe that $(\phi, \gamma)\Gamma$ is equal to $(\phi', \gamma')\Gamma$ and that $(\phi', \gamma')B_1 = (\phi, \gamma)B_1$ (and similarly for B_2) thanks to the α-renaming assumption.

By the subtyping inversion lemma, we obtain:

$$\begin{aligned}&(\phi, \gamma)\Gamma, \vec{Y} \triangleleft (\phi', \gamma')B_1 \vdash \vec{Y} \triangleleft (\phi', \gamma')B_2 \\ &(\phi, \gamma)\Gamma, \vec{Y} \triangleleft (\phi', \gamma')B_1, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash (\phi', \gamma')\vec{S}_2 <: (\phi', \gamma')\vec{S}_1 \\ &(\phi, \gamma)\Gamma, \vec{Y} \triangleleft (\phi', \gamma')B_1, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash (\phi', \gamma')T_1 <: (\phi', \gamma')T_2\end{aligned}$$

We can apply a similar reasoning as for the proof of lemma 3.4.2 to deduce that $\phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_1$ implies $\phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_2$ for all closed $\vec{A}$. Because $\bar{x}$ does not appear free in B_1 and B_2, we have by lemma 3.3.3 that $\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_1$ implies $\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_2$ for all closed paths $\vec{p}$.

We now turn our attention on:

$$(\phi, \gamma)\Gamma, \vec{Y} \triangleleft (\phi', \gamma')B_1, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash (\phi', \gamma')\vec{S}_2 <: (\phi', \gamma')\vec{S}_1$$

We observe that we can swap $\vec{Y} \triangleleft (\phi', \gamma')B_1$ and $\vec{x} : (\phi', \gamma')\vec{S}_1$ in the extension of $(\phi, \gamma)\Gamma$ because $(\phi', \gamma')B_1$ is closed and therefore does not contain any free term variable.

Next, we again apply a similar reasoning as lemma 3.4.2 to deduce that:

$$(\phi, \gamma)\Gamma, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash \vec{Y} \triangleleft (\phi', \gamma')B_1$$

implies:

$$(\phi, \gamma)\Gamma, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash [\vec{Y} \mapsto \vec{A}](\phi', \gamma')\vec{S}_2 <: [\vec{Y} \mapsto \vec{A}](\phi', \gamma')\vec{S}_1$$

for all closed $\vec{A}$. By axiom 2.3.1, we can remove $\vec{x} : (\phi', \gamma')\vec{S}_1$ from the judgement $(\phi, \gamma)\Gamma, \vec{x} : (\phi', \gamma')\vec{S}_1 \vdash \vec{Y} \triangleleft (\phi', \gamma')B_1$.

Combining this with lemma 2.3.1, we obtain that the two following judgements:

$$\begin{aligned}&(\phi, \gamma)\Gamma \vdash \vec{Y} \triangleleft (\phi', \gamma')B_1 \\ &(\phi, \gamma)\Gamma \vdash \vec{p} : [\vec{x} \mapsto \vec{p}](\phi', \gamma')\vec{S}_1\end{aligned}$$

imply:

$$(\phi, \gamma)\Gamma \vdash [\vec{x} \mapsto \vec{p}]([\vec{Y} \mapsto \vec{A}](\phi', \gamma')\vec{S}_2) <: [\vec{x} \mapsto \vec{p}]([\vec{Y} \mapsto \vec{A}](\phi', \gamma')\vec{S}_1)$$

for all closed paths $\vec{p}$.

We can hoist (ϕ', γ') and combine them with $[\vec{x} \mapsto \vec{p}]$ and $[\vec{Y} \mapsto \vec{A}]$ to get:

$$(\phi, \gamma)\Gamma \vdash (\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}])\vec{S}_2 <: (\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}])\vec{S}_1$$

By assumptions, $\bar{Y}$ and $\bar{x}$ do not appear free in Γ, we can thus "update" ϕ and γ with $[\vec{x} \mapsto \vec{p}]$ and $[\vec{Y} \mapsto \vec{A}]$ respectively:

$$\begin{aligned}(\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}])\Gamma \vdash (\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}])\vec{S}_2 <:& \\ (\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}])\vec{S}_1&\end{aligned}$$

Using (CM-TSUBTYPE) and the previous derivations, we get that:

$$\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_1 \curlywedge \vec{x} : \vec{S}_1$$

implies

$$\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models B_2 \curlywedge \vec{S}_2 \preceq \vec{S}_1$$

for all closed types $\vec{A}$ and paths $\vec{p}$.

To obtain $\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models T_1 \preceq T_2$, we apply a similar reasoning as done for $\phi[\vec{Y} \mapsto \vec{A}], \gamma[\vec{x} \mapsto \vec{p}] \models \vec{S}_2 \preceq \vec{S}_1$.

Direction **($\Longleftarrow$).**

It suffices to take the ($\Longrightarrow$) direction backwards, where we employ the weakening axiom 2.3.2 instead of the strengthening axiom 2.3.1.

□

Lemma 3.4.9 (Refinements subtyping). Let R_1 and R_2 the following refinements:

$$\begin{array}{ll}
R_1 = \{z\ \texttt{=>} & R_2 = \{z\ \texttt{=>} \\
\quad \texttt{type}\ \vec{T} \triangleleft B_1 & \quad \texttt{type}\ \vec{T} \triangleleft B_2 \\
\quad \texttt{type}\ \vec{T}' \triangleleft B' & \\
\quad \texttt{val}\ \vec{f} : \vec{F}_1 & \quad \texttt{val}\ \vec{f} : \vec{F}_2 \\
\quad \texttt{val}\ \vec{f}' : \vec{F}' & \\
\quad \texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,1}](\vec{x} : \vec{U}_1)} : \vec{V}_1 & \quad \texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,2}](\vec{x} : \vec{U}_2)} : \vec{V}_2 \\
\quad \texttt{def}\ \overrightarrow{m'[\vec{Y}' \triangleleft B'_Y](\vec{x}' : \vec{U}')} : \vec{V}' & \\
\} & \}
\end{array}$$

Then, for any assignments ϕ, γ, the assertion $\phi, \gamma \models R_1 \preceq R_2$ holds if and only if, for all closed paths p such that $\phi, \gamma \models p : R_1$, the following holds:

$$\begin{aligned}
\phi, \gamma[z \mapsto p] \models B_1 \preceq B_2 \curlywedge \vec{F}_1 \preceq \vec{F}_2 \curlywedge & \\
& \texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,1}](\vec{x} : \vec{U}_1)} : \vec{V}_1 \preceq \\
& \texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,2}](\vec{x} : \vec{U}_2)} : \vec{V}_2
\end{aligned}$$

Proof. We follow a similar reasoning as the proof for lemma 3.4.8. □

Lemma 3.4.10 (Absence of refinements subtyping irrefutability). If a member M of a refinement R_2 is not included in a refinement R_1, then the constraint $R_1 \preceq R_2$ is unsatisfiable.

Proof. Straightforward application of the subtyping inversion lemma. □

Lemma 3.4.11 (Absence of refinement and class subtyping irrefutability). For any refinements R, classes Cls of arity $L \geq 0$, and types $\vec{A} \in (\mathcal{T}^{\mathrm{cl}})^L$, the constraint $R \preceq Cls[\vec{A}]$ is unsatisfiable.

Proof. Straightforward application of the subtyping inversion lemma. □

Lemma 3.4.12.

1. $S \preceq T \,\&\, U \Vdash S \preceq T \curlywedge S \preceq U$
2. $S \mid T \preceq U \Vdash S \preceq U \curlywedge T \preceq U$

3. If $\phi, \gamma \models S \preceq T \mid U$, then $\phi, \gamma \models S \preceq T$ or $\phi, \gamma \models S \preceq U$
4. If $\phi, \gamma \models S \,\&\, T \preceq U$, then $\phi, \gamma \models S \preceq U$ or $\phi, \gamma \models T \preceq U$

Proof. Straightforward application of the subtyping rules related to intersection and union types. $\square$

Lemma 3.4.13. $p : S \,\&\, T \curlywedge T \preceq U \Vdash p : S \,\&\, U$.

Proof. Straightforward application of the (presumed) subsumption typing rule. $\square$

Now that we have established the corresponding lemmas for all considered types, we can state the following lemma:

Lemma 3.4.14. For any type $S, T \in \mathcal{T}$ of same kind and any type $U \in \mathcal{T}$, the entailment $S \asymp T \Vdash [S \mapsto T]U \asymp U$ holds.

Proof. Straightforward structural induction on U. $\square$

3.5 Determinacy of types

Before concluding this chapter, we introduce the concept of *determined types*.

We deem it is best to explain the reason for their consideration by taking a detour and having a look at the big picture.

As we will see in the next chapter, the GADT inference problem is comprised of a constraint generation and a constraint simplification part. The former generates the assumptions introduced by the GADT pattern which are then passed to the simplification part. In chapter 5, we will see that the constraint simplification problem can be tackled by maintaining a structure of accumulated knowledge coming from sequentially processing the assumptions.

Intuitively, we would like to record constraints that are as simple as possible in order to exploit the accumulated knowledge. However, sometimes it is not possible to simplify a constraint without further examining the other GADT assumptions. To avoid losing potentially precious information, we should record the constraint and come back to it whenever we learn more information about its constituents.

As an example, suppose we are given the constraint $F[A] \preceq F[B]$, with F abstract. We record that constraint and keep on. After some amount of work on the other assumptions, we find out that F is equal to $[+X \lhd B]$ `=>>` `MyCovTrait`$[X]$ with B a trivial constraint (i.e. $\bot \preceq X \preceq \top$). Assuming the covariance of `MyCovTrait`, the previous constraint then simplifies into $A \preceq B$.

If we generalize the example a bit, we observe that the simplification process for a constraint of the form $T_1 \preceq T_2$ has the following outcomes:

- *Canonical*, that is, as simple as possible. Subtyping constraints such as $X \preceq T$ and $T \preceq X$ with T an arbitrary type (which may contain type variables as well) are canonical. `true` and `false` are canonical as well. This is the type of constraints we are aiming for. It is similar to values in programming languages.
- *Reducible*, for instance the constraint `MyCovTrait`$[A] \preceq$ `MyCovTrait`$[B]$ reduces to $A \preceq B$. In more complex cases, we have to consider the accumulated knowledge to reduce a constraint. It is similar to reducible expressions in programming languages.
- *Stuck*, a constraint that cannot be reduced and not canonical. For instance, $X \,\&\, Y \preceq T \,\&\, S$ is stuck since we cannot deduce anything about X or Y individually. It is similar to non-value normal forms in programming languages. Such constraints are not really useful at the time of examination, but we should keep them since further knowledge may enable them to get unblocked.

We can observe (without giving a proof) that these following types allow progression if they are tied in a subtyping constraint:

- Ground types, as they do not contain any free type variable. Then, we can simplify them into `true` or `false` because we know how to compare ground types.

- Traits and classes, thanks to their injectivity.
- A disjunctive normal form (DNF) of ground types, and applied traits or classes with further conditions which we discuss in more detail next. The conditions essentially boil down to having each type in the DNF be distinct.

In a way, these types are *determined* by their shape. Taking a previous example, in `MyCovTrait`$[A] \preceq$ `MyCovTrait`$[B]$, we get to compare two determined types, which reduces the constraint into $A \preceq B$. Another example is `MyCovTrait`$[A] \preceq$ `Int`, where the two types are determined too. In that case, the constraint reduces to `false`.

We formally define the determinacy of a type under some assignments ϕ, γ before considering an example involving DNFs.

Definition 3.5.1 (Determinacy of a type). Given some assignments ϕ, γ and a type $T \in \mathcal{T}$, we say that T is *determined under* ϕ, γ if and only if $\mathrm{det}(\phi, \gamma, T)$ holds.

The predicate det is defined below. We note that the third case only applies to non-trivial DNFs.

$$\mathrm{det}(\phi,\gamma,T) = \begin{cases} \texttt{true} & \text{if } \mathrm{ftv}(T) \cup \mathrm{ftmv}(T) = \emptyset \\ \texttt{true} & \text{if } T = Cls[\vec{S}] \\ \bigwedge_{i}^{n} \bigwedge_{j}^{m_i} \mathrm{det}(\phi,\gamma,T_{i,j}) \wedge & \text{if } T = \big|_{i}^{n} \&_{j}^{m_i} T_{i,j} \\ \quad \bigwedge_{i}^{n} \bigwedge_{j_1 \neq j_2}^{m_i} \phi,\gamma \not\models T_{i,j_1} \preceq T_{i,j_2} \wedge \phi,\gamma \not\models T_{i,j_2} \preceq T_{i,j_1} \quad \wedge & \\ \quad \bigwedge_{i_1 \neq i_2}^{n} \phi,\gamma \not\models \&_{j}^{m_{i_1}} T_{i_1,j} \preceq \&_{j}^{m_{i_2}} T_{i_2,j} \wedge & \\ \qquad \phi,\gamma \not\models \&_{j}^{m_{i_2}} T_{i_2,j} \preceq \&_{j}^{m_{i_1}} T_{i_1,j} & \\ \texttt{false} & \text{otherwise} \end{cases}$$

Figure 3.4 – Determinacy of a type T under assignments ϕ, γ

We give four examples illustrating the reduction of determined types.

Example 3.5.1. Let `TC1`$[A]$ and `TC2`$[A]$ be two unrelated traits. Then, for any S and T, the constraint `TC1`$[S] \preceq$ `TC2`$[T]$ reduces to `false`.

Example 3.5.2. Let the following traits:

`trait Inv`$[A]$
`trait TC1`$[A]$
`trait TC2`$[A]$ `extends TC1`$[$`Inv`$[A]]$

Then, the constraint `TC2`$[S] \preceq$ `TC1`$[T]$ reduces to `TC1`$[$`Inv`$[S]] \preceq$ `TC1`$[T]$ which is itself reducible to `Inv`$[S] \preceq$ `Inv`$[T']$ if T is of the form `Inv`$[T']$ and `false` otherwise.

Example 3.5.3. Let the traits `Cov1`$[+A]$, `Cov2`$[+A]$ and `Cov3`$[+A]$ be unrelated and covariant traits. Then, for any type A, B, and C, the type:

$$(\texttt{Cov1}[\texttt{Cov2}[A]]\ \&\ \texttt{Cov1}[\texttt{Cov3}[B]]) \mid (\texttt{Cov1}[\texttt{Cov2}[C]]\ \&\ \texttt{Int})$$

is determined. In the first conjunct, we neither have $\texttt{Cov1}[\texttt{Cov2}[A]] \preceq \texttt{Cov1}[\texttt{Cov3}[B]]$ nor $\texttt{Cov1}[\texttt{Cov3}[B]] \preceq \texttt{Cov1}[\texttt{Cov2}[A]]$ for any assignments γ, ϕ (thanks to `Cov2` and `Cov3` being unrelated). The same reasoning applies to the second conjunct.

Similarly, the two disjunctions $\texttt{Cov1}[\texttt{Cov2}[A]] \,\&\, \texttt{Cov1}[\texttt{Cov3}[B]]$ and $\texttt{Cov1}[\texttt{Cov2}[C]] \,\&\, \texttt{Int}$ are not subtype of each other.

Then, for any types R, S, T, U and V, the constraint:

$$\begin{gathered}(\texttt{Cov1}[\texttt{Cov2}[R]] \,\&\, \texttt{Cov1}[\texttt{Cov3}[S]]) \mid (\texttt{Cov1}[\texttt{Cov2}[T]] \,\&\, \texttt{Int}) \\ \preceq \\ (\texttt{Cov1}[\texttt{Cov2}[U]] \,\&\, \texttt{Cov1}[\texttt{Cov3}[V]]) \mid (\texttt{Cov1}[\texttt{Cov2}[W]] \,\&\, \texttt{Int})\end{gathered}$$

reduces to $R \preceq U \curlywedge S \preceq V \curlywedge T \preceq W$.

Example 3.5.4. Let $\texttt{CovChild}[+A]$ a trait extending a trait $\texttt{Cov}[+A]$. Let $\texttt{Inv1}[A]$ and $\texttt{Inv2}[A]$ be two unrelated traits. Then, for any types A and B the types $\texttt{CovChild}[\texttt{Inv1}[A]] \,\&\, \texttt{CovChild}[\texttt{Inv2}[B]]$ and $\texttt{Cov}[\texttt{Inv1}[A]] \,\&\, \texttt{Cov}[\texttt{Inv2}[B]]$ are determined.

Furthermore, the following constraint is reducible and eliminates all outer intersection and union:

$$\begin{gathered}\texttt{CovChild}[\texttt{Inv1}[S]] \,\&\, \texttt{CovChild}[\texttt{Inv2}[T]] \\ \preceq \texttt{Cov}[\texttt{Inv1}[U]] \,\&\, \texttt{Cov}[\texttt{Inv2}[V]]\end{gathered}$$

Indeed, we first reduce it to :

$$\begin{gathered}\texttt{CovChild}[\texttt{Inv1}[S]] \,\&\, \texttt{CovChild}[\texttt{Inv2}[T]] \preceq \texttt{Cov}[\texttt{Inv1}[U]] \\ \curlywedge \\ \texttt{CovChild}[\texttt{Inv1}[S]] \,\&\, \texttt{CovChild}[\texttt{Inv2}[T]] \preceq \texttt{Cov}[\texttt{Inv2}[V]]\end{gathered}$$

For the first conjunct, we have:

$$\begin{gathered}\texttt{CovChild}[\texttt{Inv1}[S]] \preceq \texttt{Cov}[\texttt{Inv1}[U]] \\ \text{or} \\ \texttt{CovChild}[\texttt{Inv2}[T]] \preceq \texttt{Cov}[\texttt{Inv1}[U]]\end{gathered}$$

Both constraints reduce to:

$$\begin{gathered}\texttt{Cov}[\texttt{Inv1}[S]] \preceq \texttt{Cov}[\texttt{Inv1}[U]] \\ \text{or} \\ \texttt{Cov}[\texttt{Inv2}[T]] \preceq \texttt{Cov}[\texttt{Inv1}[U]]\end{gathered}$$

The first disjunction reduces to $S \asymp U$ while the second one to `false`. We similarly get that the second conjunct reduces to $T \asymp V$. The original constraint therefore reduces to $S \asymp U \curlywedge T \asymp V$.

Determinacy of types plays an important part of the simplification algorithm. However, the definition relies on subtyping querying with a given set of assumptions. Since type-checking in DOT and pDOT is thought undecidable [12, 13], we have to resort to using an approximated, conservative version of determinacy.

Finally, we point out that the concept of determinacy is not needed for proving soundness of any constraint simplification algorithm. It is nonetheless helpful when describing the strategies employed by the proposed algorithm.

Chapter 4

GADTs constraint reasoning principles

In this relatively short chapter, we present the GADT inference problem as a constraint generation and simplification problem leveraging the constraint language $\mathscr{C}$ introduced in the previous chapter.

Before doing so, we first need to set up the context in which the GADT inference problem is situated.

4.1 Context

We informally describe the instantiation of the parameters of $\mathscr{C}$, that is, $\mathcal{T}^{\text{cl}}$, $\mathcal{P}^{\text{cl}}$, $\mathcal{V}_X$, $\mathcal{V}_x$ and Γ.

Γ is set to the typing environment just after the introduction of the pattern match. The set $\mathcal{T}^{\text{cl}}$ is generated from the set of classes and traits symbols visible in the scope of the problem. $\mathcal{V}_X$ contains all type variables in scope, plus an infinite and denumerable set of distinct type variables used for fresh type variables. $\mathcal{V}_x$ is similarly created. A peculiarity is that we consider the terms introduced within the enclosing function of the pattern match as term variables and are therefore elements of $\mathcal{V}_x$. The set $\mathcal{P}^{\text{cl}}$ is composed of all terms and paths that are "outside" of the considered function.

As an example, consider the following snippet where we are interested in the inference problems at line 9 and line 12:

```
1  // Assuming Foo has a field named "f"
2  val x: Foo = Foo(f = 42)
3  // ...
4  def enclosing[X](a: Bar) {
5      // ...
6      def patmat[Y](s1: Qux, s2: Foo) = {
7          val b: String = "hello"
8          s1 match {
9              case p1: Foo =>
10                 val y: Int = 12
11                 // ...
12             case p2: Bar =>
13                 // ...
14             // ...
15         }
16     }
17 }
```

Listing 8 – GADT problem within a scope introducing type and term variables.

In both problems, $\mathcal{P}^{\text{cl}}$ would (among other) contain x, $x.f$ and a, but not $s1$, $s2$ or b. These would be contained in $\mathcal{V}_x$ and $\mathcal{P}$. Furthermore, $s2.f$ would be contained in $\mathcal{P}$ as well. In the problem at line 9, $p1$ would be contained in $\mathcal{V}_x$, and Γ would include $p1 : \texttt{Foo}$. However, $y : \texttt{Int}$ is outside of the scope of the problem and would therefore not be contained in Γ. The situation is analogous for the problem at line 12 with $p2 : \texttt{Bar}$.

We furthermore assume, for simplicity, that the matched expression is in administrative normal form. Finally, it is worthwhile to point out that the sets $\mathcal{T}^{\text{cl}}$, $\mathcal{P}^{\text{cl}}$, $\mathcal{V}_X$ and $\mathcal{V}_x$ are solely considered for establishing a setup for $\mathscr{C}$. In an implementation, we do not need to be concerned about them.

4.2 Constraint generation

Given a scrutinee $s : S$ and a pattern case $p : P$, we are interested in generating a constraint C_G describing the GADT inference problem.

Parreaux and Boruch-Gruszecki [7], and Waśko [16] explain in details the constraints brought in scope when a $p : P$ matches a scrutinee $s : S$. In essence, the bound pattern variable p must be of type $P \mathbin{\&} s.\texttt{type}$. For our needs, it is simpler to instead consider $p : P \mathbin{\&} S$.

Furthermore, it is possible to have further assumptions on the type or term variables. For instance, a type parameter in a function can be bound-constrained. We assume that such assumptions can be encoded in the constraint language $\mathscr{C}$; we denote these conjunctions of constraints as C.

Finally, Parreaux and Boruch-Gruszecki [7] argue that, in presence of a pattern whose type is a final class, the type of the pattern must be a subtype of S as well.

Based on the cited works, the constraints generation phase proceeds as follows. If P is not a final class, the constraint $(p : P \mathbin{\&} S) \curlywedge C$ is generated, where C is a (possibly trivial) constraint capturing further assumptions brought by the pattern or the enclosing scope. If P is a final class, we instead generate the more precise constraint $P \preccurlyeq S \curlywedge (p : P \mathbin{\&} S) \curlywedge C$.

We now give the generated constraints for listings 5 through 7.

Example 4.2.1 (Generated constraints for listing 5)**.** We simply generate:

$$p : \texttt{P} \mathbin{\&} \texttt{S}[X, Y]$$

In this example, C is trivial due to the lack of type or term variables constraining.

Example 4.2.2 (Generated constraints for listing 6)**.** This time, the pattern $p : \texttt{P}[pX]$ introduces an existentially quantified type variable pX. We thus get:

$$p : \texttt{P}[pX] \mathbin{\&} \texttt{S}[\texttt{Inv}[X] \mathbin{\&} Y]$$

Example 4.2.3 (Generated constraints for listing 7)**.** Here, we can incorporate further knowledge about $F[Z]$ and pF by adding these to C_G:

$$\begin{gathered} p : \texttt{P}[pX, pY, pF] \mathbin{\&} \texttt{S}[X, F] \curlywedge \\ [Z] \texttt{=>>} F[Z] \preccurlyeq [Z] \texttt{=>>} \texttt{Inv2}[Z, Y] \mathbin{\&} X \curlywedge \\ [Z] \texttt{=>>} \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \preccurlyeq [Z] \texttt{=>>} pF[Z] \end{gathered}$$

We have omitted the variance signs (defaulting to invariance i.e. ±) and the (trivial) bounds in the higher-kinded abstraction for clarity.

The second conjunct comes from the bound constraints on F introduced by the function `patmat` while the third originates from the constraint on the third type parameter of `P`.

4.3 Constraints simplification

Given the GADTs base constraints C_G, we are interested in computing a C' such that C_G entails C'. In other terms, we are interested in computing a *necessary* constraint C'. Naturally, we are striving to be as precise as possible: otherwise, the problem is trivially solved by simply returning `true`.

We may ask ourselves the reason why we are looking for a *necessary* constraint C' and not, let's say, a *sufficient* one. In fact, sufficient conditions are to type inference as necessary conditions are to GADT inference. In the type inference problem, we are interested in computing a C' entailing the given constraint problem.

Let us illustrate this with an adapted version of listing 6:

```scala
1  trait Inv[X]
2  trait S[X]
3  final class P extends S[Inv[Int] & Inv[String]]
4
5  def patmat[X, Y](s: S[Inv[X] & Inv[Y]], y: Y): Y = s match {
6      case p: P => y
7  }
8
9  val p = new P
10 val got1: String = patmat(p, "a")
11 // The annotation is here to help type inference.
12 val got2: Int = patmat(p: S[Inv[String] & Inv[Int]], 3)
```

Listing 9 – An altered version of listing 6

Starting with the type inference problem, at line 10, we are (intuitively) given the constraint C_I:

$$P \preceq \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]] \curlywedge \texttt{String} \preceq Y$$

The problem of type inference is interested in simplifying the given constraint "bottom-up". For the call at line 10, we remark that a solution found by the compiler is:

$$X \asymp \texttt{Int} \curlywedge Y \asymp \texttt{String}$$

This solution is incontestably correct. It is nonetheless useful to prove it, by starting with the solution and going downward the original constraint:

$$\begin{aligned}
X \asymp \texttt{Int} \curlywedge Y \asymp \texttt{String} \Vdash\ & X \asymp \texttt{Int} \curlywedge Y \asymp \texttt{String} \curlywedge \\
& \texttt{P} \preceq \texttt{S}[\texttt{Inv}[\texttt{Int}] \,\&\, \texttt{Inv}[\texttt{String}]] && \text{By lemmas 3.3.6 and 3.4.4} \\
\Vdash\ & \texttt{P} \preceq \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]] \curlywedge \texttt{String} \preceq Y && \text{By lemma 3.4.14}
\end{aligned}$$

At line 12, the compiler finds another solution to the same problem (i.e., the same original constraint):

$$X \asymp \texttt{String} \curlywedge Y \asymp \texttt{Int}$$

We can again see that it entails the original constraint:

$$\begin{aligned}
X \asymp \texttt{String} \curlywedge Y \asymp \texttt{Int} \Vdash\ & X \asymp \texttt{String} \curlywedge Y \asymp \texttt{Int} \curlywedge \\
& \texttt{P} \preceq \texttt{S}[\texttt{Inv}[\texttt{String}] \,\&\, \texttt{Inv}[\texttt{Int}]] && \text{By lemmas 3.3.6 and 3.4.4} \\
\Vdash\ & \texttt{P} \preceq \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]] \curlywedge \texttt{String} \preceq Y && \text{By lemma 3.4.14}
\end{aligned}$$

We observe that the type inference problem is seeking a *sufficient* solution.

We now turn our attention on the GADT inference, at line 6. We are interested in computing a *necessary* solution. For C_G, we are given the constraints:

$$\texttt{P} \preceq \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]] \curlywedge (p : \texttt{P} \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[X]])$$

We have by lemma 3.4.4:

$$\begin{aligned}
C_G \Vdash\ & \texttt{S}[\texttt{Inv}[\texttt{String}] \,\&\, \texttt{Inv}[\texttt{Int}]] \preceq \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]] \curlywedge \\
& p : \texttt{S}[\texttt{Inv}[\texttt{String}] \,\&\, \texttt{Inv}[\texttt{Int}]] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, \texttt{Inv}[X]] \\
\Vdash\ & \texttt{Inv}[\texttt{String}] \,\&\, \texttt{Inv}[\texttt{Int}] \asymp \texttt{Inv}[X] \,\&\, \texttt{Inv}[Y]
\end{aligned}$$

We cannot deduce anything about X and Y, which is to be expected because line 10 instantiates X to `Int` and Y to `String` while line 12 instantiates them the other way around.

We now go over listings 5 through 7 and compute a necessary solution for each of these.

Example 4.3.1 (A solution for listing 5)**.** A correct solution is $X \asymp \texttt{Int}$. We indeed have:

$$\begin{aligned} p : \texttt{P} \,\&\, \texttt{S}[X, Y] &\Vdash p : \texttt{S}[\texttt{Int}, \texttt{String}] \,\&\, \texttt{S}[X, Y] && \text{By lemmas 3.4.12 and 3.4.4} \\ &\Vdash X \asymp \texttt{Int} && \text{By lemmas 3.4.7} \end{aligned}$$

We remark that nothing is inferred for Y: there are no laws allowing to extract any useful information. It may be surprising that we cannot deduce $Y \preceq \texttt{String}$. It is due to its covariance definition in `S`. Giarrusso [3], Parreaux and Boruch-Gruszecki [7] remark that inferring such inequality is unsound. Indeed, since `P` is not final, one could declare the trait `WickedP` extending both `S` and `P`:

```scala
// Note: does not compile
trait Foo
trait S[X, +Y]
trait P extends S[Int, String]
trait WickedP extends P with S[Int, Foo]

def patmat[X, Y](s: S[X, Y]): Y = s match {
  case p: P => "Strings not allowed"
}
// Would produce a CastClassException if allowed
val got: Foo & String = patmat(new WickedP{})
```

Example 4.3.2 (A solution for listing 6)**.** A correct solution is:

$$\begin{aligned} p : \texttt{P}[pX] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, Y] &\Vdash p : \texttt{S}[pX \,\&\, \texttt{Inv}[\texttt{String}]] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, Y] && \text{By lemmas 3.4.12 and 3.4.4} \\ &\Vdash \texttt{Inv}[pX] \,\&\, \texttt{Inv}[\texttt{String}] \asymp \texttt{Inv}[X] \,\&\, Y && \text{By lemmas 3.4.7} \end{aligned}$$

It is not possible to reduce the solution further: for instance, it is incorrect to deduce $X \asymp \texttt{String}$. In other word, we have:

$$\texttt{Inv}[pX] \,\&\, \texttt{Inv}[\texttt{String}] \asymp \texttt{Inv}[X] \,\&\, Y \nVdash X \asymp \texttt{String}$$

Listing 6 shows a counter-example by picking $pX = X = \texttt{Int}$ and $Y = \texttt{Inv}[\texttt{String}]$. In terms of constraints, if we pick $\phi = \phi'[pX, X \mapsto \texttt{Int}; Y \mapsto \texttt{Inv}[\texttt{String}]]$ (for any arbitrary ϕ') and any arbitrary γ, the assignments ϕ, γ satisfy the equality in the antecedent but not the conclusion.

Example 4.3.3 (A solution for listing 7)**.** We claim a correct solution is $Y \asymp pY \curlywedge pX \asymp pY$.
We have:

$$\begin{aligned} C_G &\Vdash p : \texttt{P}[pX, pY, pF] \,\&\, \texttt{S}[X, F] \\ &\Vdash p : \texttt{S}[\texttt{Inv}[pX], F] \,\&\, \texttt{S}[X, F] && \text{By lemmas 3.4.12 and 3.4.4} \\ &\Vdash X \asymp \texttt{Inv}[pX] \curlywedge F \asymp pF && \text{By lemmas 3.4.7} \end{aligned}$$

Combining this with the C_G assumptions and lemma 3.4.14, we have:

$$\begin{aligned} C_G &\Vdash C_G \curlywedge [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, pY] \,\&\, \texttt{Inv}[pY] \;\preceq\; [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, Y] \,\&\, X \\ &\Vdash [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, pY] \,\&\, \texttt{Inv}[pY] \;\preceq\; [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, Y] \,\&\, \texttt{Inv}[pX] && \text{By lemmas 3.4.14} \end{aligned}$$

Let ϕ, γ be any assignments satisfying C_G. Then, they satisfy the conclusion of the above entailment as well.

By lemma 3.4.6, we have for all $A \in \mathcal{T}^{\text{cl}}$:

$$\phi[Z \mapsto A], \gamma \models \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \preceq \texttt{Inv2}[Z, Y] \mathbin{\&} \texttt{Inv}[pX]$$

where we have simplified the bounds satisfaction since these are trivial.

By lemma 3.4.12, we have:

$$\begin{aligned}\phi[Z \mapsto A], \gamma \models \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \preceq \texttt{Inv2}[Z, Y] \curlywedge{} \\ \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \preceq \texttt{Inv}[pX]\end{aligned}$$

For the first conjunct, by applying lemma 3.4.12 again, we have $\phi[Z \mapsto A], \gamma \models \texttt{Inv2}[Z, pY] \preceq \texttt{Inv2}[Z, Y]$ or $\phi[Z \mapsto A], \gamma \models \texttt{Inv}[pX] \preceq \texttt{Inv2}[Z, Y]$. The latter is unsatisfiable by lemma 3.4.5, so we have $\phi[Z \mapsto A], \gamma \models \texttt{Inv2}[Z, pY] \preceq \texttt{Inv2}[Z, Y]$ and as such $\phi[Z \mapsto A], \gamma \models pY \asymp Y$.

Using a similar reasoning, we get $\phi[Z \mapsto A], \gamma \models pY \asymp pX$ from the second conjunct.

Because Z is distinct from pX, pY and Y, we have by lemma 3.3.4 $\phi, \gamma \models Y \asymp pY \curlywedge pX \asymp pY$. All assignments ϕ, γ satisfying C_G satisfy $Y \asymp pY \curlywedge pX \asymp pY$ too, as such $C_G \Vdash Y \asymp pY \curlywedge pX \asymp pY$.

Chapter 5

A constraint simplifier

We dedicate this chapter to the presentation of a constraint simplification algorithm.

The goal of the constraint simplifier is to take a constraint C_G originating from the GADT constraints generation and rewrite it into a constraint C' such that $C_G \Vdash C'$ and that is as precise and simple as possible.

The main idea of the algorithm is to maintain a structure $\mathscr{K}$ representing the *knowledge* we have accumulated so far from decorticating the assumptions one by one. The structure $\mathscr{K}$ is essentially an organized set of accumulated constraints. Furthermore, when new information is assimilated in $\mathscr{K}$, the algorithm may unveil further knowledge arising from combining that information with $\mathscr{K}$.

This chapter is structured as follows. First, we present some preliminary notions needed to define the knowledge structure $\mathscr{K}$. Then, we properly introduce and define $\mathscr{K}$; we give it an interpretation in $\mathscr{C}$ by transforming it into a conjunctions of constraints. We then present the simplification algorithm.

5.1 Preamble

5.1.1 Equivalence classes

François Pottier and Didier Rémy [11] present the inference problem in DM as a constraint solving problem, which is based on HM(X) [6]. In particular, their constraint solver maintain a unification state establishing equality relationships between the encountered types. To do so, they (in particular) employ a union-find data structure [2].

While the setting of Pottier and Rémy constraint solver is of an equality-only free tree model [11], we will see that there are great benefits in building a structure that reserves a special treatments for types tied in an equality.

We are interested in maintaining (within $\mathscr{K}$) a partition of encountered types that are considered equal – in other words, we would like to build a data structure to represent equivalence classes (EC) of types. That is, if two types belong to the same EC, then they are considered equal. Types belonging to the same EC must have the same kind, and for higher-kinded types, the same variance as well.

To implement such a collection, we employ a union-find data structure [2]. For simplicity, we use an immutable variant where usual mutating operations such as `Union` or `MakeSet` are adapted to return an updated copy of the data structure.

We do not directly store types in the union-find structure: we store them in another structure that is not part of the union-find. Instead, the union-find data structure works with some opaque elements whose representation is not of an importance for our use case. We denote EC_H the set of these opaque elements (for **E**quivalence **C**lass **H**andle) and let the meta-variables $[a]$, $[b]$ denote elements of that set. When the context is clear, we refer to $[a]$, $[b]$ as "equivalence class" and drop the word "handle".

We list the operations we expect from a union-find data structure in appendix B.1.1. We employ the meta-variables $\mathcal{Q}$ to denote union-find structures.

For each equivalence class in $\mathcal{Q}$, we associate a set of types in a structure $\mathcal{M}$ to represent types that are equal to each other.

The figure below illustrates how the union-find data structure is essentially used as an intermediary to implement equivalence classes between types. In this example, $\mathcal{Q}$ is composed of three partitions with representatives $[a]$, $[b]$ and $[c]$. The partition $[b]$ in $\mathcal{Q}$ contains the EC handles $[c]$ and $[d]$ as well. Each representative is associated a set of types. For instance, $[a]$ is associated the set $\{T_1, T_2, T_3\}$, meaning that these types are considered equal.

Later on, we receive an update specifying that T_3 and S_2 are equal. A $\mathcal{Q}$-`Union` is performed on $[a]$ and $[b]$; the sets in $\mathcal{M}$ are merged as well.

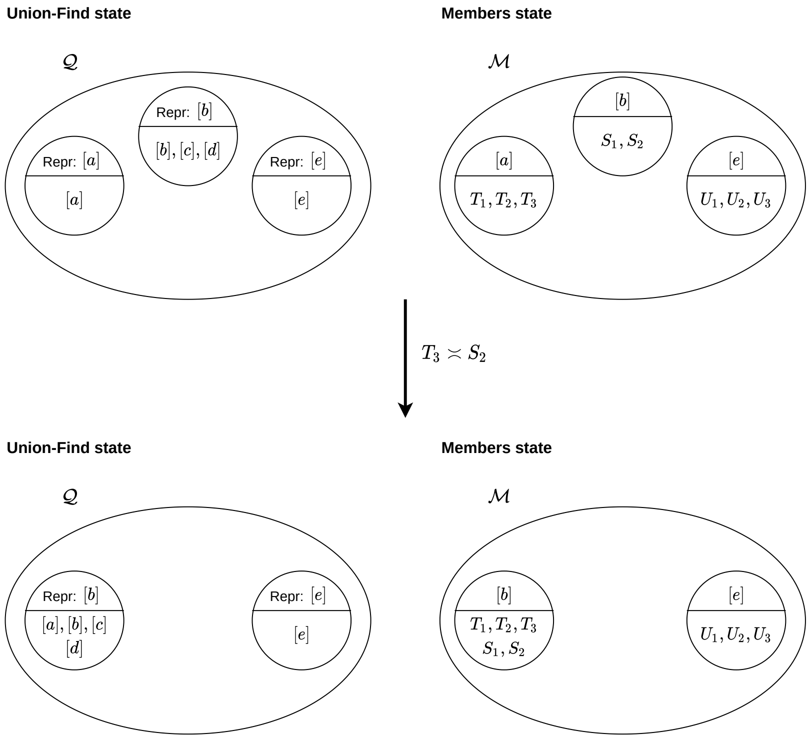


Figure 5.1 – Equivalence classes of types using a union-find data structure $\mathcal{Q}$.

One may wonder why we do not have $\mathcal{Q}$ and $\mathcal{M}$ fused into a single structure. We delay the answer to this question to the next section.

5.1.2 Types with equivalence classes handles

Equivalence classes handles become interesting when integrating them into types. For instance, suppose that we have a $\mathcal{Q}$ with two partitions, $[a]$ and $[b]$, such that $[c]$ belongs to the same EC (or partition) as $[a]$. Then, intuitively, the types $\texttt{MyTrait}\big[[a]\big]$ and $\texttt{MyTrait}\big[[c]\big]$ are equal since $[a]$ and $[c]$ both belong to the same partition. On the other hand, we may not say anything about $\texttt{MyTrait}\big[[a]\big]$ and $\texttt{MyTrait}\big[[b]\big]$.

For the sake of the example, suppose that $\mathcal{Q}$ is extended with two new partitions, $[d]$ and $[f]$, where $[d]$ is associated (in $\mathcal{M}$) the set of types $\{\texttt{MyTrait}\big[[a]\big]\}$ and $[f]$ the set of types $\{\texttt{MyTrait}\big[[b]\big]\}$. Suppose furthermore that $[a]$ and $[b]$ get merged. Then, the types $\texttt{MyTrait}\big[[a]\big]$ and $\texttt{MyTrait}\big[[b]\big]$ now become equal: we should then merge $[d]$ and $[f]$ into one partition in $\mathcal{Q}$ and $\mathcal{M}$.

We can now answer the question left in suspense in the last section. The reason behind having separate structures $\mathcal{Q}$ and $\mathcal{M}$ is to avoid having to go over all collected types and having to perform a substitution of types to some type representative. With equivalence classes, we get the substitution semantic “for free”.

We can extend equivalence classes of types to higher-kinded types as well. As an example, suppose that we have an EC $[a]$ with an associated set of types $\{[X]\,\texttt{=>>}\,\texttt{MyTrait}[\texttt{Inv}[X]], [X]\,\texttt{=>>}\,F[X]\}$. We write $[a]\left[S\right]$ to represent the type that is equal to $\texttt{MyTrait}[\texttt{Inv}[S]]$ or $F[S]$. We refer to $[a]\left[S\right]$ as an *applied equivalence class*. We naturally need to ensure that the types within $\mathcal{M}$ have all the same kind, otherwise, this definition is ill-formed.

We now have the necessary tools to introduce the following grammar to denote types with equivalence classes handles. It is loosely based on the grammar 2.2, except that there are no refinements; furthermore, intersection and union types must be in a disjunctive normal form (DNF).

$T ::=$	**Type**
$\top$	*top*
$\bot$	*bottom*
X	*type variable*
$\lvert\& \, T$	*DNF*
$Cls[\vec{T}]$	*concrete type con. app.*
$F[\vec{T}]$	*abstract type con. app.*
$[\vec{v}\vec{X} \lhd B]\,\texttt{=>>}\,T$	*HK abstraction*
$p.\texttt{type}$	*singleton type*
$p.Q$	*path-dependent type*
$p.Q[\vec{T}]$	*path-dependent type app.*
$[a]$	*equivalence class*
$[a]\left[\vec{T}\right]$	*applied equivalence class*

X, Y, Z, F	**Type variable**
a, f	**Field**
Q	**Type member**
Cls	**Class and trait**
B	**Bounds**
$p, q ::=$	**Path**
x	*variable*
$p.a$	*field selection*
$v ::=$	**Variance**
$+$	*covariance*
$-$	*contravariance*
$\pm$	*invariance*

Figure 5.2 – Syntax for types with equivalence classes handles

We employ the meta-variable $\mathcal{T}_{EC}$ to denote subsets of the set of all well-formed types generated from the above grammar. We similarly define $\mathcal{B}_{EC}$.

The structure $\mathscr{K}$ exclusively works with $\mathcal{T}_{EC}$ (except for one sub-structure that is used in a specific case). We do not need to worry about conformance of $\mathcal{T}_{EC}$ at this stage. This problem is tied with the interpretation of $\mathscr{K}$, which we examine in 5.2.2.

5.1.3 Type handles

We now refine the earlier description from 5.1.1 about the components $\mathcal{Q}$ and $\mathcal{M}$ of the structure $\mathscr{K}$.

The structure $\mathscr{K}$ will need to record information about types within equivalence classes. Instead of directly mapping types within an EC to some information concerning it, we map a *type handle* to that information. That way, if we need to update the type – for instance, we need to perform some explicit substitution – we do not have to update the information mapping (as long as the information about the new type is maintained). Then, the structure $\mathcal{M}$ stores type handles and not types. We store the mapping between the type handles and the actual types in the structure Θ.

The type handles are opaque elements. We use the meta-variable h to denote type handles and denote the set of these elements as T_H (for **T**ype **H**andle). We suppose it is possible to generate fresh type handles.

It is important to differentiate *equivalence classes handles* (set EC_H) from *type handles* (set T_H). Type handles are just indirection from actual types.

The figure below shows $\mathcal{Q}$, $\mathcal{M}$, Θ as well as some other structures cooperating together. In this example, we assume that S_1 needs to be substituted to `Bar` and U_3 to `Foo`. Assuming the properties about h_{S_1} and h_{U_3} remain, it is not needed to update the structures mentioning these handles.

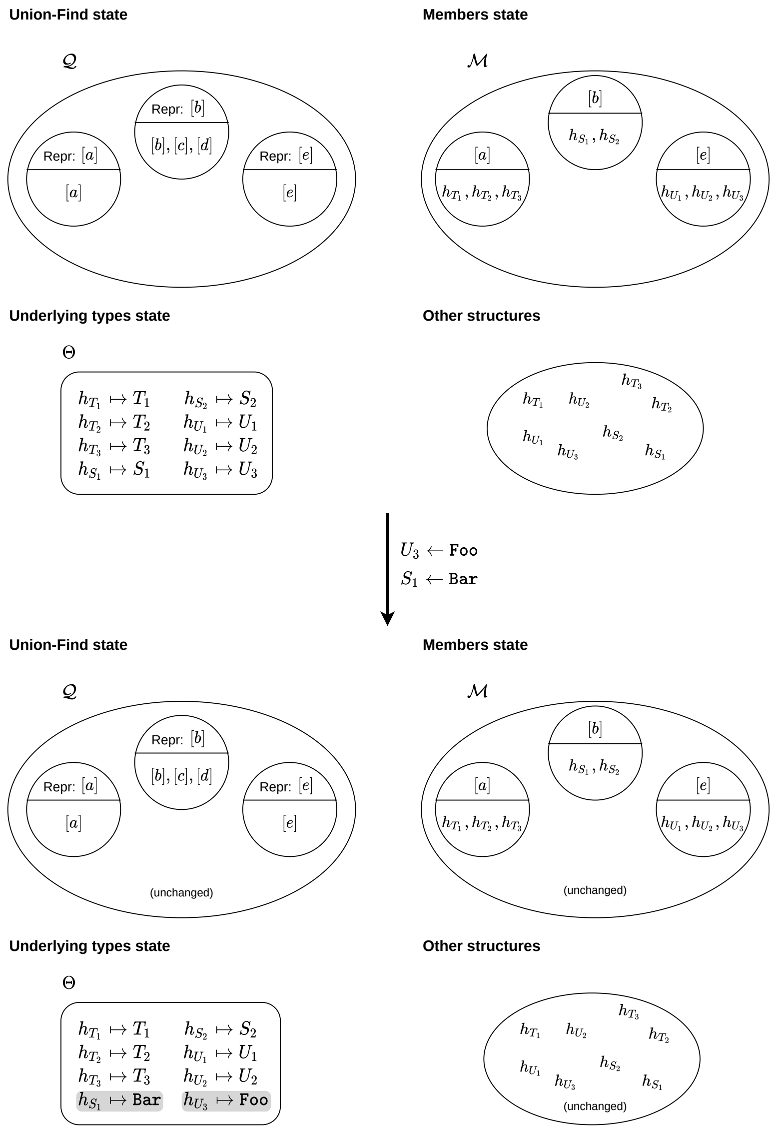


Figure 5.3 – Type handles and equivalence classes handles.

5.2 Knowledge structure $\mathscr{K}$

5.2.1 Definition

We define the knowledge structure $\mathscr{K} \triangleq (\mathcal{M}, \Theta, \mathcal{R}, \mathcal{D}, \mathcal{Q}, \mathcal{I}, T_R, G_{\preceq}, G_{EC}, G_{\mathcal{S}}, G_p)$ where:

- $\mathcal{M} : EC_H \rightharpoonup \mathscr{P}(T_H)$ gives the set of members belonging to a given EC_H.
- $\Theta : T_H \rightharpoonup \mathcal{T}_{EC}$ retrieves the underlying type of a given type handle T_H.
- $\mathcal{R} : T_H \rightharpoonup EC_H$ retrieves the EC handle of a type handle.
- $\mathcal{D} : EC_H \rightharpoonup T_H$ retrieves the type handle whose underlying type is determined, if it exists.
- $\mathcal{Q}$ is a union-find data-structure that works with the opaque equivalence classes EC_H.
- $\mathcal{I} : \mathcal{P} \rightharpoonup \mathcal{T}$ is employed to record constraints of the form $p : T$. It is the only structure involving plain $\mathcal{T}$ types and is solely used in a specific case.
- $T_R : EC_H \rightharpoonup T_H$ which associates for each equivalence class a type handle whose underlying type is a type not containing any EC_H. We refer to these types as *type representative*. This mapping allows to turn a $\mathcal{T}_{EC}$ type into a $\mathcal{T}$ type by substituting all EC_H into $\mathcal{T}$ types. This substitution is there to ease the correctness proof; in an implementation, we would not perform the substitution. We come back to the usage of T_R in the next section where we introduce the interpretation of $\mathscr{K}$ as a conjunctions of core constraints.
- The acyclic and *forward-free* graph $G_{\preceq} = (V_{\preceq}, E_{\preceq})$ where $V_{\preceq} \subseteq EC_H$ and $E_{\preceq} \subseteq V_{\preceq} \times V_{\preceq}$. This graph records the subtyping relations between the equivalence classes. The graph does not contain any *forward edge*: that is, for any chain $(u_1, u_2), (u_2, u_3), ..., (u_{n-1}, u_n) \in E_{\preceq}, n \geq 3$, we have that $(u_1, u_i) \notin E_{\preceq}$, $3 \leq i \leq n$.
- The bipartite graph $G_{EC} = (U_{EC}, V_{EC}, E_{EC}, L_{EC})$ where $U_{EC} \subseteq EC_H, V_{EC} \subseteq T_H, E_{EC} \subseteq U_{EC} \times V_{EC}$ and $L_{EC} : E_{EC} \to \{H, NH\}$. The purpose of G_{EC} is to record the appearances of equivalence classes in other equivalence classes' types. The labeling function L_{EC} specifies whether the EC appears in head (label H for **H**ead) or not (label NH for **N**on-**H**ead).

  For instance, if we have a handle h whose underlying type is $\texttt{MyTrait}\big[[a], [b]\big]$ & $[a]$ belonging to the EC $[c]$, we would have $[a], [b] \in U_{EC}$, $h \in V_{EC}$, $([a], h), ([b], h) \in E_{EC}$ and $L_{EC}([a]) = H, L_{EC}([b]) = NH$.
- The bipartite graph $G_{\mathcal{S}} = (U_{\mathcal{S}}, V_{\mathcal{S}}, E_{\mathcal{S}})$ where $U_{\mathcal{S}} \subseteq \mathcal{S}$, $V_{\mathcal{S}} \subseteq T_H$ and $E_{\mathcal{S}} \subseteq U_{\mathcal{S}} \times V_{\mathcal{S}}$. It records the appearance of abstract type constructors, traits, classes and type variable symbols in head positions within equivalence classes members.

  For instance, assuming that F is an abstract type constructor and that we have at our disposal the handle h with the underlying type $F[A]$, we would have $F \in U_{\mathcal{S}}$, $h \in V_{\mathcal{S}}$ and $(F, h) \in V_{\mathcal{S}}$.
- The bipartite graph $G_p = (U_p, V_p, E_p)$ where $U_p \subseteq \mathcal{P} \times \mathcal{S}$, $V_p \subseteq T_H$ and $E_p \subseteq U_p \times V_p$. It records the appearance of path-dependent types and singleton types in head positions within equivalence classes members.

5.2.2 Interpretation of $\mathscr{K}$

The knowledge structure $\mathscr{K}$ is nothing more than an (organized) accumulation of core constraints. Therefore, we can give $\mathscr{K}$ an interpretation by transforming it into a conjunction of constraints. In fact, in our proofs, we use the the constraints-view of $\mathscr{K}$ to show that functions returning an updated $\mathscr{K}$ yield a $\mathscr{K}'$ entailed by the original $\mathscr{K}$.

To transform $\mathscr{K}$ into a constraint, we have to be a bit careful: in $\mathscr{K}$, we are (essentially) establishing subtyping constraints within types in $\mathcal{T}_{EC}$ that may contain equivalence class handles. The constraint language $\mathscr{C}$ only treats $\mathcal{T}$ types. This is where T_R comes into play: this mapping associates for each equivalence class a $\mathcal{T}$ type [1]. The idea is to substitute each equivalence class handle with its associated type representative, allowing to turn a $\mathcal{T}_{EC}$ into a $\mathcal{T}$. In an implementation, we would not need to perform such substitution. Furthermore, the transformation of $\mathscr{K}$ into a constraint is purely used for proofs.

Before establishing the transformation of $\mathscr{K}$ into a constraint, we give below the definition of EC_H`-Subst` whose purpose is to create a partial mapping ς of $\mathcal{T}_{EC}$ into $\mathcal{T}$. EC_H`-SubstApply` defines the substitution of a $T : \mathcal{T}_{EC}$ type for a given ς, previously created with EC_H`-Subst`. The substitution is partial and may yield $\uparrow$ (undefined) for some $\mathcal{T}_{EC}$ types.

EC_H`-Subst` is rather simple: for each equivalence class (or partition) $[r]$, it maps all elements $[a]$ belonging to $[r]$ to the associated type representative of that partition.

EC_H`-SubstApply` proceeds by recursively substituting the given type T. The base cases are $[a]$ and $[a]\left[\vec{U}\right]$. When T is of the form $[a]$, we just need to perform a lookup in ς. Since ς is partial, $[a]$ may not defined in ς. In such case, the substitution is undefined as well. If T is of the form $[a]\left[\vec{A}\right]$, we similarly perform a lookup of $[a]$ in ς. The looked-up value needs to be higher-kinded and match the length of $\vec{A}$ to be defined. If it is the case, we recursively substitute $\vec{A}$ into $\vec{A}'$ and check that the kind of $\vec{A}'$ corresponds to the kind of the looked-up value. We then return the application of the higher-kinded abstraction to $\vec{A}'$.

Algorithm 1: Substitution of EC_H into $\mathcal{T}$

```
$EC_H$-Subst ($\mathscr{K}$)
    Precondition: $\mathscr{K}$-WellFormed($\mathscr{K}$)
    $\varsigma \leftarrow \emptyset$
    for $([a], [r]) \in \{([a], [r]) : [a] \in \mathcal{Q}$-MembersOf($\mathcal{Q}, [r]$)$,\ [r] \in \mathrm{dom}(\mathcal{M})\}$ do
        $\varsigma \leftarrow \varsigma[[a] \mapsto \Theta(T_R([r]))]$
    return $\varsigma$
$EC_H$-SubstApply ($\varsigma, T : \mathcal{T}_{EC}$)
    match $T$ :
        case $[a]$ :
            match $\varsigma([a])$ :
                Note: $S$ may be higher-kinded.
                case $S$ :
                    return $S$
                case $\uparrow$:
                    return $\uparrow$
        case $[a]\left[\vec{A}\right]$ :
            match $\varsigma([a])$ :
                Note: the bounds satisfcation must be ensured when forming $[a]\left[\vec{A}\right]$
                case $[\vec{v}\vec{X} \triangleleft B]$ =>> $S$ where $|\vec{X}| = |\vec{A}|$ :
                    $\vec{A}' \leftarrow EC_H$-SubstApply($\varsigma, \vec{A}$)
                    if $\vec{A}' = \uparrow \vee \mathrm{kind}(\vec{A}') \neq \mathrm{kind}(\vec{X})$ then
                        return $\uparrow$
                    else
                        return $[\vec{X} \mapsto \vec{A}']S$
                otherwise :
                    return $\uparrow$
        Other cases proceed like standard substitution and are omitted. If a recursive call is undefined, $\uparrow$ is
         returned.
```

---

[1] More precisely, a type handle whose underlying type is a $\mathcal{T}$ type.

When the context is unambiguous, we employ ς to mean $EC_H\texttt{-Subst}(\mathscr{K})$. We also abuse notation and write $\varsigma(T)$ to mean $EC_H\texttt{-SubstApply}(EC_H\texttt{-Subst}(\mathscr{K}), T)$.

Example 5.2.1 (Substitution of $\mathcal{T}_{EC}$ into $\mathcal{T}$, simply-kinded)**.** Let a structure $\mathscr{K}$ with two partitions $[a]$ and $[b]$ such that $[c]$ belongs to the partition of $[a]$. Suppose that $T_R([a]) = \texttt{MyTrait}[\texttt{Int}]$ and $T_R([b]) = \texttt{List}[X]$.

Then, $\varsigma \triangleq EC_H\texttt{-Subst}(\mathscr{K})$ is equal to $\big[[a], [c] \mapsto \texttt{MyTrait}[\texttt{Int}]; [b] \mapsto \texttt{List}[X]\big]$.

Furthermore:

$$\varsigma(\texttt{OtherTrait}\big[[b], [c]\big])) \triangleq EC_H\texttt{-SubstApply}(EC_H\texttt{-Subst}(\mathscr{K}), \texttt{OtherTrait}\big[[b], [c]\big])$$

is equal to $\texttt{OtherTrait}[\texttt{List}[X], \texttt{MyTrait}[\texttt{Int}]]$.

On the other hand, $\varsigma(\texttt{MyTrait}\big[[d]\big])$ is undefined since $[d]$ is not contained in $\mathscr{K}$ (with the assumption that $[d]$ is distinct from $[a]$, $[b]$ and $[c]$).

Example 5.2.2 (Substitution of $\mathcal{T}_{EC}$ into $\mathcal{T}$, higher-kinded)**.** Let a structure $\mathscr{K}$ with two partitions $[a]$ and $[b]$ such that $[c]$ and $[d]$ respectively belong to the partition of $[a]$ and $[b]$. Suppose that $T_R([a]) = [Z]\,\texttt{=>>}\,\texttt{ATrait}[Z, \texttt{Int}]$ and $T_R([b]) = \texttt{Option}[X]$.

Then, $\varsigma \triangleq EC_H\texttt{-Subst}(\mathscr{K})$ is equal to:

$$\big[[a], [c] \mapsto [Z]\,\texttt{=>>}\,\texttt{ATrait}[Z, \texttt{Int}]; [b], [d] \mapsto \texttt{Option}[X]\big]$$

Furthermore:

$$\varsigma([c]\big[\texttt{String} \mathbin{\&} [b]\big]) \triangleq EC_H\texttt{-SubstApply}(EC_H\texttt{-Subst}(\mathscr{K}), [c]\big[\texttt{String}[b]\big])$$

is equal to $\texttt{ATrait}[\texttt{String} \mathbin{\&} \texttt{Option}[X], \texttt{Int}]$.

The expressions $\varsigma([a]\big[X, [d]\big])$ and $\varsigma([a]\big[[a]\big])$ are undefined. The former does not respect the arity of $T_R([a])$ while the latter does not respect the kind of $T_R([a])$.

We now give the definition of the transformation of $\mathscr{K}$ into $\mathcal{C}$. The idea is to build constraints by employing the subtyping graph $G_{\preceq}$, equaling each member of an equivalence class to their type representative as well as restoring the typing constraints from $\mathcal{I}$. The typing constraints do not need to be transformed through ς because the involved types are elements of $\mathcal{T}$.

Algorithm 2: Transforming $\mathscr{K}$ into a $C : \mathcal{C}$

```
$\mathscr{K}$-to-$\mathcal{C}$ ($\mathscr{K}$) : $C : \mathcal{C}$
    $\varsigma \leftarrow EC_H\texttt{-Subst(}\mathscr{K}\texttt{)}$
    return $\bigcurlywedge\{\varsigma([x]) \preceq \varsigma([y]) : ([x], [y]) \in E_{\preceq}\} \curlywedge$
           $\bigcurlywedge\{\varsigma([r]) \asymp \varsigma(\Theta(h)) : ([r], \bar{h}) \in \mathcal{M}, h \in \bar{h}\} \curlywedge$
           $\bigcurlywedge\{p : T : (p, T) \in \mathcal{I}\}$
```

To reduce the burden of notation, we write $\mathscr{K}$ instead of $\mathscr{K}\texttt{-to-}\mathcal{C}(\mathscr{K})$ whenever we need to interpret $\mathscr{K}$ as a constraint.

5.2.3 Invariants

Because $\mathscr{K}$ plays a major part in the constraint simplification algorithm, it is primordial to establish some core properties.

There are three main invariants: $\mathscr{K}$`-WellFormed`, $\mathscr{K}$`-Valid` and $\mathcal{T}_{EC}$`-in-`Θ`-Inv`, defined below.

The first invariant, $\mathscr{K}$`-WellFormed`, ensures that all components within $\mathscr{K}$ agree with each other and are consistent. The $\mathscr{K}$`-Valid` invariant is based on $\mathscr{K}$`-WellFormed` and adds two properties. These two invariants are split for technical reasons: the properties stated by $\mathscr{K}$`-Valid` need a well-formed $\mathscr{K}$ to be meaningful.

The third invariant $\mathcal{T}_{EC}$`-in-`Θ`-Inv` concerns the $\mathcal{T}_{EC}$ stored in the mapping Θ of $\mathscr{K}$. These restrictions allow us to facilitate some of the function definitions or to rule out meaningless cases. We underline that these invariants are only applied to $\mathcal{T}_{EC}$ contained within Θ, not for all considered $\mathcal{T}_{EC}$.

We deem useful to discuss some of the rules of $\mathcal{T}_{EC}$`-in-`Θ`-Inv`, as these are seemingly arbitrary.

The rule (1) inhibits storing equivalence classes handles in Θ as-is. That is, we may not have an $[a]$ in $\mathrm{Im}(\Theta)$. On the other hand, types such as $[a]$ & $[b]$ or $[X]$ `=>>` $[a]$ are valid. The reason of this peculiar rule is best explained with an example. Suppose that we have a structure $\mathscr{K}$ containing three partitions $[a]$, $[b]$ and $[c]$ such that $[d]$ belongs to $[a]$. Suppose that the partition of $[a]$ has two associated types with handles h_1 and h_2. If we have $\Theta(h_1) = X$ and $\Theta(h_2) = T \mathbin{\&} [c]$, we should interpret X and $T \mathbin{\&} [c]$ as being equal. Furthermore, we can view all occurrences of $[a]$ and $[d]$ as being equal to X and $T \mathbin{\&} [c]$. Now, if we add an h_3 to $[a]$ with $\Theta(h_3) = [b]$, we should view $[b]$ as being equal to X and $T \mathbin{\&} [c]$, or, in other words, view $[b]$ as equal to $[a]$. Instead of storing $[b]$ in $\Theta(h_3)$, it is wiser to have $[a]$ and $[b]$ unified and merged, as we can benefit from having all associated types in $[b]$ equal to those in $[a]$. The rule (1) is here to ensure that we do not miss out on equivalence classes merges.

The rule (3) ensures that no free occurrences of type variables appear in the types contained in Θ (except if they appear as themselves). For instance, if a free type variable X is associated to $[a]$, it is preferable to have $[a] \mathbin{\&} T$ than $X \mathbin{\&} T$. Intuitively, if we get to know more about $[a]$, it is easier to search for the occurrences of $[a]$ than searching for the occurrences of all type variables associated to $[a]$.

The rule (6) is similar in essence to rule (3) but concerns applied abstract type constructor.

We finally arrive at the rule (7) which applies to non-trivial DNFs. First, all types in $\mathscr{K}$ may not contain a DNF in a non-head position (i.e., in an argument position). As we will see in 5.3.5.2, we can always extract DNFs into their own equivalence class, even if they capture bound type variables from an enclosing higher-kinded abstraction. This peculiar restriction facilitates the propagation of type determinacy of DNFs and within DNFs – which we discuss in further detail in 5.3.5.5. Second, we do not allow free type variables appearing in a head position within a DNF. We can always extract them into an equivalence class. This restriction has the same purpose as the head appearance of DNFs, namely to facilitate determinacy propagation.

$\mathscr{K}$-WellFormed ($\mathscr{K}$)

Type handle references (K-INV1):
$\mathrm{Im}(\mathcal{D}), \mathrm{Im}(T_R), V_{EC}, V_{\mathcal{S}}, V_p \subseteq \mathrm{dom}(\Theta) = \mathrm{dom}(\mathcal{R}) = \bigcup \mathrm{Im}(\mathcal{M})$

Equivalence class handles in sub-structures (K-INV2):
All EC handles are contained in $\mathcal{Q}$-`AllMembers(`$\mathcal{Q}$`)`:
$\mathrm{Im}(\mathcal{D}), V_{\preceq}, U_{EC} \subseteq \mathrm{dom}(\mathcal{M}) = \mathrm{Im}(\mathcal{R}) = \mathrm{dom}(T_R) \subseteq \mathcal{Q}\texttt{-AllMembers(}\mathcal{Q}\texttt{)}$
Furthermore, the referenced handles are the representatives:
$\forall [a] \in \mathrm{dom}(\mathcal{M}).\ \mathcal{Q}\texttt{-Find(}\mathcal{Q}, [a]\texttt{)} = [a]$

Equivalence class references in types (K-INV3):
$[a] \in T,\ T \in \mathrm{Im}(\Theta) \implies [a] \in \mathcal{Q}\texttt{-AllMembers(}\mathcal{Q}\texttt{)}$
In particular, we do not require $\mathcal{Q}\texttt{-Find(}\mathcal{Q}, [a]\texttt{)} = [a]$ for the $[a]$ contained in types. It would defeat the purpose of equivalence classes as we would have to perform substitution.

Non-empty ECs (K-INV4):
$\forall [a] \in \mathrm{dom}(\mathcal{M}).\ \mathcal{M}([a]) \neq \emptyset$

Substructures relationship (K-INV5):
$\forall (h, [a]) \in \mathcal{R}.\ h \in \mathcal{M}([a])$
$\forall ([a], h) \in \mathcal{D}.\ h \in \mathcal{M}([a])$
$\forall ([a], h) \in T_R.\ h \in \mathcal{M}([a])$

Types satisfy the $\mathcal{T}_{EC}$-in-Θ-Inv predicate (K-INV6):
$\forall T \in \mathrm{Im}(\Theta \restriction (\mathrm{dom}(\Theta) \setminus \mathrm{Im}(T_R))).\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv(}T\texttt{)}$

Form of T_R types (K-INV7):
Representative do not contain EC_H:
$\forall h \in \mathrm{Im}(T_R).\ \Theta(h) \in \mathcal{T}$
(These are not necessarily closed; they may contain free type or term variables)

Same kind within an EC (K-INV8):
$\forall \bar{h} \in \mathrm{Im}(\mathcal{M}),\ h_1, h_2 \in \bar{h}.\ \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, \Theta(h_1)\texttt{)} = \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, \Theta(h_2)\texttt{)}$

No determinacy for T_R (K-INV9):
$\mathrm{Im}(T_R) \mathbin{\#} \mathrm{Im}(\mathcal{D})$

Validity of $G_{\preceq}$ (K-INV10):
Well-formedness:
$E_{\preceq} \subseteq V_{\preceq} \times V_{\preceq}$
Acyclicity and forward-free:
$G_{\preceq}$ is acyclic and forward-free, that is, for any chain $(u_1, u_2), (u_2, u_3), ..., (u_{n-1}, u_n) \in E_{\preceq}$, $n \geq 3$, we have that $(u_1, u_i) \notin E_{\preceq}$, $3 \leq i \leq n$.
Same kind for EC tied in an inequality:
$\forall ([a], [b]) \in E_{\preceq}.\ \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, [a]\texttt{)} = \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, [b]\texttt{)}$

Validity of G_{EC} (K-INV11):
Well-formedness:
$E_{EC} \subseteq U_{EC} \times V_{EC}$ and $\mathrm{dom}(L_{EC}) = E_{EC}$

Validity of $G_{\mathcal{S}}$ (K-INV12):
Well-formedness:
$E_{\mathcal{S}} \subseteq U_{\mathcal{S}} \times V_{\mathcal{S}}$
Appearance of symbols in head:
$\forall (sym, h) \in E_{\mathcal{S}}.\ \mathcal{T}_{EC}\texttt{-InHead(}\mathcal{Q}, sym, \Theta(h)\texttt{)}$

Validity of G_p (K-INV13):
Well-formedness:
$E_p \subseteq U_p \times V_p$
Appearance of path-dependent types in head:
$\forall ((p, ty), h) \in E_{\mathcal{S}}.\ \mathcal{T}_{EC}\texttt{-InHead(}\mathcal{Q}, p.ty, \Theta(h)\texttt{)}$

```
$\mathcal{K}$-Valid ($\mathcal{K}$)
  Well-formedness:
    $\mathcal{K}$-WellFormed($\mathcal{K}$)
  Types marked as determined are determined (K-INV14):
    $\forall h \in \mathrm{Im}(\mathcal{D}).\ \mathcal{T}_{EC}$-IsDet($\mathcal{K}, \Theta(h)$)
  $\varsigma$ is defined on all contained types (K-INV15):
    $\forall T \in \mathrm{Im}(\Theta).\ \varsigma(T)\downarrow$
```

```
$\mathcal{T}_{EC}$-in-$\Theta$-Inv  ($T : \mathcal{T}_{EC}$, inHead : $\mathbb{B}$, boundTyVars : $\mathscr{P}(\mathcal{V}_X)$)
  Remark: Default arguments: inHead ← true, boundTyVars ← $\emptyset$
  match $T$ :
(1)   case $[a]$ :
        inHead $\implies$ boundTyVars $\neq \emptyset$
(2)   case $[a][\vec{S}]$ :
        $\forall S \in \bar{S}.\ \mathcal{T}_{EC}$-in-$\Theta$-Inv($S$, false, boundTyVars)
(3)   case $X$ :
        $X \notin$ boundTyVars $\implies$ inHead
(4)   case $p.Q$ :
        true
(5)   case $Cls[\vec{S}]$ or $p.F[\vec{S}]$ :
        $\forall S \in \bar{S}.\ \mathcal{T}_{EC}$-in-$\Theta$-Inv($S$, false, boundTyVars)
(6)   case $F[\vec{S}]$ :
        ($\forall S \in \bar{S}.\ \mathcal{T}_{EC}$-in-$\Theta$-Inv($S$, false, boundTyVars)) $\wedge$
            $F \notin$ boundTyVars $\implies$ inHead
(7)   case $|_i^n \&_j^{m_i} T_{i,j}$ :
        inHead $\wedge$
         $\forall T_{i,j}$. match $T_{i,j}$ :
          case $[a]$ :
            true
          We do not want free occurrences of $X$ to appear in a DNF; we would like them to be extracted
            into their own EC.
          case $X$ where $X \notin$ boundTyVars :
            false
          Same applies for $F$.
          case $F[\vec{S}]$ where $F \notin$ boundTyVars :
            false
          Note: An HK appearing is an intersection type is ill-formed and is not a $\mathcal{T}_{EC}$ as such.
          otherwise :
            $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T_{i,j}$, inHead, boundTyVars)
(8)   case $[\vec{v}\vec{X} \lhd B]$ =>> $S$ :
        $\mathrm{dom}(B) = \bar{X}\ \wedge$
            $\mathcal{B}_{EC}$-in-$\Theta$-Inv($B$, boundTyVars) $\wedge$
            $\mathcal{T}_{EC}$-in-$\Theta$-Inv($S$, inHead, boundTyVars $\cup\ \bar{X}$)
(9)   otherwise :
        false
$\mathcal{B}_{EC}$-in-$\Theta$-Inv  ($B : \mathcal{B}_{EC}$, boundTyVars : $\mathscr{P}(\mathcal{V}_X)$)
  $\mathrm{dom}(B)$ # boundTyVars $\wedge$
      $\forall (L, U) \in \mathrm{Im}(B)$.
          $\mathcal{T}_{EC}$-in-$\Theta$-Inv($L$, true, boundTyVars $\cup\ \mathrm{dom}(B)$) $\wedge$
          $\mathcal{T}_{EC}$-in-$\Theta$-Inv($U$, true, boundTyVars $\cup\ \mathrm{dom}(B)$)
```

5.3 Constraint simplification

We first present a high-level overview of the constraint simplifier. Next, we quickly discuss some conventions used in the definition of the algorithm. Finally, we present the core parts of the simplifier.

5.3.1 High-level overview

The algorithm is composed of three principal parts.

The entry point is $\mathcal{C}$`-Simplify`, whose goal is to rewrite C_G. It maintains the accumulated knowledge $\mathscr{K}$ and the set of core constraints that need to be processed. Initially, $\mathscr{K}$ is empty (i.e. `true`), and the set is initialized with the GADT assumptions C_G. $\mathcal{C}$`-Simplify` proceeds by picking a constraint in the set and repeatedly calls the *deduction* and *compaction* phase until the set is empty.

The *deduction* phase, `Deduction`, is given the current knowledge $\mathscr{K}$ and a core constraint $C_\#$. It reduces $C_\#$ into conjunctions of constraints $\bigwedge_i D_i$ where the D_i are of the form `true`, `false`, $S \preceq T$ – with S and T free of any refinement – or $p : U$. Sometimes, the deduction phase does not have enough information to completely reduce a constraint. In such cases, it yields the constraint back. As we have seen in section 3.5, doing so may allow further accumulated knowledge to "unblock" the constraint and have it reduced.

The *compaction* phase `Compact` combines $\mathscr{K}$ and $\bigwedge_i D_i$ together into a $\mathscr{K}'$, one constraint at a time. This phase may yield back new constraints coming from merging $\mathscr{K}$ and $\bigwedge_i D_i$. If so, these are added to the set of constraints. The compaction phase is itself composed of four sub-phases that we do not need to discuss for this overview.

We now give three examples of how $\mathcal{C}$`-Simplify` would operate if we were to give the GADT assumptions for listings 5 through 7.

Example 5.3.1.1 (Workflow for listing 5)**.** We remind ourselves we start with the following C_G:

$$p : \texttt{P} \,\&\, \texttt{S}[X, Y]$$

A run of $\mathcal{C}$`-Simplify` would be:

1. The set of constraints is initialized to $\{p : \texttt{P} \,\&\, \texttt{S}[X, Y]\}$ and $\mathscr{K}$ to $\mathscr{K}$`-New`.
2. The constraint $p : \texttt{P} \,\&\, \texttt{S}[X, Y]$ is picked:
   (a) `Deduction` simplifies the picked constraint into $X \asymp \texttt{Int}$.
   (b) The compaction phase assimilate $X \asymp \texttt{Int}$ into $\mathscr{K}$, yielding an updated $\mathscr{K}$. It does not yield any additional constraint.
3. All constraints in the maintained set have been processed. We return the maintained $\mathscr{K}$ whose interpretation is $X \asymp \texttt{Int}$.

Example 5.3.1.2 (Workflow for listing 6)**.** We are given the following C_G:

$$p : \texttt{P}[pX] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, Y]$$

A run of $\mathcal{C}$`-Simplify` would be:

1. The set of constraints is initialized to $\{p : \texttt{P}[pX] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, Y]\}$ and $\mathscr{K}$ to $\mathscr{K}$`-New`.
2. The constraint $p : \texttt{P}[pX] \,\&\, \texttt{S}[\texttt{Inv}[X] \,\&\, Y]$ is picked:
   (a) `Deduction` simplifies the picked constraint into $\texttt{Inv}[pX] \,\&\, \texttt{Inv}[\texttt{String}] \asymp \texttt{Inv}[X] \,\&\, Y$ but cannot reduce it further.
   (b) The compaction phase assimilate the deduced constraint into $\mathscr{K}$, yielding an updated $\mathscr{K}$. No additional constraints result from the compaction.
3. All constraints in the maintained set have been processed. We return the maintained $\mathscr{K}$ whose interpretation is $\texttt{Inv}[pX] \,\&\, \texttt{Inv}[\texttt{String}] \asymp \texttt{Inv}[X] \,\&\, Y$.

Example 5.3.1.3 (Workflow for listing 7, run (i))**.** We are given the following C_G:

$$\begin{aligned}
&p : \texttt{P}[pX, pY, pF] \mathbin{\&} \texttt{S}[X, F] \curlywedge \\
&[Z] \,\texttt{=>>}\, F[Z] \;\preceq\; [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, Y] \mathbin{\&} X \curlywedge \\
&[Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \;\preceq\; [Z] \,\texttt{=>>}\, pF[Z]
\end{aligned}$$

To help with readability, we name the conjuncts C_1, C_2 and C_3 respectively.
A run of $\mathcal{C}$`-Simplify` where C_1 is picked after C_2 and C_3 would be:

1. The set of constraints is initialized to $\{C_1, C_2, C_3\}$ and $\mathscr{K}$ to $\mathscr{K}$`-New`.
2. The constraint C_2 is picked:
   (a) The deduction phase cannot deduce any useful information out of it with the current knowledge. Therefore, this constraint is given back. Future knowledge may help extracting information from C_2.
   (b) The compaction phase assimilates C_2 into $\mathscr{K}$ and does not create new constraints.
3. The constraint C_3 is picked:
   (a) The deduction phase cannot reduce C_3 and gives it back.
   (b) The compaction phase assimilates C_3 into $\mathscr{K}$. No new constraints are created.
4. The constraint C_1 is picked:
   (a) The deduction phase reduces C_1 into $X \asymp \texttt{Inv}[pX] \curlywedge F \asymp pF$
   (b) The compaction phase assimilates $X \asymp \texttt{Inv}[pX]$ and $F \asymp pF$ into $\mathscr{K}$ sequentially. It yields the constraint:

   $$C_4 \triangleq [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, pY] \mathbin{\&} \texttt{Inv}[pY] \;\preceq\; [Z] \,\texttt{=>>}\, \texttt{Inv2}[Z, Y] \mathbin{\&} \texttt{Inv}[pX]$$

   (c) C_4 is added to the set of constraints to process.
5. The constraint C_4 is picked:
   (a) The deduction phase reduces C_4 into $Y \asymp pY \curlywedge pX \asymp pY$.
   (b) $Y \asymp pY$ and $pX \asymp pY$ are assimilated into $\mathscr{K}$. The compaction phase does not produce new constraints.
6. All constraints in the maintained set have been processed. We returned the maintained $\mathscr{K}$ whose interpretation is $Y \asymp pY \curlywedge pX \asymp pY$.

Example 5.3.1.4 (Workflow for listing 7, run (ii))**.** If we pick C_1, C_2 and C_3 in that order, we would instead get:

1. The constraint C_1 is picked:
   (a) The deduction phase reduces C_1 into $X \asymp \texttt{Inv}[pX] \curlywedge F \asymp pF$
   (b) The compaction phase assimilates $X \asymp \texttt{Inv}[pX]$ and $F \asymp pF$ into $\mathscr{K}$ sequentially. No further constraints are produced.
2. The constraint C_2 is picked:
   (a) The deduction phase cannot deduce any useful information out of it with the current knowledge and gives it back.
   (b) The compaction phase assimilates C_2 into $\mathscr{K}$ and does not create new constraints.
3. The constraint C_3 is picked:
   (a) The deduction phase cannot reduce C_3 and gives it back.

   (b) The compaction phase assimilates C_3 into $\mathscr{K}$ and generate the same constraint C_4 as the previous run.

4. The constraint C_4 is picked: the output is similar to the previous run.

5.3.2 Conventions

We discuss some assumptions and conventions used when describing the different functions. We think these are all reasonable but state them nonetheless to reduce ambiguity.

We start with type pattern matching. Given case pattern such as $Cls[\vec{S}]$, only (applied) traits and classes may match the case. Furthermore, we assume that nullary type constructors (e.g. `Int`) match the case, with $\vec{S}$ being the empty vector.

All applied type constructors (including bound and path-dependent) match the pattern case $TyCon[\vec{S}]$.

Example.

```
match $T$ :
    $\texttt{String}$ matches the pattern.
    $\texttt{FooTrait}[X, \texttt{Int}]$ matches the pattern with $\vec{S} = (X, \texttt{Int})$.
    $F[X]$ does not match the pattern (assuming $F$ is abstract)
    case $Cls[\vec{S}]$ :
        (...)
```

We assume that appropriate implicit η-expansion and uncurrying are performed to have the body of the higher-kinded abstractions be of simple kind. Furthermore, we assume that implicit α-renaming is performed on the scrutinee to match a given pattern.

Example.

```
match $(S, T)$ :
    The pair $([+Y <: \texttt{Foo}] \texttt{=>>} A_1, [+Z >: \texttt{Bar}] \texttt{=>>} A_2)$ matches the pattern.
    The pair $([-X] \texttt{=>>} A_1, [+X] \texttt{=>>} A_2)$ does not match the pattern due to the sign difference.
    The pair $([X <: \texttt{Foo}] \texttt{=>>} [-Y] \texttt{=>>} A_1, [Z >: \texttt{Bar}] \texttt{=>>} [-W <: \texttt{Qux}] \texttt{=>>} A_2)$ matches the pattern with:
        $\vec{v} = (\pm, -)$,
        $B_1 = \{(X, (\bot, \texttt{Foo})), (Y, (\bot, \top))\}$,
        $B_2 = \{(X, (\texttt{Bar}, \top)), (Y, (\bot, \texttt{Qux}))\}$,
        $U = A_1$,
        $V = [Z \mapsto X, W \mapsto Y]A_2$
    case $([\vec{v}\vec{X} \triangleleft B_1] \texttt{=>>} U, [\vec{v}\vec{X} \triangleleft B_2] \texttt{=>>} V)$ :
        (...)
```

We assume that the pattern $\left|_i^n \&_j^{m_i} T_{i,j}\right.$ may only be matched by non-trivial DNF types. No assumptions on the order of the $T_{i,j}$ are needed.

We conclude by stating a final assumption that is not needed for correctness but is desired to help the process of constraint simplification. Given two types S and T, we assume that the $\&$ (and similarly $|$) constructor is "smart": if S is $\bot$ or $\top$, then $S \,\&\, T$ is simplified to $\bot$ and T respectively (and analogously for $|$). Furthermore, $\&$ and $|$ coalesce the arguments of type constructors as follows. For a covariant (resp. contravariant) type constructor $TyCon$, $TyCon[A] \,\&\, TyCon[B]$ is reduced to $TyCon[A \,\&\, B]$ (resp. $TyCon[A \mid B]$). The same applies for $|$, with the result swapped. The coalescing behavior is extend to type constructors of arity greater than one as well; however, the coalescing may not be possible if there is at least one invariant position.

5.3.3 Entry point: $\mathcal{C}$-Simplify

The entry point of the constraint simplification process is $\mathcal{C}$`-Simplify`, defined in algorithm 3. The body of the function is quite simple and closely follows the description from 5.3.1. We maintain a set `cstrts` of core constraints that we wish to simplify, as well as the knowledge structure $\mathscr{K}$.

We pick a constraint from the set and, given our accumulated knowledge $\mathscr{K}$, try to simplify it by passing it through `Deduction`. It results in a conjunctions of potentially simpler constraints. We then compact and assimilate these constraints into $\mathscr{K}$. The compaction phase may give back some new constraints, which are added to the set of maintained constraint.

There are two notable differences from the description in 5.3.1. The first is that we allow $\mathcal{C}$`-Simplify` to be passed an initial $\mathscr{K}$. One may create an initial $\mathscr{K}$ (whose interpretation is just `true`) with $\mathscr{K}$`-New`.

The second has to do with the compaction phase. The high-level overview states that the compaction phases assimilates any core constraint into $\mathscr{K}$. It is not quite true, as we only feed `Compact` subtyping constraint of the form $S \preceq T$. Path typing constraints such as $p : T$ are processed within $\mathcal{C}$`-Simplify` by just adding them into $\mathcal{I}$. Finally, `true` and `false` are trivially handled without involving $\mathscr{K}$.

The $\mathcal{C}$`-Simplify` function guarantees that the returned $\mathscr{K}$ is entailed by the conjunction of the original $\mathscr{K}$ and the GADT assumptions C_G. We prove partial correctness of $\mathcal{C}$`-Simplify` in appendix A.3. We do not provide a formal proof of $\mathcal{C}$`-Simplify` termination, but sketch one in appendix A.10.

Algorithm 3: Constraints simplifier

```
     C-Simplify $(\mathscr{K}, C_G = \bigcurlywedge_i^n C_i) : \mathscr{K} \uplus \{\texttt{false}\}$
         Input: An initial $\mathscr{K}$ and a conjunction of core constraints
         Output: A conjunction of core constraints entailed by the original ones. The value false denotes that
           the given constraints are unsatisfiable.
         Precondition: $\mathscr{K}$-Valid($\mathscr{K}$)
         Precondition: The $C_i$ are core constraints.
         Postcondition: Entailment of the result:
           • If the result is not false:
                 $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}'$
           • If the result is false:
                 $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \texttt{false}$

         cstrts ← $\{C_i, 1 \leq i \leq n\}$
         $\mathscr{K}^{(1)} \leftarrow \mathscr{K}$
         Loop Invariant: $\mathscr{K}$-Valid($\mathscr{K}^{(1)}$)
         Loop Invariant: $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge$ cstrts
(1)      while $\exists C \in$ cstrts do
(1a)         if $C =$ false then
                 return false
(1b)         if $C =$ true then
                 cstrts ← cstrts \ $\{C\}$
                 continue
(1c)         $\bigcurlywedge_j^m D_j \leftarrow$ Deduction($\mathscr{K}^{(1)}$, $C$)
             n stands for next. This little dance with indices eases a bit the analysis because we need to reference
               "old" $\mathscr{K}$'s and "new" $\mathscr{K}$'s.
             $\mathscr{K}^{(n)} \leftarrow \mathscr{K}^{(1)}$
             cstrts$^{(n)}$ ← cstrts
```

```
              **Loop Invariant:** $\mathscr{K}$-`Valid`$(\mathscr{K}^{(n)})$
              **Loop Invariant:** $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(n)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(n)}$
(1d)          for $j \leftarrow 1$ to $m$ do
                  match $D_j$ :
(1d.i)                case `true` :
                          $\mathscr{K}' \leftarrow \mathscr{K}^{(n)}$
                          $\mathsf{cstrts}' \leftarrow \emptyset$
(1d.ii)               case `false` :
                          return `false`
(1d.iii)              case $S_j \preceq T_j$ :
                          $(\mathscr{K}', \mathsf{cstrts}') \leftarrow$ `Compact`$(\mathscr{K}^{(n)}, S_j, T_j)$
                      Here, we could do better by propagating the inhabitation to the upper bounds of $T_j$. To avoid
                        duplicating inhabitation constraints, we would have to store a set of types for which the
                        inhabitation constraints have already been examined.
(1d.iv)               case $p : T_j$ :
                          if $p \in \mathrm{dom}(\mathcal{I})$ then
                              $\mathcal{I}' \leftarrow \mathcal{I}[p \mapsto T_j]$
                          else
                              $\mathcal{I}' \leftarrow \mathcal{I}[p \mapsto T_j \mathbin{\&} \mathcal{I}(p)]$
                          $\mathscr{K}' \leftarrow \mathscr{K}[\mathcal{I} \mapsto \mathcal{I}']$
                          $\mathsf{cstrts}' \leftarrow \emptyset$
                  Update and continue.
                  $\mathscr{K}^{(n)} \leftarrow \mathscr{K}'$
                  $\mathsf{cstrts}^{(n)} \leftarrow \mathsf{cstrts}^{(n)} \cup \mathsf{cstrts}'$
              Update and continue.
              $\mathscr{K}^{(1)} \leftarrow \mathscr{K}^{(n)}$
              $\mathsf{cstrts} \leftarrow \mathsf{cstrts}^{(n)} \setminus \{C\}$
(2)       return $\mathscr{K}^{(1)}$
```

5.3.4 The deduction phase

Given a valid $\mathscr{K}$ and a core constraint $C_{\#}$, the goal of the deduction phase is to reduce $C_{\#}$ into a conjunction of core constraints $\curlywedge_i^n D_i$ that are entailed by $\mathscr{K}$ and $C_{\#}$. Furthermore, the subtyping constraints must not contain any refinements because we do not support them in the latter phases.

As briefly discussed in 5.3.1, the deduction may return the original constraint (provided its satisfies the above requirements) if it is irreducible at the current iteration but is nonetheless worthwhile to keep.

The function `Deduction` (defined in algorithm 4) is the entry-point of the deduction phase. It is just a wrapper performing a case analysis on the given (core) constraint $C_{\#}$. If the constraint $C_{\#}$ is a subtyping constraint, the work is delegated to `DeductionIneq`, which we describe afterwards.

Otherwise, the constraint is a path typing of the form $p : T$. The path p can also be a term variable, such as x. We then extract all fields accessible from p and hand them over to `DeductionTypedPath`. For instance, if we consider the following definitions:

$$\begin{aligned}
&\texttt{case class InvCov}[A, +B](a : A, b : B)\\
&\texttt{case class Foo}[A, B, C, D](a : \texttt{Bar}, b : \texttt{InvCov}[A, B]\ \&\ \texttt{InvCov}[C, D])\\
&\texttt{case class Bar}(a : \texttt{List}[\texttt{Int}])
\end{aligned}$$

and that we are given p : `Foo`$[X, Y, Z, W]$, we would pass to `DeductionTypedPath` the following constraints[2]:

$$\begin{aligned}
&p.a : \texttt{Bar}\\
&p.b : \texttt{InvCov}[X, Y]\ \&\ \texttt{InvCov}[Z, W]\\
&p.a.a : \texttt{List}[\texttt{Int}]
\end{aligned}$$

Algorithm 4: Deduction entry-point

```
Deduction ($\mathscr{K}$, $C_{\#}$) : $\curlywedge_i^n D_i$
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$)
    Precondition: $C_{\#}$ is a core constraint that is not true or false
    Postcondition: $\mathscr{K} \curlywedge C_{\#} \Vdash \curlywedge_i^n D_i$ where the $D_i$ are of the form true, false, $S \preceq T$
        with $S$ and $T$ free of any refinement or $p : U$.
    match $C$ :
(1)     case $p : T$ :
            $D \leftarrow$ true
            for $(q, S) \in \mathcal{T}$-InhabitedTypes($p, T$) do
                $D \leftarrow D \curlywedge$ DeductionTypedPath($\mathscr{K}, q, S$)
            return $D$
(2)     case $T_1 \preceq T_2$ :
            return DeductionIneq($\mathscr{K}, T_1, T_2$)
```

`DeductionTypedPath` (algorithm 5) reduces a path typing by essentially equating the types of intersections of traits (or classes) at invariant positions. Resuming the previous example, `DeductionTypedPath` reduces $p.b$: `InvCov`$[X, Y]$ & `InvCov`$[Z, W]$ to $X \asymp Z$ because X and Z appear at an invariant position of the same class `InvCov`. Y and W are untouched because they appear in a covariant position [3].

For the other path typings, however, nothing is deduced because their types are not intersection types.

---

[2] Note that we do not return fields from intersection of types, such as $p.b.a : A$ or $p.b.a : B$. It is unclear whether it is always sound to do so.

[3] This reduction stems from lemma 3.4.7 which is based on rule (PATH-&)

Algorithm 5: Reduction of a path constraint into simpler constraints

```
DeductionTypedPath ($\mathscr{K}$, $p : \mathcal{P}$, $T : \mathcal{T}$) : $\curlywedge_i^n D_i$
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$)
    Postcondition: $\mathscr{K} \curlywedge p : T \Vdash \curlywedge_i^n D_i$ where the $D_i$ are of the form true, false, $S_1 \preceq S_2$
        with $S_1$ and $S_2$ free of any refinement or $q : U$.
    Set of types that are common to all disjunctions in a DNF. For example, the set of common types in
     $(T \,\&\, S \,\&\, U) \mid (T \,\&\, S \,\&\, V)$ is $\{T, S\}$
    commonTys ← $\mathcal{T}$-CommonTypes($\mathcal{T}_{EC}$-SimplifyDNF($\mathscr{K}, T$))
(1) if |commonTys| $\leq 1$ then
        Nothing to deduce.
        return $p : T$
    $D \leftarrow p : T$
(2) for $(S, U) \in \{(S, U) : S, U \in$ commonTys$, S \neq U\}$ do
        match $(S, U)$ :
(2a)        case $(Cls[\vec{A}], Cls[\vec{B}])$ :
                Then, $\vec{A}$ and $\vec{B}$ must agree on invariant positions. $\vec{v}$ is the variance sign vector of $Cls$
                for $j \in \{j : v_j = \pm,\ 1 \leq j \leq |\vec{v}|\}$ do
                    $D \leftarrow D \curlywedge$ DeductionIneq($\mathscr{K}, A_j, B_j$) $\curlywedge$ DeductionIneq($\mathscr{K}, B_j, A_j$)
(2b)        case $(Cls_1[\vec{A}], Cls_2[\vec{B}])$ where $Cls_1$ extends $Cls_2$ :
                Then, $Cls_1$ extends $Cls_2$ $N \geq 1$ times through $\boldsymbol{\sigma}_1, ..., \boldsymbol{\sigma}_N$ such that:
                $Cls_1[\vec{A}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{A})] \preceq Cls_2[\vec{B}]$.
                for $i \leftarrow 1$ to $N$, $j \in \{j : v_j = \pm,\ 1 \leq j \leq |\vec{v}|\}$ do
                    $D \leftarrow D \curlywedge$ DeductionIneq($\mathscr{K}, \boldsymbol{\sigma}_i(\vec{A})_j, B_j$) $\curlywedge$ DeductionIneq($\mathscr{K}, B_j, \boldsymbol{\sigma}_i(\vec{A})_j$)
            otherwise :
                pass
    return $D$
```

The reduction of a constraint subtyping requires a bit more work. This task is granted to `DeductionIneq`, defined in algorithm 6. We give it an overview by going over some of its interesting cases.

Case (1).

When the compared types are closed, we simply leverage $\mathcal{T}$-`IsSubtype`. The subtyping can be incomplete (and it is expected to be so), but we require uncertainty to be denoted as `undet`.

Case (6).

We must first verify whether B_2 is "bigger" than B_1 by checking if it subsumes B_1. In case of uncertainty, we return the constraint unchanged only if it the bodies do not contain any refinement. It is possible that some further accumulations of knowledge render the constraint reducible.

Then, we recursively perform a deduction on the body of the abstractions. For each D_i, there are four possible cases, of which two are of interest. The first case, (6c), adds the simplified conjunct to the returned solution whenever the bounds of B_1 are satisfied under $\mathscr{K}$ (that is, entailed by $\mathscr{K}$) and whenever the type variables $\bar{X}$ do not appear in the reduced constraint. For example, suppose $\mathscr{K} \Vdash U \preceq V$ holds and that we would like to simplify:

$$[X >: U <: V] \texttt{=>>} \texttt{InvCovTrait}[X, Y] \preceq [X >: U <: V] \texttt{=>>} \texttt{InvCovTrait}[X, Z]$$

Then, the recursive call would yield $Y \preceq Z$ and we would return the constraint as it is.

If we cannot prove a subtyping relationship between the lower and upper bounds of B_1 (with the assumption that $\mathscr{K}$ holds), we may not return the simplified constraint as-is because we could introduce subtyping relationship that do not hold.

The case (6d) is here to avoid wasting potentially useful information by introducing the assumptions on the subtyping relationship back, by just constructing an abstraction on top of the reduction. It also covers the case where the reduced constraint contains a bound type variable.

Case (8).

Reducing a subtyping constraint between two refinements closely follows the results from lemmas 3.4.9 and 3.4.10. At (8a), we ensure that all members of R_2 are included in R_1 as well. Then, at (8b), we check in $\mathcal{I}$ if we have previously recorded a constraint witnessing the inhabitation of R_1. If not, we give up by returning `true` [4].

Deducing new constraints from the type of the fields and the bounds of the type members is quite straightforward (lines (8c) and (8d)).

The reduction of subtyping between methods performed within the loop at (8e) is more interesting. Given a method of name m_i, we first ask an (incomplete) oracle whether the types $\vec{U}_{1,i}$ of the arguments $\vec{x}$ are inhabited. We do not know whether such check is necessary to be sound: lemma 3.4.8 (based on rule (Met-<:-Met) which extends pDOT's (All-<:-All) rule) requires a $\vec{q}$ such that $\phi, \gamma[\vec{x} \mapsto \vec{q}] \models \vec{x} : \vec{U}_{1,i}$.

The rest loosely follows the results of lemma 3.4.8. Because type parameters and higher-kinded types have similar constraints semantics, we leverage the higher-kinded `DeductionIneq` cases by construction an HK abstraction out of the type parameters $\vec{Y}_i$.

Cases (11) and (12).

For conciseness, we only consider the case (11); the same reasoning applies to (12).

Assuming $T_1 \preceq U \mid V$, we have $T_1 \preceq U$ *or* $T_1 \preceq V$. When performing a deduction on $T_1 \preceq U$ and $T_2 \preceq V$, we have D_1 *or* D_2. The constraint language $\mathscr{C}$ does not have a notion of disjunction $\curlyvee$, so we cannot return $D_1 \curlyvee D_2$.

We can however approximate the disjunction as follows, knowing that D_1 and D_2 are solely composed of subtyping constraints (modulo `true` and `false`). We take all types appearing in the subtyping constraints in D_1 and D_2. For each type T, we look-up its lower and upper bounds L_1 and U_1 in D_1. If the lower bound (respectively the upper bound) is missing, we default it to $\bot$ (respectively $\top$). We do the same for the bounds in D_2. We then add the constraint $L_1 \,\&\, L_2 \preceq T \curlywedge T \preceq U_1 \mid U_2$ to the approximation result. The function `ApproxDisjunction` is charged in performing such approximation.

As an example, suppose that $D_1 \triangleq T \preceq S \curlywedge T \preceq V \curlywedge S \preceq W$ and that $D_2 \triangleq B \preceq T \curlywedge T \preceq A$.

Then, the lower and upper bounds of S, T, V and W in D_1 are:

$$\begin{aligned} L_1(S) &= T & \qquad U_1(S) &= W \\ L_1(T) &= \bot & U_1(T) &= S \,\&\, V \\ L_1(V) &= T & U_1(V) &= \top \\ L_1(W) &= S & U_1(W) &= \top \end{aligned}$$

In D_2, the lower and upper bounds of T, A, B are:

$$\begin{aligned} L_2(T) &= B & \qquad U_2(T) &= A \\ L_2(A) &= T & U_2(A) &= \top \\ L_2(B) &= \bot & U_2(B) &= T \end{aligned}$$

The approximation for T is $B \,\&\, \bot \preceq T \curlywedge T \preceq (S \,\&\, U) \mid A$, which is equal to $\bot \preceq T \curlywedge T \preceq (S \,\&\, U) \mid A$. Intuitively, if D_1 holds, then $T \preceq S \,\&\, U$ holds as well – thus entailing the approximation. On the other hand, if D_2 holds, then $B \preceq T \curlywedge T \preceq A$ holds, which entails the approximation too.

For S, the approximation yields $T \,\&\, \bot \preceq S \curlywedge S \preceq W \mid \top$, which is equal to $\bot \preceq S \curlywedge S \preceq \top$ which is equivalent to `true`. In other words, we did not extract any information for S. We similarly get trivial results for V, W, A and B.

The result of the approximation of the disjunction of D_1 and D_2 is then $T \preceq (S \,\&\, U) \mid A$. We have unfortunately lost a substantial amount of information; we believe however that such approximation should be good enough for most real-world use cases.

---

[4] One may notice that it is wiser to instead return $R_1 \preceq R_2$ and come back latter to that constraint if we find a p inhabiting R_1. Because the next phases do not support refinements, we must unfortunately drop the constraint. We leave such enhancement for future improvements.

Case (13).

We first simplify T_1 and T_2. It is possible that we eliminate some conjunctions or disjunctions with the accumulated knowledge. For instance, if T_1 is $X \mathbin{\&} Y \mathbin{\&} Z$ and that $\mathscr{K} \Vdash X \asymp Y$, the simplification process would yield $X \mathbin{\&} Z$ (or $Y \mathbin{\&} Z$). It is possible for a DNF to be simplified into a type that is not an union or an intersection type.

Next, we perform a deduction on the simplified types as we would normally do. We do not directly recur here because we would end up in the same case, so we use an helper function, `DeductionIneqDNF`, that dissects the DNFs into "atomic" types and calls `DeductionIneq` on them.

Once the result D obtained from `DeductionIneqDNF`, we may be tempted to return it and continue. However, by just returning D, we may lose some crucial information that could be useful latter on. For example, suppose that we are given $X \mathbin{\&} Y \preceq Z \mathbin{\&} W$ with no knowledge on X, Y, Z and W. Then, `DeductionIneqDNF` would return $D = \texttt{true}$ due to how logical disjunctions are approximated. If we just returned D (i.e. `true`), we would inadvertently throw the constraint $X \mathbin{\&} Y \preceq Z \mathbin{\&} W$. In such case, we should return $X \mathbin{\&} Y \preceq Z \mathbin{\&} W$ as well.

Yet, we would also like to minimize the amount of duplicated information we return.

It turns out that, if U and V are both determined under $\mathscr{K}$, then we have extracted all useful information that we could possibly derive. Further knowledge will not benefit us if we add the original constraint, so we may just return D. On the other hand, if one of the type is not determined, we may miss out on further knowledge refinement.

As an example, let us consider the following traits:

$$\texttt{trait TC1}[A];\ \texttt{trait TC2}[A];\ \texttt{trait InvInv}[A, B]$$

as well as the following constraint:

$$T_1 \preceq T_2 \triangleq \texttt{InvInv}[U, V] \mathbin{\&} \texttt{InvInv}[U, W] \preceq \texttt{InvInv}[X, Y] \mathbin{\&} \texttt{InvInv}[X, Z]$$

where U, V, W, X, Y and Z are all type variables. Suppose that we have a $\mathscr{K}$ that does not contain any information about these type variables.

We observe that T_1 and T_2 are not determined under $\mathscr{K}$: indeed, for T_1 we cannot prove that no subtyping relationship exists between $\texttt{InvInv}[U, V]$ and $\texttt{InvInv}[U, W]$ since we do not know anything about U, V and W. The same applies for T_2 as well.

The only useful simplified constraint we deduce is $U \asymp X$.

For the sake of the example, suppose furthermore that we deduced elsewhere the following constraint:

$$V \asymp \texttt{TC1}[V'] \curlywedge W \asymp \texttt{TC2}[W'] \curlywedge Y \asymp \texttt{TC1}[Y'] \curlywedge Z \asymp \texttt{TC2}[Z']$$

and that this knowledge is latter on integrated into $\mathscr{K}$.

By using the above knowledge, the original constraint becomes:

$$\texttt{InvInv}[U, \texttt{TC1}[V']] \mathbin{\&} \texttt{InvInv}[U, \texttt{TC2}[W']] \preceq \texttt{InvInv}[X, \texttt{TC1}[Y']] \mathbin{\&} \texttt{InvInv}[X, \texttt{TC2}[Z']]$$

Since `TC1` and `TC2` are unrelated, the updated versions of T_1 and T_2 are determined.

If we give the updated constraint a second chance by passing it to `DeductionIneq`, we would obtain (with the left conjunct of T_2):

$$\begin{aligned}
&\texttt{DeductionIneq(InvInv}[U, \texttt{TC1}[V']] \mathbin{\&} \texttt{InvInv}[U, \texttt{TC2}[W']],\ \ \texttt{InvInv}[X, \texttt{TC1}[Y']]\texttt{)}\\
&\quad \Rightarrow \texttt{DeductionIneq(InvInv}[U, \texttt{TC1}[V']],\ \ \texttt{InvInv}[X, \texttt{TC1}[Y']]\texttt{)}\\
&\qquad\quad \Rightarrow U \asymp X \curlywedge V' \asymp Y'\\
&\quad \Rightarrow \texttt{DeductionIneq(InvInv}[U, \texttt{TC2}[W']],\ \ \texttt{InvInv}[X, \texttt{TC1}[Y']]\texttt{)}\\
&\qquad\quad \Rightarrow \texttt{false}\\
&\ \Rightarrow U \asymp X \curlywedge V' \asymp Y'
\end{aligned}$$

We similarly obtain $U \asymp X \curlywedge W' \asymp Z'$ from the right conjunct of T_2. In particular, we have deduced a new knowledge $V' \asymp Y' \curlywedge W' \asymp Z'$ that we would have missed if we dropped the original constraint and only kept $U \asymp X$.

We provide correctness proofs (including termination) for `Deduction`, `DeductionTypedPath` and `DeductionIneq` in appendix A.4.

Algorithm 6: Reduction of a subtyping constraint into simpler constraints

```
DeductionIneq ($\mathscr{K}$, $T_1 : \mathcal{T}$, $T_2 : \mathcal{T}$) : $\curlywedge_i^n D_i$
    Inputs: The knowledge $\mathscr{K}$, the assumed constraint $T_1 \preceq T_2$ to be reduced
    Output: A (possibly trivial) conjunction of constraints $\curlywedge_i^n D_i$ entailed by $\mathscr{K}$ and $T_1 \preceq T_2$.
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$)
    Postcondition: $\mathscr{K} \curlywedge T_1 \preceq T_2 \Vdash \curlywedge_i^n D_i$ where the $D_i$ are of the form true, false or $U_1 \preceq U_2$
        with $U_1$ and $U_2$ free of any refinement.
    match $T_1 \preceq T_2$ :
(1)         case $T_1 \preceq T_2$ where $T_1, T_2 \in \mathcal{T}^{\mathrm{cl}}$ :
(1a)            if $\mathcal{T}$-IsSubtype($T_1, T_2$) = false then
                    return false
(1b)            else
                    return true
(2)         case $T \preceq T$ :
                return true
(3)         case $T_1 \preceq \top$ :
                return true
(4)         case $\bot \preceq T_2$ :
                return true
(5)         case $Cls_1[\vec{S}_1] \preceq Cls_2[\vec{S}_2]$ :
(5a)            if $Cls_1$ does not extend $Cls_2$ then
                    return false
(5b)            else if $Cls_1 = Cls_2$ then
                    With $\vec{v}$ the variance signs of $Cls_1$
                    return DeductionIneqVec($\mathscr{K}, \vec{S}_1, \vec{S}_2, \vec{v}$)
(5c)            else
                    Then, $Cls_1$ extends $Cls_2$ $N \geq 1$ times through $\boldsymbol{\sigma}_1, ..., \boldsymbol{\sigma}_N$ such that:
                    $Cls_1[\vec{S}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})] \preceq Cls_2[\vec{S}_2]$.
                    return DeductionIneq($\mathscr{K}, \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})], Cls_2[\vec{S}_2]$)
            Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
(6)         case $[\vec{v}\vec{X} \lhd B_1]$ =>> $S_1 \preceq [\vec{v}\vec{X} \lhd B_2]$ =>> $S_2$ where $B_1$ and $B_2$ do not contain refinements:
                sub ← $\mathcal{B}_{EC}$-Subsumes($\mathscr{K}, B_2, B_1$)
(6a)            if sub = false then
                    return false
(6b)            else if sub = undet then
(6b.i)              if $T_1$ and $T_2$ do not contain refinements then
                        return $T_1 \preceq T_2$
(6b.ii)             else
                        Give up.
                        return true
                $\curlywedge_i^m D_i$ ← DeductionIneq($\mathscr{K}, S_1, S_2$)
                entailed ← $\mathcal{B}_{EC}$-BoundsEntailed($\mathscr{K}, B_1$)
```

```
              $D_{acc} \leftarrow \texttt{true}$
              for $D_i \in \curlywedge_i^m D_i$ do
                  match $D_i$ :
(6c)                  case $U_1 \preceq U_2$ where entailed $\wedge\ \mathrm{ftv}(U_1, U_2) \# \bar{X}$ :
                          $D_{acc} \leftarrow D_{acc} \curlywedge U_1 \preceq U_2$
(6d)                  case $U_1 \preceq U_2$ :
                          $D_{acc} \leftarrow D_{acc} \curlywedge \big([\vec{v}\vec{X} \triangleleft B_1] \texttt{=>>} U_1 \preceq [\vec{v}\vec{X} \triangleleft B_2] \texttt{=>>} U_2\big)$
(6e)                  case $\texttt{false}$ where entailed :
                          return $\texttt{false}$
                      otherwise :
                          pass
              return $D_{acc}$
(7)       case $R \preceq Cls[\vec{S}]$ :
              return $\texttt{false}$
(8)       case $R_1 \preceq R_2$ :
              $R_1$ cannot be a subtype of $R_2$ if there are some "missing" members.
(8a)          if $R_2 \not\subseteq R_1$ then
                  return $\texttt{false}$
              Deconstructing $R_1$ and $R_2$. We assume that the $z$ (the "$\texttt{self}$") is $\alpha$-renamed to be fresh and equal
               in both $R_1$ and $R_2$.
              We furthermore assume that the arguments $\vec{x}$ for all methods are $\alpha$-renamed such that they are all
               distinct between themselves, $z$ and $ftmv(\mathscr{K})$.
              $\{z\ \texttt{=>}$
               $\texttt{type}\ \vec{T} \triangleleft B_1$
               $\texttt{type}\ \vec{T}' \triangleleft B'$
               $\texttt{val}\ \vec{f} : \vec{F}_1$
               $\texttt{val}\ \vec{f}' : \vec{F}'$
               $\texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,1}](\vec{x} : \vec{U}_1)} : \vec{V}_1$
               $\texttt{def}\ \overrightarrow{m'[\vec{Y}' \triangleleft B'_Y](\vec{x}' : \vec{U}')} : \vec{V}'\} \leftarrow R_1$
              $\{z\ \texttt{=>}$
               $\texttt{type}\ \vec{T} \triangleleft B_2$
               $\texttt{val}\ \vec{f} : \vec{F}_2$
               $\texttt{def}\ \overrightarrow{m[\vec{Y} \triangleleft B_{Y,2}](\vec{x} : \vec{U}_2)} : \vec{V}_2\} \leftarrow R_2$
(8b)          if $\neg(\exists p.\ (p, R_1) \in \mathcal{I})$ then
                  We simply give up.
                  return $\texttt{true}$
              $D_{acc} \leftarrow \texttt{true}$
              Subtyping between the type of the fields...
(8c)          for $i \leftarrow 1$ to $|\vec{F}|$ do
                  $D \leftarrow \texttt{DeductionIneq}(\mathscr{K}, [z \mapsto p]F_{1,i}, [z \mapsto p]F_{2,i})$
(8c.i)            $D_{acc} \leftarrow D_{acc} \curlywedge D$
              ... between the bounds of type members...
(8d)          for $i \leftarrow 1$ to $|\vec{T}|$ do
                  $(L_1, U_1) \leftarrow [z \mapsto p]B_{1,i}$
                  $(L_2, U_2) \leftarrow [z \mapsto p]B_{2,i}$
                  $D_1 \leftarrow \texttt{DeductionIneq}(\mathscr{K}, L_2, L_1)$
                  $D_2 \leftarrow \texttt{DeductionIneq}(\mathscr{K}, U_1, U_2)$
(8d.i)            $D_{acc} \leftarrow D_{acc} \curlywedge D_1 \curlywedge D_2$
```

```
            … and finally between the methods.
(8e)        for $i \leftarrow 1$ to $|\vec{m}|$ do
(8e.i)          if $\neg(\bigwedge_i^{|\bar{x}|}$ $\mathcal{T}$-IsInhabitedOracle($x_i, U_{1,i,j}$)) then
                    continue
                $\vec{v} \leftarrow (\pm)^{|\vec{Y}_i|}$
                if $|\vec{Y}_i| \neq 0$ then
                    Here, we take advantage of the fact that HK abstraction subtyping rules are similar to the
                     ones for methods.
(8e.ii)             for $j \leftarrow 1$ to $|\vec{x}_i|$ do
                        $U_1' \leftarrow [z \mapsto p]\big([\vec{v}\vec{Y}_i \lhd B_{Y,2,i}]$ =>> $U_{1,i,j}\big)$
                        $U_2' \leftarrow [z \mapsto p]\big([\vec{v}\vec{Y}_i \lhd B_{Y,1,i}]$ =>> $U_{2,i,j}\big)$
                        $\bigcurlywedge_l^n D_l \leftarrow$ DeductionIneq($\mathcal{K}, U_2', U_1'$)
                        We only keep constraints that do not mention the parameters $\bar{x}$. Otherwise, they would
                         escape their original scope.
                        $D_{acc} \leftarrow D_{acc} \curlywedge \bigcurlywedge\{D_l : D_l \in \bigcurlywedge_l^n D_l,\ \bar{x} \# \text{ftmv}(D_l)\}$
                    $V_1' \leftarrow [z \mapsto p]\big([\vec{v}\vec{Y} \lhd B_{Y,1,i}]$ =>> $V_{1,i}\big)$
                    $V_2' \leftarrow [z \mapsto p]\big([\vec{v}\vec{Y} \lhd B_{Y,2,i}]$ =>> $V_{2,i}\big)$
(8e.iii)            $\bigcurlywedge_l^n D_l \leftarrow$ DeductionIneq($\mathcal{K}, V_1', V_2'$)
                    $D_{acc} \leftarrow D_{acc} \curlywedge \bigcurlywedge\{D_l : D_l \in \bigcurlywedge_l^n D_l,\ \bar{x} \# \text{ftmv}(D_l)\}$
                else
(8e.iv)             for $j \leftarrow 1$ to $|\vec{x}_i|$ do
                        $\bigcurlywedge_l^n D_l \leftarrow$ DeductionIneq($\mathcal{K}, [z \mapsto p]U_{2,i,j}, [z \mapsto p]U_{1,i,j}$)
                        $D_{acc} \leftarrow D_{acc} \curlywedge \bigcurlywedge\{D_l : D_l \in \bigcurlywedge_l^n D_l,\ \bar{x} \# \text{ftmv}(D_l)\}$
(8e.v)              $\bigcurlywedge_l^n D_l \leftarrow$ DeductionIneq($\mathcal{K}, [z \mapsto p]V_{1,i}, [z \mapsto p]V_{2,i}$)
                    $D_{acc} \leftarrow D_{acc} \curlywedge \bigcurlywedge\{D_l : D_l \in \bigcurlywedge_l^n D_l,\ \bar{x} \# \text{ftmv}(D_l)\}$
            return $D_{acc}$
(9)     case $T_1 \preceq U \,\&\, V$ where DNF($T_1$) does not contain non-trivial conjunctions:
            Examples:
             $X \preceq T \,\&\, Y$ matches the branch
             $X \mid Y \mid$ Trait$[A] \preceq T \,\&\, S$ matches the branch
             $X \,\&\, Y \preceq T \,\&\, S$ does not match the branch
             $X \mid Y \mid (Z \,\&\,$ Trait$[A]) \preceq T \,\&\, S$ does not match the branch
            $D_1 \leftarrow$ DeductionIneq($\mathcal{K}, T_1, U$)
            $D_2 \leftarrow$ DeductionIneq($\mathcal{K}, T_1, V$)
            return $D_1 \curlywedge D_2$
(10)    case $U \mid V \preceq T_2$ where DNF($T_2$) does not contain non-trivial disjunctions:
            $D_1 \leftarrow$ DeductionIneq($\mathcal{K}, U, T_2$)
            $D_2 \leftarrow$ DeductionIneq($\mathcal{K}, V, T_2$)
            return $D_1 \curlywedge D_2$
(11)    case $T_1 \preceq U \mid V$ where DNF($T_1$) does not contain non-trivial conjunctions:
            $D_1 \leftarrow$ DeductionIneq($\mathcal{K}, T_1, U$)
            $D_2 \leftarrow$ DeductionIneq($\mathcal{K}, T_1, V$)
            return ApproxDisjunction($D_1, D_2$)
(12)    case $U \,\&\, V \preceq T_2$ where DNF($T_2$) does not contain non-trivial disjunctions:
            $D_1 \leftarrow$ DeductionIneq($\mathcal{K}, U, T_2$)
            $D_2 \leftarrow$ DeductionIneq($\mathcal{K}, V, T_2$)
            return ApproxDisjunction($D_1, D_2$)
```

```
(13)        case $T_1 \preceq T_2$ where $T_1$ and $T_2$ are intersection or union types and do not contain refinements:
                $U$ is a (possibly trivial) DNF of the form $|_i^n \&_j^{m_i} U_{i,j}$.
                Note that $T_1$ and $T_2$ do not contain any refinement, and are therefore $\mathcal{T}_{EC}$.
                $U \leftarrow \mathcal{T}_{EC}$-SimplifyDNF($\mathcal{K}$, DNF($T_1$))
                Same goes for $V$, though the $n$ and $m_j$ are likely different.
                $V \leftarrow \mathcal{T}_{EC}$-SimplifyDNF($\mathcal{K}$, DNF($T_2$))
                $D \leftarrow$ DeductionIneqDNF($\mathcal{K}, U, V$)
(13a)           if $D =$ false then
                    return false
(13b)           else
                    isDet$_U \leftarrow \mathcal{T}_{EC}$-IsDet($\mathcal{K}, U$)
                    isDet$_V \leftarrow \mathcal{T}_{EC}$-IsDet($\mathcal{K}, V$)
                    if isDet$_U \wedge$ isDet$_V$ then
                        return $D$
                    else
                        return $D \curlywedge U \preceq V$
(14)        case $T_1 \preceq T_2$ where $T_1$ and $T_2$ do not contain refinements:
                return $T_1 \preceq T_2$
(15)        otherwise :
                We just give up.
                return true
```

5.3.5 The compaction phase

5.3.5.1 Entry point: `Compact`

The compaction phase is defined in `Compact`, algorithm 7. The goal of this phase is to integrate a subtyping constraint $S \preceq T$ into $\mathcal{K}$ so that we may use it for further constraints deductions.

As previously seen in the running examples, the assimilation process can give new constraints back. The returned constraints are solely subtyping constraints and are not arbitrary. They tie two determined types together: we have seen in 3.5 that a subtyping constraint of two determined types is interesting because it can always be reduced. These constraints, along with the updated $\mathcal{K}$, are then fed back to $\mathcal{C}$-`Simplify`.

The compaction phase is composed of four sub-phases. The first phase (lines (1)-(2)) consists of looking for the equivalence classes of S and T, or creating them if they do not exist.

The second phase (line (3)) connects S and T by their subtyping relationship through their associated equivalence classes. This step may result in having to merge multiple ECs and generate new constraints. We will see the reasons when discussing the inequality phase.

The two last phases are interleaved and are executed within the loop at (4). The third phase merges the equivalence classes coming from the second phase. It may schedule further ECs for merging. The fourth phase propagates the determinacy of ECs becoming determined due to being merged to determined ECs.

We provide a correctness proof of the compaction phase in appendix A.5. To help readability, we employ the following shorthands in the specifications:

$$[a] \in \mathcal{K} \triangleq [a] \in \mathcal{Q}\text{-}\texttt{AllMembers}(\mathcal{K})$$

$$h \in \mathcal{K} \triangleq h \in \mathrm{dom}(\Theta)$$

$$M(\mathcal{K}, \mathsf{toMerge}) \triangleq \left\{\varsigma([x]) \asymp \varsigma([y]) : \{[x],[y]\} \in \mathsf{toMerge}\right\}$$

$$I(\mathcal{K}, \mathsf{ineqs}) \triangleq \left\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in \mathsf{ineqs}\right\}$$

$$L(\mathcal{K}, \mathsf{toMerge}) \triangleq \sum_{\{[a],[b]\} \in \mathsf{toMerge}} 1 \text{ if } ([a],[b]), ([b],[a]) \notin E_{\mathcal{S}} \;\; 0 \text{ otherwise}$$

We furthermore remind that ς is a shorthand for EC_H-`Subst`($\mathscr{K}$). When there are multiple $\mathscr{K}$ in scope, we accordingly annotate the ς.

Algorithm 7: Compaction entry-point

```
     Compact $(\mathscr{K}, S : \mathcal{T}, T : \mathcal{T}) : (\mathscr{K}', \mathsf{cstrts})$
         Inputs: The structure knowledge $\mathscr{K}$ and the constraint $S \preceq T$ to assimilate into $\mathscr{K}$
         Outputs: $(\mathscr{K}', \mathsf{cstrts})$ where cstrts is a set of core constraints to be added to the maintained
           constraints set of $\mathcal{C}$-Simplify.
         Precondition: $S$ and $T$ do not contain any refinement.
         Precondition: $\mathscr{K}$-Valid$(\mathscr{K})$
         Postcondition: $\mathscr{K} \curlywedge C \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}$
         Postcondition: $\mathscr{K}$-Valid$(\mathscr{K}')$
(1)      $(\mathscr{K}^{(1)}, [s]) \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathscr{K}, S, \emptyset, \emptyset, \mathtt{true}, \mathtt{true})$
(2)      $(\mathscr{K}^{(2)}, [t]) \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathscr{K}^{(1)}, T, \emptyset, \emptyset, \mathtt{true}, \mathtt{true})$
(3)      $(\mathscr{K}^{(3)}, \mathsf{cstrts}, \mathsf{toMerge}) \leftarrow$ TryAddInequality$(\mathscr{K}^{(2)}, [s], [t])$
         $\mathscr{K}^{(4)} \leftarrow \mathscr{K}^{(3)}$
         Loop Invariant: $\mathscr{K}$-Valid$(\mathscr{K}^{(4)}) \wedge \bigcup \mathsf{toMerge} \subseteq \mathscr{K}^{(4)} \wedge$
             $\forall \{[x], [y]\} \in \mathsf{toMerge}.\ \mathcal{T}_{EC}$-kind$(\mathscr{K}^{(4)}, [x]) = \mathcal{T}_{EC}$-kind$(\mathscr{K}^{(4)}, [y])$
         Loop Invariant: $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(4)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(4)}, \mathsf{toMerge})$
(4)      while $\exists \{[a], [b]\} \in \mathsf{toMerge}$ do
             We should first "refresh" $[a]$ and $[b]$ by getting their respective representatives.
             $[a] \leftarrow \mathcal{Q}$-Find$(\mathcal{Q}^{(4)}, [a])$
             $[b] \leftarrow \mathcal{Q}$-Find$(\mathcal{Q}^{(4)}, [b])$
             If it turns out that $[a]$ and $[b]$ are already merged, we simply continue.
(4a)         if $[a] = [b]$ then
                 $\mathsf{toMerge}^{(n)} \leftarrow \mathsf{toMerge} \setminus \{[a], [b]\}$
                 $\mathsf{toMerge} \leftarrow \mathsf{toMerge}^{(n)}$
(4b)         else
                 $(\mathscr{K}^{(n)}, \mathsf{cstrts}', \mathsf{toMerge}') \leftarrow$ Merge$(\mathscr{K}^{(4)}, [a], [b])$
                 $\mathsf{cstrts}^{(n)} \leftarrow \mathsf{cstrts} \cup \mathsf{cstrts}'$
                 $\mathsf{toMerge}^{(n)} \leftarrow (\mathsf{toMerge} \cup \mathsf{toMerge}') \setminus \{[a], [b]\}$
                 $(\mathscr{K}^{(4)}, \mathsf{cstrts}, \mathsf{toMerge}) \leftarrow (\mathscr{K}^{(n)}, \mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)})$
(5)      return $(\mathscr{K}^{(4)}, \mathsf{cstrts})$
```

5.3.5.2 Finding and creating equivalence classes

The goal of the first phase is to turn S and T (elements of $\mathcal{T}$) into $[s]$ and $[t]$ (elements of EC_H), which are more suited for our needs.

To do so, we employ the function $\mathcal{T}$-`FindOrCreateEC`, defined in algorithm 8. The first two arguments, $\mathscr{K}$ and $T : \mathcal{T}$, are rather straightforward. The third and fourth arguments, B_X and $\vec{v}_X$, respectively represent the enclosing bounds and the variance sign of the bound type variables that are introduced by a higher-kinded abstraction. When there are no enclosing bounds, we simply pass these arguments the empty set. The fifth argument $\mathsf{inHead} : \mathbb{B}$ indicates whether we are situated in a head position or not. We set the argument to `false` when entering the arguments of applied type constructors. Finally, the sixth argument, $\mathsf{create} : \mathbb{B}$, tells the function to create equivalence classes as needed to turn the given type into a $\mathcal{T}_{EC}$ type.

The $\mathcal{T}$-`FindOrCreateEC` function returns an updated $\mathscr{K}$ and the result of turning the given type into a $\mathcal{T}_{EC}$. Furthermore, the property Q-FEC3 states that, if B_X is set to the empty set, $\mathcal{T}$-`FindOrCreateEC` is guaranteed to return an EC_H, which is also a $\mathcal{T}_{EC}$ type. Hence, the $[s]$ and $[t]$ returned within `Compact` at lines (1) and (2) are indeed EC_H elements. If not, the returned type satisfies some other properties that are specified by Q-FEC4. In essence, these properties state that the returned type is suitable to be stored in Θ by maintaining the $\mathcal{T}_{EC}$-`in`-Θ-`Inv` invariant, provided it is not an EC_H.

To help accomplish its task, $\mathcal{T}$-`FindOrCreateEC` leverages $\mathcal{T}_{EC}$-`FindOrCreateEC` (algorithm 9) which turns a given $\mathcal{T}_{EC}$ type into $\mathcal{T}_{EC}$ guaranteed to satisfy Q-FEC3 and Q-FEC4. In $\mathcal{T}_{EC}$`-FindOrCreateEC`, we first try to find if the given T type is equivalent to at least one type in each equivalence classes. To avoid iterating over all types and equivalence classes, we call T_H`-Candidates` to generate a list of candidates that could be equivalent T. This function leverages the graphs $G_{\mathcal{S}}$, G_{EC} and G_p to rule out types that cannot possibly be equivalent to T. If we find a type equivalent to T, we return the EC of that type (lines (1a.i) and (2a.i)).

Otherwise, if T has a simple kind, we try to find an EC, when applied, is equivalent to T. This attempt is carried out at line (1b).

Finally, if all these attempts fail, we create an EC for T if create is `true` and return NIL otherwise. The creation of ECs is undertaken by the function $\mathcal{T}_{EC}$`-CreateEC`, defined in algorithm 10.

We provide a proof for the claims upheld by these ECs processing functions in appendix A.6.

Algorithm 8: Finding an equivalence class for a $\mathcal{T}$ type if it exists

```
$\mathcal{T}$-FindOrCreateEC $(\mathscr{K}, T : \mathcal{T}, B_X : \mathcal{B}_{EC}, \vec{v}_X, \mathsf{inHead} : \mathbb{B}, \mathsf{create} : \mathbb{B})$
                              $: (\mathscr{K}', T' : \mathcal{T}_{EC} \uplus \{NIL\})$
    Precondition (P-FEC1): $\mathscr{K}$-Valid($\mathscr{K}$)
    Precondition (P-FEC2): Valid scope:
        $|\mathrm{dom}(B_X)| = |\vec{v}_X| \wedge$
        $\mathrm{dom}(B_X) \# \mathrm{ftv}(\mathscr{K}) \wedge$
        $\mathrm{dom}(B_X) \neq \emptyset \implies (\varsigma(B_X){\downarrow} \wedge \mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_X, \emptyset))$
    Precondition (P-FEC3): $T$ does not contain any refinement.
    Postcondition (Q-FEC1): $\mathscr{K}'$ and $\mathscr{K}$ agree on common domains:
        $\mathscr{K}\texttt{-Valid}(\mathscr{K}') \ \wedge\ \mathcal{M}' \restriction \mathscr{K} = \mathcal{M} \ \wedge\ \Theta' \restriction \mathscr{K} = \Theta \ \wedge\ \mathcal{R}' \restriction \mathscr{K} = \mathcal{R} \ \wedge$
        $\mathcal{D}' \restriction \mathscr{K} = \mathcal{D} \ \wedge\ Q' \restriction \mathscr{K} = \mathcal{Q} \ \wedge\ T'_R \restriction \mathscr{K} = T_R$
      where the $\restriction \mathscr{K}$ is a shorthand for restriction to elements contained in $\mathscr{K}$.
      This implies that $\varsigma' \restriction \mathscr{K} = \varsigma$.
      It also implies that the $EC_H$ in $\mathscr{K}$ have the same kind as in $\mathscr{K}'$:
        $\forall [a] \in \mathscr{K}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, [a]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}', [a])$
    Postcondition (Q-FEC2): $T' \neq NIL \implies [\varsigma'(T'){\downarrow} \wedge$
        $\mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}', T) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}', T')]$
    Postcondition (Q-FEC3): Provided that $T'$ is not $NIL$, if $\mathrm{dom}(B_X) = \emptyset$, then $T'$ is an $EC_H$
      contained in $\mathscr{K}'$ and the representative of its EC under $\mathscr{K}'$.
    Postcondition (Q-FEC4): Provided that $T'$ is not $NIL$, if $\mathrm{dom}(B_X) \neq \emptyset$, then $T'$ is not
      of the form $X$ with $X \notin \mathrm{dom}(B_X)$ or $F[\vec{S}]$ with $F \notin \mathrm{dom}(B_X)$. Furthermore, either $T'$ an $EC_H$,
      or the assertion $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T', \mathsf{inHead}, \mathrm{dom}(B_X))$ holds.
    Postcondition (Q-FEC5): Find mode does no update; create guarantees non-nil:
        $(\neg\mathsf{create} \implies \mathscr{K} = \mathscr{K}') \wedge (\mathsf{create} \implies T' \neq NIL)$
    Postcondition (Q-FEC6): Entailment of $\mathscr{K}'$:
        $\mathscr{K} \Vdash \mathscr{K}'$
    Postcondition (Q-FEC7): Equivalence of $T$ and $T'$:
        $T' \neq NIL \implies \mathscr{K} \Vdash \varsigma'(T) \asymp \varsigma'(T')$
    match $T$ :
(1)     case $X$ where $X \in \mathrm{dom}(B_X)$ :
            return $(\mathscr{K}, X)$
(2)     case $F[\vec{S}]$ where $F \in \mathrm{dom}(B_X)$ :
(2a)        $(\mathscr{K}^{(1)}, \vec{S}') \leftarrow \mathcal{T}\texttt{-FindOrCreateECVec}(\mathscr{K}, \vec{S}, B_X, \vec{v}_X, \texttt{false}, \mathsf{create})$
(2b)        if $\vec{S}' = NIL$ then
                return $(\mathscr{K}, NIL)$
(2c)        else
                return $(\mathscr{K}^{(1)}, F[\vec{S}'])$
(3)     case $X$ or $p.Q$ :
            return $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}(\mathscr{K}, T, B_X, \vec{v}_X, \mathsf{create})$
```

```
(4)       case $TyCon[\vec{S}]$ :
(4a)          $(\mathcal{K}^{(1)}, \vec{S}') \leftarrow \mathcal{T}$-FindOrCreateECVec($\mathcal{K}, \vec{S}, B_X, \vec{v}_X$, false, create)
(4b)          if $\vec{S}' = NIL$ then
                  return $(\mathcal{K}, NIL)$
              We keep the head constructor if TyCon is a trait or a class whenever we are under an enclosing
                HK abstraction as substituting them into an EC may hinder determinacy (a type with an EC in
                a head position is not considered determined, even if the EC itself is determined). On the other
                hand, an abstract type constructor F is not determined, and substituting it with an applied EC
                can help the propagation of determinacy. If F becomes determined, we can benefit from its
                determinacy as well.
(4c)          else if $TyCon$ is a class/trait $\wedge$ $\mathrm{dom}(B_X) \neq \emptyset$ then
                  return $(\mathcal{K}^{(1)}, TyCon[\vec{S}'])$
(4d)          else
                  $(\mathcal{K}^{(2)}, T') \leftarrow \mathcal{T}_{EC}$-FindOrCreateEC($\mathcal{K}^{(1)}, TyCon[\vec{S}'], B_X, \vec{v}_X$, create)
                  return $(\mathcal{K}^{(2)}, T')$

(5)       case $T_1 \,\&\, T_2$ or $T_1 \mid T_2$ :
              $|_i^n \&_j^{m_i} S_{i,j} \leftarrow$ DNF($T$)
              $S'_{i,j} \leftarrow NIL$   for $1 \leq i \leq n, 1 \leq j \leq m_i$
              $\mathcal{K}^{(1)} \leftarrow \mathcal{K}$
(5a)          for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $m_i$ do
                  If $S_{i,j}$ is a class or trait, we keep the head constructor as substituting it to an EC or applied
                    EC would render further analysis on DNFs a bit harder.
                  match $S_{i,j}$ :
(5a.i)                case $Cls[\vec{U}]$ :
                          $(\mathcal{K}^{(n)}, \vec{U}') \leftarrow \mathcal{T}$-FindOrCreateECVec($\mathcal{K}^{(1)}, \vec{U}, B_X, \vec{v}_X$, inHead, create)
                          if $\vec{U}' = NIL$ then
                              return $(\mathcal{K}, NIL)$
                          else
                              $S'_{i,j} \leftarrow Cls[\vec{U}']$

(5a.ii)               otherwise :
                          $(\mathcal{K}^{(n)}, S'_{i,j}) \leftarrow \mathcal{T}$-FindOrCreateEC($\mathcal{K}^{(1)}, S_{i,j}, B_X, \vec{v}_X$, inHead, create)
                          if $S'_{i,j} = NIL$ then
                              return $(\mathcal{K}, NIL)$
                  $\mathcal{K}^{(1)} \leftarrow \mathcal{K}^{(n)}$
              Before going on, attempt to simplify what we have.
(5b)          $S' \leftarrow \mathcal{T}_{EC}$-SimplifyDNF($\mathcal{K}^{(1)}, |_i^n \&_j^{m_i} S'_{i,j}$)
(5c)          if $S' \in EC_H$ then
                  The simplification yielded a simple $EC_H$ (i.e., $S'$ is of the form $[a]$), so we directly return.
                  return $(\mathcal{K}^{(1)}, S')$
              If we appear in head and under an HK abstraction, we leave the DNF as-is. The reasoning is
                similar to (4c), except here we require to be in a head position to be compliant with
                $\mathcal{T}_{EC}$-in-$\Theta$-Inv
(5d)          else if $\mathcal{T}_{EC}$-IsDNF($S'$) $\wedge$ inHead $\wedge$ $\mathrm{dom}(B_X) \neq \emptyset$ then
                  return $(\mathcal{K}^{(1)}, S')$
(5e)          else
                  $(\mathcal{K}^{(2)}, S'') \leftarrow \mathcal{T}_{EC}$-FindOrCreateEC($\mathcal{K}^{(1)}, S', B_X, \vec{v}_X$, create)
                  return $(\mathcal{K}^{(2)}, S'')$
```

```
          This is a special case of (7) where we avoid creating an applied EC for $F[\vec{Y}]$ as it is not necessary.
           This case is not needed for correctness.
          Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
(6)       case $[\vec{v}_Y \vec{Y} \triangleleft B_Y]$ =>> $F[\vec{Y}]$ where $F \notin \mathrm{dom}(B_X)$ :
(6a)          $(\mathcal{K}^{(1)}, B'_Y) \leftarrow \mathcal{B}$-FindOrCreateEC$(\mathcal{K}, B_Y, \vec{v}_Y, B_X, \vec{v}_X, \mathsf{create})$
(6b)          if $B'_Y = NIL$ then
                  return $(\mathcal{K}, NIL)$
(6c)          else
                  $(\mathcal{K}^{(2)}, T') \leftarrow \mathcal{T}_{EC}$-FindOrCreateEC$(\mathcal{K}, [\vec{v}_Y \vec{Y} \triangleleft B'_Y]$ =>> $F[\vec{Y}], \mathsf{create})$
                  return $(\mathcal{K}^{(2)}, T')$
          Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
(7)       case $[\vec{v}_Y \vec{Y} \triangleleft B_Y]$ =>> $S$ :
(7a)          $(\mathcal{K}^{(1)}, B'_Y) \leftarrow \mathcal{B}$-FindOrCreateEC$(\mathcal{K}, B_Y, \vec{v}_Y, B_X, \vec{v}_X, \mathsf{create})$
(7b)          if $B'_Y = NIL$ then
                  return $(\mathcal{K}, NIL)$
(7c)          $(\mathcal{K}^{(2)}, S') \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathcal{K}^{(1)}, S, B_X B'_Y, \vec{v}_X \vec{v}_Y, \mathsf{inHead}, \mathsf{create})$
(7d)          if $S' = NIL$ then
                  return $(\mathcal{K}, NIL)$
(7e)          else
                  $(\mathcal{K}^{(3)}, T') \leftarrow \mathcal{T}_{EC}$-FindOrCreateEC$(\mathcal{K}^{(2)}, [\vec{v}_Y \vec{Y} \triangleleft B'_Y]$ =>> $S', B_X, \vec{v}_X, \mathsf{create})$
                  return $(\mathcal{K}^{(3)}, T')$
          Match is syntactically exhaustive: all cases are covered
```

Algorithm 9: Finding an equivalence class for a $\mathcal{T}_{EC}$ type if it exists

```
  $\mathcal{T}_{EC}$-FindOrCreateEC $(\mathcal{K}, T : \mathcal{T}_{EC},\ B_X : \mathcal{B}_{EC}, \vec{v}_X, \mathsf{create} : \mathbb{B})$
      Precondition: Same as P-FEC1 and P-FEC2
      Precondition: $\varsigma(T)\downarrow \wedge\ \mathcal{T}_{EC}$-in-$\Theta$-Inv$(T, \mathtt{true}, \mathrm{dom}(B_X))$
        Note: even if $T$ does not appear in head, we require it to satisfy the predicate as-if it appeared in head.
      Precondition: $T$ not of the form $X$ with $X \in \mathrm{dom}(B_X)$ or $F[\vec{S}]$ with $F \in \mathrm{dom}(B_X)$.
      Postcondition: Same as Q-FEC1 - Q-FEC7
      match $T$ :
(1)       case $T$ where $\mathcal{T}_{EC}$-kind$(\mathcal{K}, T) = \star$ :
              candidates $\leftarrow T_H$-Candidates$(\mathcal{K}, T, \mathrm{dom}(B_X))$
(1a)          if $\mathrm{ftv}(T) \# \mathrm{dom}(B_X)$ then
                  for $h \in$ candidates do
                      match $\Theta(h)$ :
                          case $S$ where $\mathrm{kind}(S) = \star$ :
(1a.i)                        if $\mathcal{T}_{EC}$-Equiv$(\mathcal{K}, T, S)$ then
                                  return $(\mathcal{K}, \mathcal{R}(h))$
                          otherwise :
                              continue
(1b)          $T' \leftarrow \mathcal{T}_{EC}$-TryFindApplied$(\mathcal{K}, T, B_X)$
(1c)          if $T' \neq NIL$ then
                  return $(\mathcal{K}, T')$
(1d)          if create then
                  return $\mathcal{T}_{EC}$-CreateEC$(\mathcal{K}, T, B_X, \vec{v}_X)$
(1e)          else
                  return $(\mathcal{K}, NIL)$
```

```
            Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
(2)         case $[\vec{v}_Y \vec{Y} \lhd B_1]$ =>> $S_1$ :
                if $\mathrm{ftv}(T) \# \mathrm{dom}(B_X)$ then
                    for $h \in T_H$-Candidates($\mathcal{K}, T, \mathrm{dom}(B_X)$) do
(2a)                    match $\Theta(h)$ :
                            case $[\vec{v}_Y \vec{Y} \lhd B_2]$ =>> $S_2$ :
(2a.i)                          if $\mathcal{B}_{EC}$-Equiv($\mathcal{K}, B_1, B_2$) $\wedge$ $\mathcal{T}_{EC}$-Equiv($\mathcal{K}, S_1, S_2$) then
                                    return $(\mathcal{K}, \mathcal{R}(h))$
                            otherwise :
                                continue
                Here, we could do better by trying to find an appropriate $[\vec{v}_X \vec{v}_Y \vec{X}\vec{Y} \lhd B_X, B_Y]$ =>> $[a]\,[\vec{X}, \vec{Y}]$
(2b)            if create then
                    return $\mathcal{T}_{EC}$-CreateEC($\mathcal{K}, T, B_X, \vec{v}_X$)
(2c)            else
                    return $(\mathcal{K}, NIL)$
            Match is syntactically exhaustive: all cases are covered

$\mathcal{T}_{EC}$-TryFindApplied ($\mathcal{K}, T : \mathcal{T}_{EC},\ B_X : \mathcal{B}_{EC}$, create : $\mathbb{B}$)
    Precondition: Same as P-FEC1 and P-FEC2
    Precondition: $\varsigma(T)\!\downarrow \wedge$ $\mathcal{T}_{EC}$-kind($\mathcal{K}, T$) $= \star \wedge$ $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$, true, $\mathrm{dom}(B_X)$)
    Precondition: $T$ not of the form $X$ with $X \in \mathrm{dom}(B_X)$ or $F[\vec{S}]$ with $F \in \mathrm{dom}(B_X)$.
    Postcondition: Similar to Q-FEC2, Q-FEC3, Q-FEC4, Q-FEC7
    match $T$ :
        We are not interested in finding an applied EC in such cases.
        case $[a]$ or $X$ or $p.Q$ :
            return $NIL$
        otherwise :
            for $h \in T_H$-Candidates($\mathcal{K}, T, \mathrm{dom}(B_X)$) do
                match $\Theta(h)$ :
                    Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
                    case $[\vec{v}_Y \vec{Y} \lhd B_Y]$ =>> $S$ :
(1)                     $\sigma \leftarrow$ $\mathcal{T}_{EC}$-TryMatch($\mathcal{K}, \bar{Y}, S, T$)
                        if $\sigma = NIL$ then
                            continue
(2)                     Extending $\sigma$ with $\top$ for $Y$'s not appearing in $S$
                        $\sigma' \leftarrow \sigma[Y \mapsto \top_{\mathrm{kind}(Y)}, Y \in \bar{Y} \setminus \mathrm{dom}(\sigma)]$
                        Destructuring $\sigma'$.
                        $[\vec{Y} \mapsto \vec{A}] \leftarrow \sigma'$
(3)                     if $\neg\mathcal{B}_{EC}$-Satisified($\mathcal{K}, B_Y, [\vec{Y} \mapsto \vec{A}]$) then
                            continue
(4)                     applied $\leftarrow \mathcal{R}(h)[\vec{A}]$
(5)                     if $\mathrm{ftv}(T) \# \mathrm{dom}(B_X) \wedge \mathrm{ftv}(\text{applied}) = \emptyset$ then
(5a)                        $[a] \leftarrow$ $\mathcal{T}_{EC}$-TryFindECOfApplied($\mathcal{K}$, applied)
(5b)                        if $[a] \neq NIL$ then
                                return $[a]$
(6)                     if $\mathrm{ftv}(T) \cap \mathrm{dom}(B_X) \neq \emptyset$ then
                            return applied
                    otherwise :
                        continue
    return $NIL$
```

Algorithm 10: Creating an equivalence class for a given $\mathcal{T}_{EC}$

```
        $\mathcal{T}_{EC}$-CreateEC  $(\mathscr{K}, T : \mathcal{T}_{EC}, B_X : \mathcal{B}_{EC}, \vec{v}_X)$
            Precondition: Same as P-FEC1 and P-FEC2
            Precondition: $\varsigma(T)\downarrow \wedge \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T, \texttt{true}, \mathrm{dom}(B_X))$
            Precondition: $T$ not of the form $X$ with $X \in \mathrm{dom}(B_X)$ or $F[\vec{S}]$ with $F \in \mathrm{dom}(B_X)$.
            Postcondition: Same as Q-FEC1 - Q-FEC7
            $\vec{X} \leftarrow \mathrm{dom}(B_X)$
            The type $\tilde{T}$ will be put in $\Theta$ whereas $T'$ is the type that we will return. $T'$ will either be an $EC_H$ or an applied $EC_H$.
            let $\tilde{T}, T'$
            Note: since $[a]$ is fresh, it cannot appear in $T$.
(1)         $(\mathcal{Q}^{(1)}, [a]) \leftarrow \mathcal{Q}\texttt{-MakeSet}(Q)$
(2)         match $T$ :
(2a)            case $T$ where $\mathrm{kind}(T) = \star$ :
(2a.i)              if $\mathrm{ftv}(T) \,\#\, \bar{X}$ then
                        $\tilde{T} \leftarrow T$
                        $T' \leftarrow [a]$
(2a.ii)             else
                        $\tilde{T} \leftarrow [\vec{v}_X \vec{X} \lhd B_X] \texttt{=>>} T$
                        EC application well formed because the $\vec{X}$ are guarded by the enclosing scope, ensuring the bounds are satisfied.
                        $T' \leftarrow [a][\vec{X}]$
(2b)            Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
                case $[\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} S$ :
(2b.i)              if $\mathrm{ftv}(T) \,\#\, \bar{X}$ then
                        $\tilde{T} \leftarrow T$
                        $T' \leftarrow [a]$
(2b.ii)             else
                        $\tilde{T} \leftarrow [\vec{v}_X \vec{v}_Y \vec{X}\vec{Y} \lhd B_X, B_Y] \texttt{=>>} S$
                        $T' \leftarrow [\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} [a][\vec{X}, \vec{Y}]$
(3)         Type handle for $\tilde{T}$
            $h_{\tilde{T}} \leftarrow$ fresh $T_H$
            Now we create the "representative" type handle.
            $h_R \leftarrow$ fresh $T_H$
            $\mathcal{M}^{(1)} \leftarrow \mathcal{M}\big[[a] \mapsto \{h_{\tilde{T}}, h_R\}\big]$
            $\mathcal{R}^{(1)} \leftarrow \mathcal{R}\big[h_{\tilde{T}}, h_R \mapsto [a]\big]$
            $T_R^{(1)} \leftarrow T_R\big[[a] \mapsto h_R\big]$
            $\Theta^{(1)} \leftarrow \Theta\big[h_{\tilde{T}} \mapsto \{\tilde{T}\}, h_R \mapsto \{\varsigma(\tilde{T})\}\big]$
            $\mathscr{K}^{(1)} \leftarrow \mathscr{K}[\mathcal{M} \mapsto \mathcal{M}^{(1)}, \Theta^{(1)} \mapsto \Theta, \mathcal{R}^{(1)} \mapsto \mathcal{R}, \mathcal{Q}^{(1)} \mapsto \mathcal{Q}, T_R^{(1)} \mapsto T_R]$
(4)         $(\mathsf{syms}, \mathsf{ecsH}, \mathsf{ecsNH}, \mathsf{pathDep}) \leftarrow \mathcal{T}_{EC}\texttt{-Composition}(\tilde{T}, \bar{X})$
(5a)        if $\mathsf{syms} \neq \emptyset$ then
                $U_{\mathcal{S}}^{(2)} \leftarrow U_{\mathcal{S}}^{(1)} \cup \mathsf{syms}$
                $V_{\mathcal{S}}^{(2)} \leftarrow V_{\mathcal{S}}^{(1)} \cup \{h_{\tilde{T}}\}$
                $E_{\mathcal{S}}^{(2)} \leftarrow E_{\mathcal{S}}^{(1)} \cup \{(sym, h_{\tilde{T}}) : sym \in \mathsf{syms}\}$
                $G_{\mathcal{S}}^{(2)} \leftarrow (U_{\mathcal{S}}^{(2)}, V_{\mathcal{S}}^{(2)}, E_{\mathcal{S}}^{(2)})$
(5b)        else
                $G_{\mathcal{S}}^{(2)} \leftarrow G_{\mathcal{S}}^{(1)}$
```

```
(6)    $\mathscr{K}^{(2)} \leftarrow \mathscr{K}[G_{\mathcal{S}} \mapsto G_{\mathcal{S}}^{(2)}]$
(7a)   if $\mathsf{ecsH} \cup \mathsf{ecsNH} \neq \emptyset$ then
           $U_{EC}^{(3)} \leftarrow U_{EC}^{(2)} \cup \mathsf{ecsH} \cup \mathsf{ecsNH}$
           $V_{EC}^{(3)} \leftarrow V_{EC}^{(2)} \cup \{h_{\tilde{T}}\}$
           $E_{EC}^{(3)} \leftarrow E_{EC}^{(2)} \cup \{([b], h_{\tilde{T}}) : [b] \in \mathsf{ecsH} \cup \mathsf{ecsNH}\}$
           $L_{EC}^{(3)} \leftarrow L_{EC}^{(2)} \cup \{(([b], h_{\tilde{T}}), H) : [b] \in \mathsf{ecsH}\} \cup \{(([b], h_{\tilde{T}}), NH) : [b] \in \mathsf{ecsNH}\}$
           $G_{EC}^{(3)} \leftarrow (U_{EC}^{(3)}, V_{EC}^{(3)}, E_{EC}^{(3)}, L_{EC}^{(3)})$
(7b)   else
           $G_{EC}^{(1)} \leftarrow G_{EC}$
(8)    $\mathscr{K}^{(3)} \leftarrow \mathscr{K}[G_{EC} \mapsto G_{EC}^{(3)}]$
(9a)   if $\mathsf{pathDep} \neq \emptyset$ then
           $U_p^{(4)} \leftarrow U_p^{(3)} \cup \mathsf{pathDep}$
           $V_p^{(4)} \leftarrow V_p^{(3)} \cup \{h_{\tilde{T}}\}$
           $E_p^{(4)} \leftarrow E_p^{(3)} \cup \{((p, sym), h_{\tilde{T}}) : (p, sym) \in \mathsf{pathDep}\}$
           $G_p^{(4)} \leftarrow (U_p^{(4)}, V_p^{(4)}, E_p^{(4)})$
(9b)   else
           $G_p^{(4)} \leftarrow G_p^{(3)}$
(10)   $\mathscr{K}^{(4)} \leftarrow \mathscr{K}[G_p \mapsto G_p^{(4)}]$
(11a)  if $\mathcal{T}_{EC}$-IsDet($\mathscr{K}^{(4)}, \tilde{T}$) then
           $\mathcal{D}^{(5)} \leftarrow \mathcal{D}^{(4)}\big[[a] \mapsto h_{\tilde{T}}\big]$
(11b)  else
           $\mathcal{D}^{(5)} \leftarrow \mathcal{D}^{(4)}$
(12)   $\mathscr{K}^{(5)} \leftarrow \mathscr{K}[\mathcal{D} \mapsto \mathcal{D}^{(5)}]$
(13)   return $(\mathscr{K}^{(5)}, T')$
```

5.3.5.3 Adding a subtyping relationship

The second phase – consisting in tying S and T through their ECs $[s]$ and $[t]$ – is defined in `TryAddInequality`, algorithm 11.

We first start with the reason why `TryAddInequality` may return new constraints or schedule ECs for merging. When we add a link in the subtyping DAG $G_{\preceq}$ between $[s]$ and $[t]$, we may create some interesting subtyping relations as a result.

As an example, let us consider the figure 5.4. In this scenario, tying $[s]$ and $[t]$ results in transitively connecting many ECs together, such as $[x]$ to $[a]$, $[c]$, and $[b]$ to $[a]$, $[c]$. In particular, the established relationship between $[a]$ and $[x]$ is of interest because both are determined. The determined type for $[a]$ is $\texttt{Cov}\big[[b]\big]$ and the determined type for $[x]$ is $\texttt{Cov}\big[[g]\big]$. Having a constraint tying two determined types is interesting since that constraint can be simplified. In that example, we add $\texttt{Cov}\big[[g]\big] \preceq \texttt{Cov}\big[[b]\big]$ to the set of returned constraints `cstrts`. Latter on, the $\mathcal{C}$-Simplify loop will in turn simplify the above constraint into $[g] \preceq [b]$.

On the other hand, we would like avoid “polluting” the returned constraint sets with pointless information we already have. For instance, we do not return the constraint $F\big[[a]\big] \preceq [d]\ \&\ [e]$ since we cannot say much about it.

One may notice that we do not exactly return $\texttt{Cov}\big[[g]\big] \preceq \texttt{Cov}\big[[b]\big]$ but $\varsigma(\texttt{Cov}\big[[g]\big]) \preceq \varsigma(\texttt{Cov}\big[[b]\big])$. In 5.2.2, we have explained that ς serves to substitute the equivalence classes appearing in types with their associated type representative. We have furthermore affirmed that such substitution is only needed for interpreting constraints with $\mathcal{T}_{EC}$ types because $\mathscr{C}$ solely treats $\mathcal{T}$ types. It is not quite true: in (4d), we use ς to substitute the $\mathcal{T}_{EC}$ types into $\mathcal{T}$ types before forming a subtyping constraint. The reason is to simplify the proofs, as we would need to introduce a syntax for constraints composed of $\mathcal{T}_{EC}$ *and* $\mathcal{T}$ and define their interpretation under ς. In an implementation, we would not perform such substitution and return these types as-is. We would need to adapt `DeductionIneq` to process equivalence classes similarly to how they are treated in $\mathcal{T}_{EC}$-`IsSubtype`.

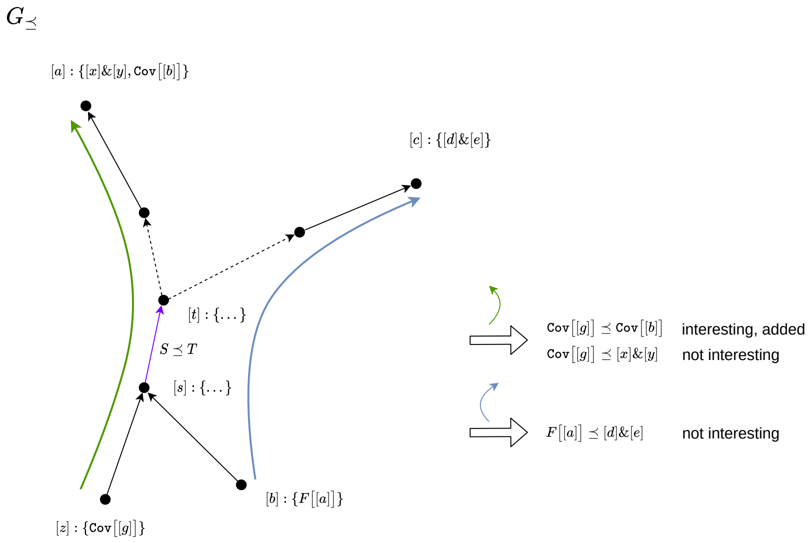


Figure 5.4 – Discovering new constraints as the result of the new subtyping relation between $[s]$ and $[t]$

Connecting $[s]$ and $[t]$ may also result in creating forward edges. However, these can be dealt with locally and do not require any “global action”. Tying $[s]$ and $[t]$ may nonetheless result in a cycle, as shown in figure 5.5. In that case, we *do not* connect $[s]$ and $[t]$ together to maintain the acyclicity of $G_{\preceq}$ (hence the **Try** prefix of `TryAddInequality`). A cycle essentially means that the ECs in that cycle are actually equivalent to each other. Instead, we schedule $[s]$ and $[t]$ for merging by returning them into the set toMerge. The merge loop phase at (4) will take care of merging all ECs appearing in the cycle that would be formed if we connected $[s]$ and $[t]$.

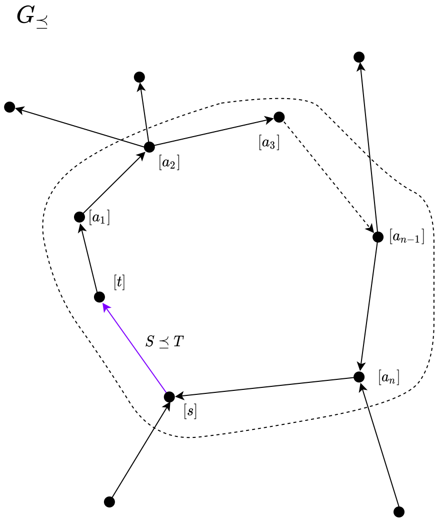


Figure 5.5 – A new subtyping relation between $[s]$ and $[t]$ resulting in a cycle.

We prove the claims stated by `TryAddInequality` in appendix A.7.

Algorithm 11: Tying two ECs in an inequality

```
TryAddInequality ($\mathscr{K}, [a], [b]$) : ($\mathscr{K}'$, cstrts : $\mathscr{P}(\mathcal{C})$, toMerge : $\mathscr{P}(\binom{EC_H}{2})$)
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$) $\wedge$ $[a], [b] \in \mathscr{K}$ $\wedge$ $\mathcal{T}_{EC}$-kind($\mathscr{K}, [a]$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}, [b]$)
    Postcondition: $\mathscr{K}$-Valid($\mathscr{K}'$) $\wedge$ $\mathcal{M}' = \mathcal{M}$ $\wedge$ $\Theta' = \Theta$ $\wedge$ $\mathcal{R}' = \mathcal{R}$ $\wedge$
        $\mathcal{D}' = \mathcal{D}$ $\wedge$ $\mathcal{Q}' = \mathcal{Q}$ $\wedge$ $T_R' = T_R$
    Postcondition: $\bigcup$toMerge $\subseteq \mathscr{K}$ $\wedge$
        $\forall\{[x], [y]\} \in$ toMerge. $\mathcal{T}_{EC}$-kind($\mathscr{K}', [x]$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}', [y]$)
    Postcondition: $\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge$cstrts $\curlywedge M(\mathscr{K}'$, toMerge$)$
    Postcondition: $\neg$ExistUndirChain($G_{\preceq}, [a], [b]$) $\implies$ $([a], [b]) \in E_{\preceq}'$

    $[a] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [a]$)
    $[b] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [b]$)
(1) if $[a] = [b]$ then
        return ($\mathscr{K}, \emptyset, \emptyset$)
(2) else if ExistChain($G_{\preceq}, [a], [b]$) then
        Either [a] and [b] are directly connected with an edge, or there are some intermediate types between
          them. In either case, we already have $[a] \preceq [b]$ so we just return.
        return ($\mathscr{K}, \emptyset, \emptyset$)
(3) else if ExistChain($G_{\preceq}, [b], [a]$) then
        Adding an edge between [a] and [b] would result in cycle, which we do not want. We schedule [a] and
          [b] for merging. The Merge function will determine what to do with the other types tied between [b]
          and [a].
        return ($\mathscr{K}, \emptyset, \{\{[a], [b]\}\}$)
(4) else
(4a)    $\varsigma \leftarrow EC_H$-Subst($\mathscr{K}$)
        Get all (known) lower bounds of [a].
        allLower $\leftarrow$ LeadingTo($G_{\preceq}, [a]$) $\cup \{[a]\}$
        Do the same for the upper bounds of [b].
        allUpper $\leftarrow$ ReachableFrom($G_{\preceq}, [b]$) $\cup \{[b]\}$
        In case [a] and [b] were not already contained in the graph.
(4b)    $V_{\preceq}' \leftarrow V_{\preceq} \cup \{[a], [b]\}$
        We remove all edges becoming forward due to directly connecting [a] and [b]
        $E_{\preceq}' \leftarrow E_{\preceq} \setminus$ (allLower $\times$ allUpper)
        $G_{\preceq}' \leftarrow (E_{\preceq}', V_{\preceq}')$
(4c)    Retain (strict) lower bounds of [a] that have a determined type.
        $\text{allLower}_{\text{det}} \leftarrow \text{dom}(\mathcal{D}) \cap (\text{allLower} \setminus \{[a]\})$
        Do the same for the strict upper bounds of [b].
        $\text{allUpper}_{\text{det}} \leftarrow \text{dom}(\mathcal{D}) \cap (\text{allUpper} \setminus \{[b]\})$
        Create new constraints based on the cartesian products of these two sets and return.
(4d)    cstrts $\leftarrow \{\varsigma(\Theta(\mathcal{D}([l]))) \preceq \varsigma(\Theta(\mathcal{D}([u]))) : [l] \in \text{allLower}_{\text{det}}, [u] \in \text{allUpper}_{\text{det}}\}$
        return ($\mathscr{K}[G_{\preceq} \mapsto G_{\preceq}']$, cstrts, $\emptyset$)
```

5.3.5.4 Merging two equivalence classes

The merge loop phase, situated at (4), maintains a set of equivalence classes to merge, and is initialized with the returned set by `TryAddInequality` at line (3) (which is either empty or $\{\{[s], [t]\}\}$). It similarly maintains a set of constraints, and is initialized with the returned set by `TryAddInequality` as well.

The loop repeatedly dequeues pairs of ECs to merge and calls `Merge`, defined in algorithm 12. The `Merge` function, charged in merging the given $[a]$ and $[b]$, may return more ECs to merge and new constraints, which are added to the set toMerge and `cstrts` respectively.

Let us have a look at the definition of `Merge`. At (1) and (4), we prepare $\mathscr{K}$ to facilitate the merging. The actual merge occurs at (5) with the call to `MergeHelper`, defined in algorithm 13. We start by considering (4).

The preparations at (4) essentially involve treating equivalence classes determinacy. If one equivalence class has a determined type (making the EC determined) while the other not, the merging of these two classes will turn the non-determined class determined ((4b) and (4c)). In such scenarios, we perform a *propagation of determinacy* which can reveal new constraints and equivalency between other ECs. The propagation of determinacy is carried out by the `PropagateDeterminacy` function that we describe in more detail in the next section. Otherwise, if both classes are determined (4a), we may retain only one determined type. We arbitrarily keep the determined type of $[b]$: to not lose the information provided by $[a]$, we generate a constraint stating that the determined type of $[a]$ and $[b]$ must be equal. It must be the case, because we are asked to merge $[a]$ and $[b]$ due to becoming equivalent.

We now go back to (1) whose task is to adjust $G_{\preceq}$. Intuitively, updating $\mathscr{K}$ to accommodate the union of $[a]$ and $[b]$ is rather straightforward, except for the subtyping graph $G_{\preceq}$. We need to make sure it remains acyclic and forward-free.

The arrangements at (1) are there to ease the process. We start by analyzing the relationship between $[a]$ and $[b]$ with respect to subtyping.

If there is a directed edge between $[a]$ and $[b]$ or $[b]$ and $[a]$ (cases (1a) or (1c.i)), we do not perform any preparation and do not touch $G_{\preceq}$. Then, the `MergeHelper` function can proceed. We do not need to worry about forming a cycle. We have to however take care of removing edges becoming forward as a result of merging $[a]$ and $[b]$. This case is illustrated by figure 5.6. Here, $[ab]$ refers to either $[a]$ or $[b]$ and depends on which one is picked by $\mathcal{Q}$`-Union` as a representative for the merged partition.

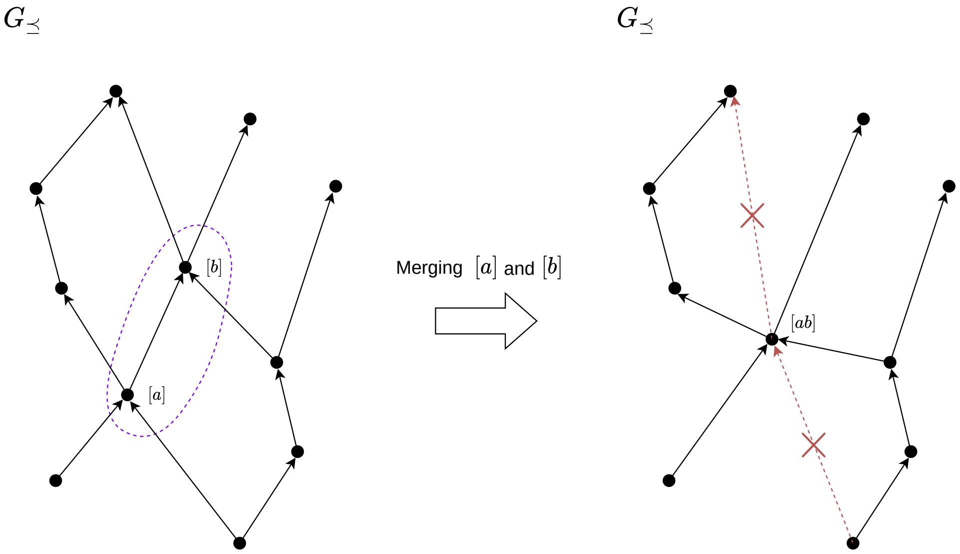


Figure 5.6 – $[a]$ and $[b]$ are directly connected and are merged into the node $[ab]$. The red dotted lines refer to edges becoming forward and need to be removed.

On the other hand, if there is a path $[x_1], ..., [x_n]$ with n greater or equal to 1 between $[a]$ and $[b]$ or $[b]$ and $[a]$ (cases (1b) or (1c.ii)), merging $[a]$ and $[b]$ would cause a cycle. Instead, we schedule the merge of $[a]$ with $[x_1]$, $[x_1]$ with $[x_2]$ and so on. We do so by adding them to a set `toMerge` and have it returned from `Merge` [5]. These ECs will be picked up by the merge loop in `Compact` at (4). Figure 5.7 illustrates this scenario.

Finally, if there are no path between $[a]$ and $[b]$ or $[b]$ and $[a]$, we artificially add a subtyping relationship between $[a]$ and $[b]$. We are "allowed" to do so because we are under the assumption that $[a]$ and $[b]$ became equivalent. Note that this cannot create a cycle; as a consequence, `TryAddInequality` will add a subtyping edge between $[a]$ and $[b]$. The merging then operates identically to (1a).

---

[5] We could alternatively recur; since we already have a merge loop at disposal, we may as well use it.

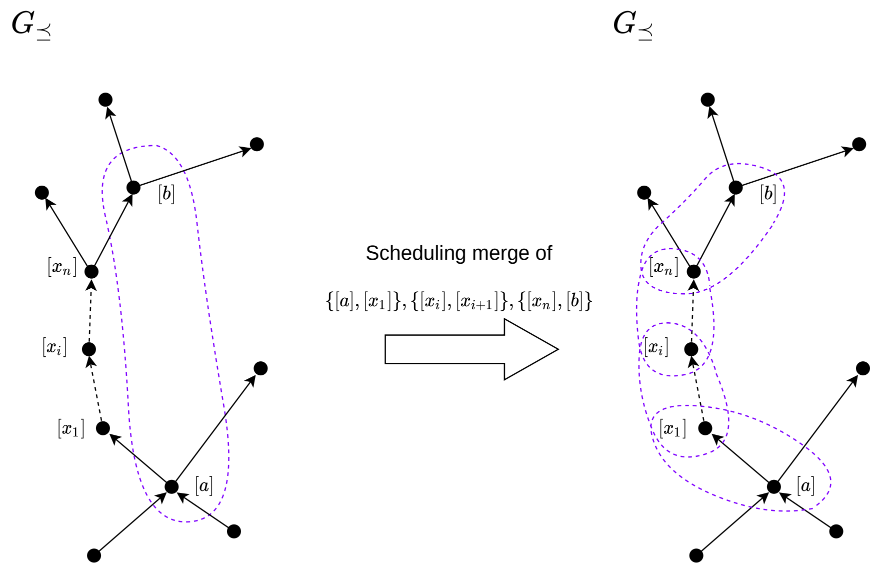


Figure 5.7 – $[a]$ and $[b]$ are connected by a non-trivial path: the nodes constituting it are scheduled for merging.

The `Merge` function (and similarly `MergeHelper`) essentially guarantees that the returned $\mathscr{K}$ and constraints set are entailed by the conjunction of the original $\mathscr{K}$ and the constraint tying $[a]$ and $[b]$ in an equality. We provide a proof for these two functions in appendix A.8.

Algorithm 12: Fusing two equivalence classes into one

```
     Merge $(\mathscr{K}, [a], [b]) : (\mathscr{K}', \mathsf{cstrts} : \mathscr{P}(\mathcal{C}), \mathsf{toMerge} : \mathscr{P}(\binom{EC_H}{2}))$
         Precondition: $\mathscr{K}$-Valid$(\mathscr{K}) \wedge [a],[b] \in \mathscr{K} \wedge \mathcal{Q}$-Find$(\mathscr{K},[a]) = [a] \wedge$
             $\mathcal{Q}$-Find$(\mathscr{K},[b]) = [b] \wedge [a] \neq [b] \wedge \mathcal{T}_{EC}$-kind$(\mathscr{K},[a]) = \mathcal{T}_{EC}$-kind$(\mathscr{K},[b])$
         Postcondition (Q-MG1): $\mathscr{K}$-Valid$(\mathscr{K}') \wedge \big([x] \in \mathscr{K} \iff [x] \in \mathscr{K}'\big) \wedge$
             $\forall [x] \in \mathscr{K}.\ \mathcal{T}_{EC}$-kind$(\mathscr{K},[x]) = \mathcal{T}_{EC}$-kind$(\mathscr{K}',[x])$
         Postcondition (Q-MG2): $\bigcup \mathsf{toMerge} \subseteq \mathscr{K} \wedge$
             $\forall \{[x],[y]\} \in \mathsf{toMerge}.\ \ \mathcal{T}_{EC}$-kind$(\mathscr{K}',[x]) = \mathcal{T}_{EC}$-kind$(\mathscr{K}',[y])$
         Postcondition (Q-MG3):
             $\big[\big(([a],[b]),([b],[a]) \notin E_{\preceq} \wedge$ ExistUndirChain$(G_{\preceq},[a],[b])\big) \implies$
               $\mathscr{K}' = \mathscr{K} \wedge L(\mathscr{K}, \mathsf{toMerge}) = 0\big] \wedge$
             $\big[\neg\big(([a],[b]),([b],[a]) \notin E_{\preceq} \wedge$ ExistUndirChain$(G_{\preceq},[a],[b])\big) \implies$
               $|\mathrm{dom}(\mathcal{M}')| < |\mathrm{dom}(\mathcal{M})|\big]$
         Postcondition (Q-MG4): $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}', \mathsf{toMerge}) \curlywedge$
             $\bigcurlywedge \{\varsigma([x]) \asymp \varsigma'([x]) : [x] \in \mathscr{K}\}$
         $\mathscr{K}^{(1)} \leftarrow \mathscr{K}$
         $\mathsf{cstrts}^{(1)} \leftarrow \emptyset$
         We now verify what ties [a] and [b] together, and adjust if necessary.
(1)      match Chain$(G_{\preceq},[a],[b])$:
(1a)         case $([a],[b])$ :
                 i.e. [a] and [b] are directly connected, in which case we do not perform anything special.
                 pass
(1b)         case $([a],[x_1],\dots,[x_n],[b]),\ n \geq 1$ :
                 We break the chain into simple chains that we can merge more easily. We note that we do not
                  need to keep the order of the chain.
                 $\mathsf{toMerge} \leftarrow \big\{\{[a],[x_1]\}\big\} \cup \big\{\{[x_i],[x_{i+1}]\}, 1 \leq i < n\big\} \cup \big\{\{[x_n],[b]\}\big\}$
                 return $(\mathscr{K}, \emptyset, \mathsf{toMerge})$
```

```
            case $NIL$ :
                Maybe there is a chain in the other direction.
(1c)            match Chain($G_{\preceq}, [b], [a]$):
(1c.i)              case $([b], [a])$ :
                        As in the outer match.
                        pass
(1c.ii)             case $([b], [x_1], ..., [x_n], [a]),\ n \geq 1$ :
                        As in the outer match.
                        toMerge $\leftarrow \big\{\{[b], [x_1]\}\big\} \cup \big\{\{[x_i], [x_{i+1}]\}, 1 \leq i < n\big\} \cup \big\{\{[x_n], [a]\}\big\}$
                        return $(\mathcal{K}, \emptyset, \text{toMerge})$
(1c.iii)            case $NIL$ :
                        No link between $[a]$ and $[b]$, so we artificially add one and keep on. Note that this cannot
                         result in a cycle; as such, we can ignore the returned toMerge set.
                        $(\mathcal{K}^{(1)}, \text{cstrts}^{(1)}, \_) \leftarrow$ TryAddInequality($\mathcal{K}, [a], [b]$)
        Note: the first assertion is about $G_{\preceq}$ (from the original $\mathcal{K}$) while the second one is about $G_{\preceq}^{(1)}$ (from the
         updated $\mathcal{K}^{(1)}$).
(2)     assert $\neg\big(([a], [b]), ([b], [a]) \notin E_{\preceq} \land$ ExistUndirChain($G_{\preceq}, [a], [b]$)$\big)$
(3)     assert $([a], [b]) \in E_{\preceq}^{(1)} \lor ([b], [a]) \in E_{\preceq}^{(1)}$
        $\mathcal{K}^{(2)} \leftarrow \mathcal{K}^{(1)}$
        $\text{cstrts}^{(2)} \leftarrow \text{cstrts}^{(1)}$
        $\text{toMerge}^{(2)} \leftarrow \emptyset$
        If one of $[a]$ and $[b]$ is determined while the other one is not, the merge will render the non-determined
         EC determined, by the virtue of merging it with a determined EC.
(4)     match $([a] \in \text{dom}(\mathcal{D}^{(1)}), [b] \in \text{dom}(\mathcal{D}^{(1)}))$ :
(4a)        case (true, true) :
                Both ECs are determined and we may only keep one determined type. We generate a constraint
                 tying both determined types in an equality and arbitrarily remove the determined type of $[a]$.
                No determinacy propagation is needed, since both ECs were already determined.
                $\varsigma^{(1)} \leftarrow$ $EC_H$-Subst($\mathcal{K}^{(1)}$)
                $\text{cstrts}^{(2)} \leftarrow \text{cstrts}^{(1)} \cup \{\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a]))) \asymp \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b])))\}$
                $\mathcal{K}^{(2)} \leftarrow$ RemoveMember($\mathcal{K}^{(1)}, \mathcal{D}^{(1)}([a])$)
(4b)        case (true, false) :
                $(\mathcal{K}', \text{cstrts}, \text{toMerge}) \leftarrow$ PropagateDeterminacy($\mathcal{K}^{(1)}, [b], \mathcal{D}^{(1)}([a])$)
                $\text{cstrts}' \leftarrow \text{cstrts}^{(1)} \cup \text{cstrts}$
                $\text{toMerge}^{(2)} \leftarrow \text{toMerge}$
                The propagation of determinacy may have rendered $[b]$ determined under $\mathcal{K}'$. If it is the case, we
                 do something similar to the (4a) case.
(4b.i)          if $[b] \in \text{dom}(\mathcal{D}')$ then
                    $\varsigma' \leftarrow$ $EC_H$-Subst($\mathcal{K}'$)
                    $\text{cstrts}^{(2)} \leftarrow \text{cstrts}' \cup \{\varsigma'(\Theta'(\mathcal{D}'([a]))) \asymp \varsigma'(\Theta'(\mathcal{D}'([b])))\}$
                    $\mathcal{K}^{(2)} \leftarrow$ RemoveMember($\mathcal{K}', \mathcal{D}'([b])$)
(4b.ii)         else
                    $\mathcal{K}^{(2)} \leftarrow \mathcal{K}'$
                    $\text{cstrts}^{(2)} \leftarrow \text{cstrts}'$
```

```
(4c)          case $(\mathtt{false}, \mathtt{true})$ :
                  $(\mathscr{K}', \mathsf{cstrts}, \mathsf{toMerge}) \leftarrow \mathtt{PropagateDeterminacy}(\mathscr{K}^{(1)}, [a], \mathcal{D}^{(1)}([b]))$
                  $\mathsf{cstrts}' \leftarrow \mathsf{cstrts}^{(1)} \cup \mathsf{cstrts}$
                  $\mathsf{toMerge}^{(2)} \leftarrow \mathsf{toMerge}$
(4c.i)            if $[a] \in \mathrm{dom}(\mathcal{D}')$ then
                      $\varsigma' \leftarrow EC_H\mathtt{-Subst}(\mathscr{K}')$
                      $\mathsf{cstrts}^{(2)} \leftarrow \mathsf{cstrts}' \cup \{\varsigma'(\Theta'(\mathcal{D}'([a]))) \asymp \varsigma'(\Theta'(\mathcal{D}'([b])))\}$
                      $\mathscr{K}^{(2)} \leftarrow \mathtt{RemoveMember}(\mathscr{K}', \mathcal{D}'([a]))$
(4c.ii)           else
                      $\mathscr{K}^{(2)} \leftarrow \mathscr{K}'$
                      $\mathsf{cstrts}^{(2)} \leftarrow \mathsf{cstrts}'$
(4d)          otherwise
                  No propagation is needed if both ECs are non-determined.
                  pass
(5)       $(\mathscr{K}^{(3)}, \mathsf{cstrts}, \mathsf{toMerge}) \leftarrow \mathtt{MergeHelper}(\mathscr{K}^{(2)}, [a], [b])$
(6)       return $(\mathscr{K}^{(3)}, \mathsf{cstrts}^{(2)} \cup \mathsf{cstrts}, \mathsf{toMerge}^{(2)} \cup \mathsf{toMerge})$
```

Algorithm 13: Updating $\mathscr{K}$ to accommodate for the fusion of two ECs

```
      MergeHelper $(\mathscr{K}, [a], [b]) : (\mathscr{K}', \mathsf{cstrts} : \mathscr{P}(\mathcal{C}), \mathsf{toMerge} : \mathscr{P}(\binom{EC_H}{2}))$
          Precondition: $\mathscr{K}\mathtt{-Valid}(\mathscr{K}) \wedge [a], [b] \in \mathscr{K} \wedge \mathcal{Q}\mathtt{-Find}(\mathscr{K}, [a]) = [a] \wedge$
              $\mathcal{Q}\mathtt{-Find}(\mathscr{K}, [b]) = [b] \wedge [a] \neq [b] \wedge \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}, [a]) = \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}, [b])$
          Precondition: $(([a], [b]) \in E_{\preceq} \vee ([b], [a]) \in E_{\preceq}) \wedge$
              $\neg([a] \in \mathrm{dom}(\mathcal{D}) \wedge [b] \in \mathrm{dom}(\mathcal{D}))$
          Postcondition (Q-MGH1): $\mathscr{K}\mathtt{-Valid}(\mathscr{K}') \wedge |\mathrm{dom}(\mathcal{M}')| < |\mathrm{dom}(\mathcal{M})| \wedge$
              $[x] \in \mathscr{K} \iff [x] \in \mathscr{K}' \wedge$
              $\forall [x] \in \mathscr{K}.\ \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}, [x]) = \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}', [x])$
          Postcondition (Q-MGH2): $\bigcup \mathsf{toMerge} \subseteq \mathscr{K} \wedge$
              $\forall \{[x], [y]\} \in \mathsf{toMerge}.\ \ \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}', [x]) = \mathcal{T}_{EC}\mathtt{-kind}(\mathscr{K}', [y])$
          Postcondition (Q-MGH3): $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}', \mathsf{toMerge}) \curlywedge$
              $\bigcurlywedge \{\varsigma([x]) \asymp \varsigma'([x]) : [x] \in \mathscr{K}\}$
          Unifying $[a]$ and $[b]$
          Merging $[a]$ and $[b]$ together. The yielded $[ab]$ is either $[a]$ or $[b]$ (it is not a new element).
(1a)      $(\mathcal{Q}^{(1)}, [ab]) \leftarrow \mathcal{Q}\mathtt{-Union}(\mathcal{Q}, [a], [b])$
          assert $[ab] \in \{[a], [b]\}$
          $\mathscr{K}^{(1)}$ does not satisfy $\mathscr{K}\mathtt{-Valid}$ because the second part of K-INV2 does no longer hold.
(1b)      $\mathscr{K}^{(1)} \leftarrow \mathscr{K}[\mathcal{Q} \mapsto \mathcal{Q}^{(1)}]$
          Merging the members $\mathcal{M}$
          We first "undefined" $[a]$ and $[b]$. Then, we add the entry for $[ab]$. We remind that $[ab]$ is either $[a]$ or $[b]$,
           so we have to perform the copy-update sequentially.
(2a)      $\mathcal{M}^{(2)} \leftarrow \mathcal{M}\big[[a], [b] \mapsto \uparrow\big]\big[[ab] \mapsto \mathcal{M}([a]) \cup \mathcal{M}([b])\big]$
(2b)      $\mathscr{K}^{(2)} \leftarrow \mathscr{K}^{(1)}[\mathcal{M} \mapsto \mathcal{M}^{(2)}]$
          We do not update $\Theta$ to use the new $[ab]$ over $[a]$ and $[b]$
          One of the reason of using ECs is to avoid having to perform substitution. For that, when comparing types
           in $\Theta$, we have to be careful to first perform a $\mathcal{Q}\mathtt{-Find}$ on each EC constituting the type to get the
           "up-to-date" representatives. We use $\mathcal{T}_{EC}\mathtt{-Equiv}$ to test whether two types in $\Theta$ are equivalent.
```

```
       Updating the range of $\mathcal{R}$ to refer to $[ab]$
(3a)   $\mathcal{R}^{(3)} \leftarrow \{(h,[ec]) : h \in \mathrm{dom}(\mathcal{R}), [ec] = [ab] \text{ if } \mathcal{R}(h) \in \{[a],[b]\} \text{ or } \mathcal{R}(h) \text{ otherwise}\}$
(3b)   $\mathscr{K}^{(3)} \leftarrow \mathscr{K}^{(2)}[\mathcal{R} \mapsto \mathcal{R}^{(3)}]$
       Merging the determined type
       By assumptions, at most one of $[a]$ or $[b]$ has a determined type.
(4a)   if $[a] \in \mathrm{dom}(\mathcal{D})$ then
           $\mathcal{D}^{(4)} \leftarrow \mathcal{D}\big[[a],[b] \mapsto \uparrow\big]\big[[ab] \mapsto \mathcal{D}([a])\big]$
(4b)   else if $[b] \in \mathrm{dom}(\mathcal{D})$ then
           $\mathcal{D}^{(4)} \leftarrow \mathcal{D}\big[[a],[b] \mapsto \uparrow\big]\big[[ab] \mapsto \mathcal{D}([b])\big]$
(4c)   else
           $\mathcal{D}^{(4)} \leftarrow \mathcal{D}\big[[a],[b] \mapsto \uparrow\big]$
(4d)   $\mathscr{K}^{(4)} \leftarrow \mathscr{K}^{(3)}[\mathcal{D} \mapsto \mathcal{D}^{(4)}]$
       Merging the type representative
       We arbitrarily choose to keep $[a]$'s type representative.
(5a)   $T_R^{(5)} \leftarrow T_R\big[[a],[b] \mapsto \uparrow\big]\big[[ab] \mapsto T_R([a])\big]$
(5b)   $\mathscr{K}^{(5)} \leftarrow \mathscr{K}^{(4)}[T_R \mapsto T_R^{(5)}]$
       Updating $G_{\preceq}$
       Retrieve all (known) lower bounds of $[a]$ and $[b]$.
(6a)   $\mathsf{allLower} \leftarrow (\texttt{LeadingTo}(G_{\preceq},[a]) \cup \texttt{LeadingTo}(G_{\preceq},[b])) \cup \{[a],[b]\}$
       Do the same for their upper bounds.
       $\mathsf{allUpper} \leftarrow (\texttt{ReachableFrom}(G_{\preceq},[a]) \cup \texttt{ReachableFrom}(G_{\preceq},[b])) \cup \{[a],[b]\}$
       $\mathsf{forward} \leftarrow \{([l],[u]) : [l] \in \mathsf{allLower}, [u] \in \mathsf{allUpper}\}$
                   $\setminus \Big(\{([l],[ab]) : ([l],[a]),([l],[b]) \in E_{\preceq}\}$
                   $\cup \{([ab],[u]) : ([a],[u]),([b],[u]) \in E_{\preceq}\}\Big)$
(6b)   $\mathsf{lower} \leftarrow \{[l] : ([l],[a]) \in E_{\preceq}, [l] \neq [b]\} \cup \{[l] : ([l],[b]) \in E_{\preceq}, [l] \neq [a]\}$
       $\mathsf{upper} \leftarrow \{[u] : ([a],[u]) \in E_{\preceq}, [u] \neq [b]\} \cup \{[u] : ([b],[u]) \in E_{\preceq}, [u] \neq [a]\}$
       Edges containing $[a]$ or $[b]$
       $\mathsf{abConns} \leftarrow \{([x],[y]) : ([x],[y]) \in E_{\preceq},\ [x] \in \{[a],[b]\} \vee [y] \in \{[a],[b]\}\}$
       $\mathsf{extra} \leftarrow \{([l],[ab]) : [l] \in \mathsf{lower}\} \cup \{([ab],[u]) : [u] \in \mathsf{upper}\}$
(6c)   $V_{\preceq}^{(6)} \leftarrow (V_{\preceq} \setminus \{[a],[b]\}) \cup \{[ab]\}$
(6d)   $E_{\preceq}^{(6)} \leftarrow (E_{\preceq} \setminus (\mathsf{forward} \cup \mathsf{abConns})) \cup (\mathsf{extra} \setminus \mathsf{forward})$
(6e)   $\mathscr{K}^{(6)} \leftarrow \mathscr{K}^{(5)}[G_{\preceq} \mapsto (V_{\preceq}^{(6)}, E_{\preceq}^{(6)})]$
       Updating $G_{EC}$
       Getting all members where $[a]$ and $[b]$ occur. We will use the set to update $E_{EC}$.
(7a)   $\mathsf{occ}_{[a]} \leftarrow \{h : ([a],h) \in E_{EC}\}$
(7b)   $\mathsf{occ}_{[b]} \leftarrow \{h : ([b],h) \in E_{EC}\}$
       Grouping all of the above members by their belonging equivalence class. Note that $[a]$ and $[b]$ may appear
         in $\mathsf{grp}$ as well, in case of cyclical or mutually recursive references. We will use this set further below.
(7c)   $\mathsf{grp} \leftarrow \big\{([ec], \{h : h \in \mathsf{occ}_{[a]} \cup \mathsf{occ}_{[b]},\ \mathcal{R}^{(6)}(h) = [ec]\}) :\ [ec] \in U_{EC}\big\}$
       We now remove $[a]$ and $[b]$ from $G_{EC}$. We will use what we have built earlier to reconstruct $G_{EC}$ with $[a]$
         and $[b]$ merged.
(7d)   $U_{EC}^{(7)} \leftarrow (U_{EC} \setminus \{[a],[b]\}) \cup \{[ab]\}$
(7e)   $E'_{EC} \leftarrow \big(E_{EC} \setminus \big\{([x],[y]) : ([x],[y]) \in E_{EC},\ [x] \in \{[a],[b]\} \vee [y] \in \{[a],[b]\}\big\}\big)$
(7f)   $E_{EC}^{(7)} \leftarrow E'_{EC} \cup \{([ab],h) : h \in \mathsf{occ}_{[a]} \cup \mathsf{occ}_{[b]}\}$
(7g)   $L_{EC}^{(7)} \leftarrow (L_{EC} \restriction E'_{EC}) \cup \big\{(([ab],h),\mathsf{pos}) : h \in \mathsf{occ}_{[a]} \cup \mathsf{occ}_{[b]},$
                   $\mathsf{pos} = NH \text{ if } L_{EC}([a],h) = L_{EC}([b],h) = NH \text{ or } H \text{ otherwise}\big\}$
(7h)   $\mathscr{K}^{(7)} \leftarrow \mathscr{K}^{(6)}[G_{EC} \mapsto (U_{EC}^{(7)}, V_{EC}, E_{EC}^{(7)})]$
```

```
      No update for $G_S$ and $G_p$
      These graphs work with type handles and not with with EC handles.
(7i)  assert $\mathscr{K}$-Valid($\mathscr{K}^{(7)}$)
      Removing duplicate members
      We collect all members whose underlying type became equivalent thanks to the merging of $[a]$ and $[b]$. It is
        also possible that we remove some members of $[ab]$ too.
      rmTyHandles $\leftarrow \emptyset$
(8)   for $([ec], \bar{h}) \in$ grp do
          for $\{h_1, h_2\} \in \big\{\{h_1, h_2\} : h_1, h_2 \in \bar{h}, h_1 \neq m_2\big\}$ do
              If one of the member is to be removed, ignore that entry.
              if $\{h_1, h_2\} \cap$ rmTyHandles $\neq \emptyset$ then
                  continue
              That is, both $[a]$ and $[b]$ appear in both $h_1$ and $h_2$.
              if $\{h_1, h_2\} \cap \mathsf{occ}_{[a]} \neq \emptyset \wedge \{h_1, h_2\} \cap \mathsf{occ}_{[b]} \neq \emptyset$ then
                  if $\mathcal{T}_{EC}$-Equiv($\mathscr{K}^{(7)}, \Theta^{(7)}(h_1), \Theta^{(7)}(h_2)$) then
                      $h_1$ should not be a type representative, but doing so eases the correctness proof.
                      rmTyHandles $\leftarrow$ rmTyHandles $\cup (\{h_1\} \setminus \mathrm{Im}(T_R^{(7)}))$
      $\mathscr{K}^{(8)} \leftarrow \mathscr{K}^{(7)}$
(9)   for $h \in$ rmTyHandles do
          $\mathscr{K}^{(n)} \leftarrow$ RemoveMember($\mathscr{K}^{(8)}, h$)
          $\mathscr{K}^{(8)} \leftarrow \mathscr{K}^{(n)}$
      $\mathsf{occ}_{[a]} \leftarrow \mathsf{occ}_{[a]} \setminus$ rmTyHandles
      $\mathsf{occ}_{[b]} \leftarrow \mathsf{occ}_{[b]} \setminus$ rmTyHandles
      Searching for other classes to merge.
      Due to the fusion of $[a]$ and $[b]$, it is possible that some (distinct) ECs $[x]$ and $[y]$ become equivalent by
        having a member in $[x]$ become equivalent to a member in $[y]$.
      toMerge $\leftarrow \emptyset$
(10)  for $\{h_1, h_2\} \in \big\{\{h_1, h_2\} : h_1 \in \mathsf{occ}_{[a]}, h_2 \in \mathsf{occ}_{[b]}\big\}$ do
          $[ec_1] \leftarrow \mathcal{R}^{(8)}(h_1)$
          $[ec_2] \leftarrow \mathcal{R}^{(8)}(h_2)$
          We are not interested in merging an EC with itself. We also skip ECs that are already marked for
            merging.
(10a)     if $[ec_1] = [ec_2] \vee \{ec_1, ec_2\} \in$ toMerge then
              continue
(10b)     if $\mathcal{T}_{EC}$-kind($\mathscr{K}^{(8)}, [ec_1]$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}^{(8)}, [ec_2]$) $\wedge$ $\mathcal{T}_{EC}$-Equiv($\mathscr{K}^{(8)}, \Theta^{(8)}(h_1), \Theta^{(8)}(h_2)$) then
              toMerge $\leftarrow$ toMerge $\cup \{[ec_1], [ec_2]\}$
      Similarly to TryAddInequality, we generate subtyping constraints between the determined types of the
        lower and upper bounds of $[ab]$.
(11)  $U \leftarrow$ ReachableFrom($G_{\preceq}^{(8)}, [ab]$)$\setminus\{[ab]\}$
      $U_{det} \leftarrow \{\Theta^{(8)}(\mathcal{D}^{(8)}([u])) : [u] \in \mathrm{dom}(\mathcal{D}^{(8)}([u])), [u] \in U\}$
      $L \leftarrow$ LeadingTo($G_{\preceq}^{(8)}, [ab]$)$\setminus\{[ab]\}$
      $L_{det} \leftarrow \{\Theta^{(8)}(\mathcal{D}^{(8)}([l])) : [l] \in \mathrm{dom}(\mathcal{D}^{(8)}([l])), [l] \in L\}$
      cstrts $\leftarrow \{\varsigma^{(8)}(T_l) \preceq \varsigma^{(8)}(T_u) : T_l \in L_{det}, T_u \in U_{det}\}$
(12)  return ($\mathscr{K}^{(8)}$, cstrts, toMerge)
```

5.3.5.5 Propagation of determinacy

The function `PropagateDeterminacy` is charged in performing the propagation of determinacy of an equivalence class. This phase can reveal new constraints and equivalency between other ECs. It is defined in algorithm 14.

As seen in the last section, this phase happens during a merge of two equivalence classes $[a]$ and $[b]$. In particular, if one class is determined while the other is not, the propagation is triggered for the class that is not determined. For this discussion, we suppose that $[b]$ is determined and $[a]$ is not (corresponding to case (4c) in `Merge`). Then, a propagation is performed for $[a]$ with the determined type of $[b]$ which we refer to as T. We point out that, while $[a]$ is technically speaking *not yet* determined, we treat and view it as if it was determined with T.

The result of `PropagateDeterminacy` is a triplet of the updated structure $\mathscr{K}$, a set of constraints arising from the determinacy propagation as well as a set of unordered pairs of ECs that are equivalent under the updated $\mathscr{K}$ – awaiting to be merged by the merge loop of `Compact`.

The propagation phase proceeds as follows:

- We look for non-determined types within Θ where $[a]$ appears in a head position and collect their type handles T_H into a set $\mathsf{headSubst}$. We similarly search types where $[a]$ transitively appears in a non-head (i.e. argument) position within a non-determined DNF and gather the associated type handles into a set $\mathsf{refreshDNF}$. This step is done at line (1) through the function `GatherAffected`, defined in algorithm 18.
- We scan all types where $[a]$ *could appear* through its members constituted of an abstract type constructors and assemble all type handles into a set $\mathsf{trySubst}$. This step is done at line (2) through `GatherPotentiallyAffected` (algorithm 19).
- We explicitly substitute $[a]$ to the determined type T for all types referenced by the set $\mathsf{headSubst}$. These substitutions can yield new constraints, reveal equivalency between ECs needing to be merged, and render ECs to be determined. This step is performed at line (3) with `PropagateHeadSubst`, defined in algorithm 15.
- We attempt to simplify all DNFs referenced by the set $\mathsf{refreshDNF}$ as a result of the determinacy of $[a]$. This step is carried out by `PropagateDNFRefresh` at line (4). Similarly to `PropagateHeadSubst`, we may unveil new constraints, equivalency between ECs and turn some ECs determined. `PropagateDNFRefresh` is defined in algorithm 16.
- We attempt to substitute the abstract type constructors referenced by $\mathsf{trySubst}$ with `PropagateTrySubst` at line (4). We get the same type of result as `PropagateHeadSubst` and `PropagateDNFRefresh`. `PropagateTrySubst` is defined in algorithm 17.
- All ECs that became determined from steps (3)-(5), have their determinacy recursively propagated. This operation is carried out within the loop at (7).

As an example, consider the scenario depicted in figure 5.8. We are interested in propagating the determinacy of $[a]$, whose determined type is $T = \mathsf{Cov}[S]$. We will ignore the steps done at (2) and (5) and reserve them for the next example.

We naturally start with the gathering step, at line (1). Leveraging the G_{EC} graph, we get to know that $[a]$ appears in a head position in h_{ec_2} (a member of $[e]$), in a head position in h_{c_2} and in a non-head position in h_{c_1} (both members of $[c]$). Because $[a]$ appears in head within h_{c_2} and h_{e_2}, we schedule these for substitution by inserting them into $\mathsf{headSubst}$.

We are also interested in searching for types where $[a]$ transitively appears in an argument (or non-head) position within non-determined DNFs. The rational behind is, if some argument in a DNF becomes determined, then there is a chance that the DNF become determined as a result. In this example, $[a]$ appears in an argument position within h_{c_1}, a member of $[c]$ which is also the determined type of $[c]$ [6]. We then look

---

[6] If h_{c_1} was not the determined type of $[c]$, we would have eagerly stopped the search, as the determinacy of $[a]$ could not possibly turn the candidates DNFs determined.

for appearance of $[c]$ in any positions. We stumble on h_{d_1}, whose class $[d]$ does not appear anywhere; h_{d_1} is not a DNF so we do not retain it. Contrarily, h_{f_1} is a non-determined DNF and is therefore inserted into refreshDNF.

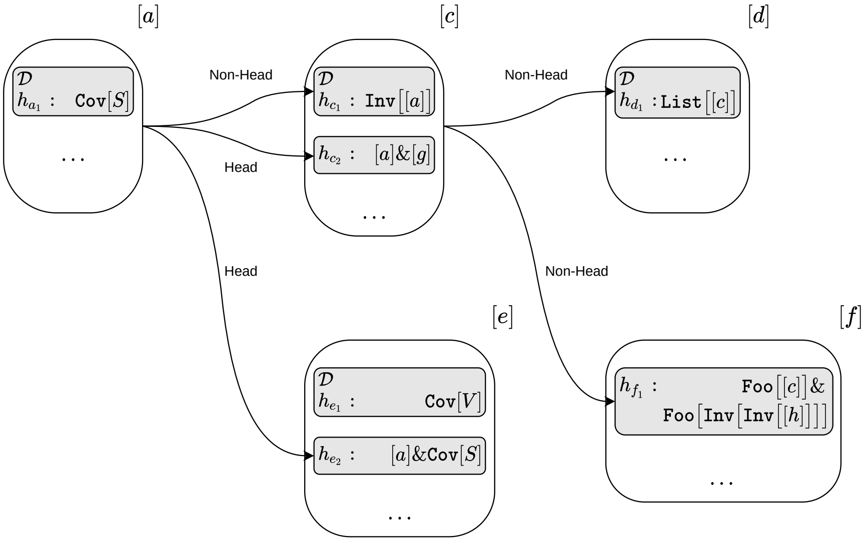


Figure 5.8 – Appearance of an EC $[a]$ in other ECs.
Each EC is comprised of at least the stated members.

We arrive at line (3). Starting with h_{c_2}, a substitution of $[a]$ to $\mathtt{Cov}[S]$ yields $\mathtt{Cov}[S]\ \&\ [g]$, which is not determined [7]. Nonetheless, we update h_{c_2} with this new type. For h_{e_2}, the substitution yields $\mathtt{Cov}[S]\,\&\,\mathtt{Cov}[S]$ which is equal to $\mathtt{Cov}[S]$. This type is determined, but $[e]$ already has a determined type. Since all types within an EC are equal, we deduce that $\mathtt{Cov}[V]$ and $\mathtt{Cov}[S]$ must be equal. As such, we generate the constraint $\mathtt{Cov}[V] \asymp \mathtt{Cov}[S]$ and remove h_{e_2} from $[e]$ because it is no longer useful. Later on, the deduction phase will infer that V and S are equal. Note that, akin to `TryAddInequality`, the actual generated constraints are passed under ς to substitute out the potential ECs involved in the constraints. For readability, we omit the ς for the current and the next example.

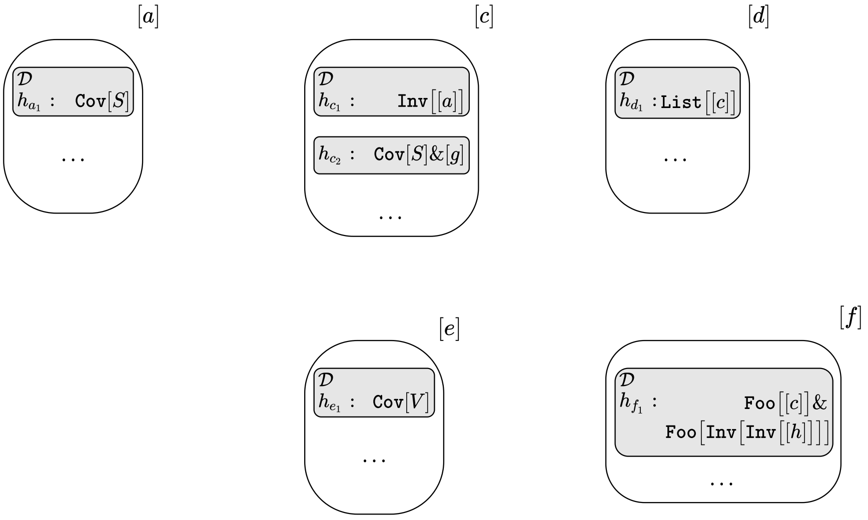
The function `PropagateDNFRefresh` is called with a set of a single element, namely h_{f_1}. We proceed by trying to simplify the DNF referred by h_{f_1}, i.e. $\mathtt{Foo}\big[[c]\big]\ \&\ \mathtt{Foo}\big[\mathtt{Inv}\big[\mathtt{Inv}\big[[h]\big]\big]\big]$. $\mathcal{T}_{EC}$-`SimplifyDNF` returns the DNF unchanged because the determined type of $[c]$ and $\mathtt{Inv}\big[\mathtt{Inv}\big[[h]\big]\big]$ are not equivalent. However, assuming that `Cov` and `Inv` are unrelated, we can check that the underlying type h_{f_1} is now determined. Indeed, the determined type of $[c]$, $\mathtt{Inv}[\mathtt{Cov}[S]]$, and $\mathtt{Inv}\big[\mathtt{Inv}\big[[h]\big]\big]$ are provably not subtype of each other. Since h_{f_1} is now determined, $[f]$ becomes determined and is thus added to the set of determined ECs. `PropagateDNFRefresh` continues by updating the underlying type of h_{f_1} with the result returned by $\mathcal{T}_{EC}$-`SimplifyDNF`. $\mathcal{T}_{EC}$-`SimplifyDNF` returned the identity, so the update is a no-op. Note that we do not substitute $[c]$ in $\mathtt{Foo}\big[[c]\big]\ \&\ \mathtt{Foo}\big[\mathtt{Inv}\big[\mathtt{Inv}\big[[h]\big]\big]\big]$ with its determined type: because $[c]$ appears in an argument position, the substitution does not offer any advantage and is therefore avoided.

We finally reach the loop at (7). Only $[f]$ became determined as a result of the determined of $[a]$. Because $[f]$ does not appears in other ECs, the recursive call is trivial.

Figure 5.9 shows the state of $\mathscr{K}$ once the propagation of determinacy has been accomplished, as well as the unveiled, resulting constraints.

---

[7] A DNF with an EC appears in a head position is not considered determined. Had the EC been determined, we would have substituted it with its determined type in an earlier iteration.



Resulting constraints set: $\{\texttt{Cov}[S] \asymp \texttt{Cov}[V]\}$

Figure 5.9 – The result of passing the scenario described by 5.8 through the determinacy propagation of $[a]$.

We now illustrate steps (2) and (5) through the scenario depicted in figure 5.10. We will only consider equivalence classes of simple kind, these steps nonetheless apply to higher-kinded ECs as well.

We first look for all abstract type constructors appearing in $[a]$. In this case, F and G are the sole candidates. Using the $G_{\mathcal{S}}$ graph, we then search for appearances of F and G within types of other equivalence classes and record a mapping for substitutability check. We get $\textsf{trySubst} = \big\{h_{c_2} \mapsto \{F[\texttt{Inv}[S]], F[\texttt{Cov}[S]]\}, h_{d_2} \mapsto \{G[\texttt{Cov}[V]]\}\big\}$.

Step (5) then proceeds as follows. We loop through $\textsf{trySubst}$ and check if h_{c_2} and h_{d_2} are equivalent to one of their respective candidates. Starting with h_{c_2}, we check whether $F[\texttt{Inv}[V]]$ is (provably) equivalent to $F[\texttt{Cov}[U]]$. Because we do not know the nature of F, we cannot say anything and continue with the next entry, namely $F[\texttt{Cov}[U]]$. Because $F[\texttt{Cov}[U]]$ is equivalent to itself, we can substitute it to $\texttt{Foo}[A]$. We then proceed similarly to steps (3) and (4). Due to $[c]$ already having a determined type and $\texttt{Foo}[A]$ being determined, we remove h_{c_2} from $[c]$ and create the constraint $\texttt{Foo}[A] \asymp \texttt{Foo}[B]$. We carry out analogous operations for h_{d_2}, resulting in the generation of the constraint $\texttt{Foo}[A] \asymp \texttt{Foo}[C]$ and the removal of h_{d_2} from $[d]$.

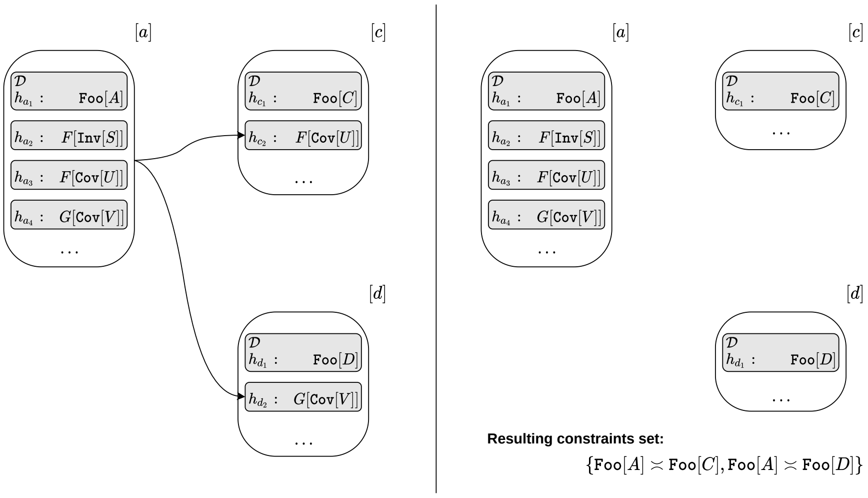


Figure 5.10 – Left: appearance of an EC $[a]$ in other ECs, all containing the stated abstract type constructors. Right: resulting $\mathscr{K}$ from the determinacy propagation of $[a]$.

We provide correctness proofs for all the functions (directly) involved in the determinacy propagation in appendix A.9.

Algorithm 14: Propagating the determinacy of an EC

```
$\textbf{PropagateDeterminacy}\ (\mathscr{K}, [a] : EC_H, T : \mathcal{T}_{EC}) : (\mathscr{K}', \mathsf{cstrts} : \mathscr{P}(\mathcal{C}), \mathsf{toMerge} : \mathscr{P}(\binom{EC_H}{2}))$
    Precondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}) \land [a] \in \mathscr{K} \land \mathcal{Q}\texttt{-Find}(\mathcal{Q}, [a]) = [a] \land$
        $\varsigma(T)\downarrow \land \mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}, T) \land \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, [a]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, T) \land$
        $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T)$
    Postcondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}') \land \varsigma = \varsigma' \land \mathcal{Q} = \mathcal{Q}' \land T_R = T'_R \land G_{\preceq} = G'_{\preceq}$
    Postcondition: $\bigcup \mathsf{toMerge} \subseteq \mathscr{K} \land$
        $\forall \{[x], [y]\} \in \mathsf{toMerge}.\ \ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}', [x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}', [y])$
    Postcondition: $\mathrm{dom}(\mathcal{D}) \subseteq \mathrm{dom}(\mathcal{D}')$
    Postcondition: $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}', \mathsf{toMerge})$
(1) $(\mathsf{headSubst}, \mathsf{refreshDNF}, \_) \leftarrow \texttt{GatherAffected}(\mathscr{K}, [a], \emptyset)$
(2) $\mathsf{trySubst} \leftarrow \texttt{GatherPotentiallyAffected}(\mathscr{K}, [a])$
(3) $(\mathscr{K}^{(1)}, \mathsf{cstrts}^{(1)}, \mathsf{toMerge}^{(1)}, \mathsf{dets}^{(1)}) \leftarrow \texttt{PropagateHeadSubst}(\mathscr{K}, [a], T, \mathsf{headSubst})$
    It is not necessary to “refresh” DNFs for which a substitution has been performed.
(4) $(\mathscr{K}^{(2)}, \mathsf{cstrts}^{(2)}, \mathsf{toMerge}^{(2)}, \mathsf{dets}^{(2)}) \leftarrow \texttt{PropagateDNFRefresh}(\mathscr{K}^{(1)}, \mathsf{refreshDNF} \setminus \mathsf{headSubst})$
(5) $(\mathscr{K}^{(3)}, \mathsf{cstrts}^{(3)}, \mathsf{toMerge}^{(3)}, \mathsf{dets}^{(3)}) \leftarrow \texttt{PropagateTrySubst}(\mathscr{K}^{(2)}, \mathsf{trySubst}, T)$
(6) $(\mathscr{K}^{(4)}, \mathsf{cstrts}^{(4)}, \mathsf{toMerge}^{(4)}, \mathsf{dets}^{(4)}) \leftarrow (\mathscr{K}^{(3)}, \bigcup_{i=1}^{3} \mathsf{cstrts}^{(i)}, \bigcup_{i=1}^{3} \mathsf{toMerge}^{(i)}, \bigcup_{i=1}^{3} \mathsf{dets}^{(i)})$
    Loop Invariant: $\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(4)}) \land \varsigma = \varsigma^{(4)} \land \varsigma = \mathcal{Q}^{(4)} \land T_R = T_R^{(4)} \land$
        $G_{\preceq} = G_{\preceq}^{(4)}$
    Loop Invariant: $\bigcup \mathsf{toMerge}^{(4)} \subseteq \mathscr{K} \land$
        $\forall \{[x], [y]\} \in \mathsf{toMerge}^{(4)}.\ \ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(4)}, [x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(4)}, [y])$
    Loop Invariant: $\mathrm{dom}(\mathcal{D}) \uplus \mathsf{dets} \subseteq \mathrm{dom}(\mathcal{D}^{(4)})$
    Loop Invariant: $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(4)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(4)}, \mathsf{toMerge}^{(4)})$
(7) for $[x] \in \mathsf{dets}^{(4)}$ do
        $(\mathscr{K}^{(n)}, \mathsf{cstrts}', \mathsf{toMerge}') \leftarrow \texttt{PropagateDeterminacy}(\mathscr{K}^{(4)}, [x], \Theta^{(4)}(\mathcal{D}^{(4)}([x])))$
        $\mathsf{cstrts}^{(n)} \leftarrow \mathsf{cstrts}^{(4)} \cup \mathsf{cstrts}'$
        $\mathsf{toMerge}^{(n)} \leftarrow \mathsf{toMerge}^{(4)} \cup \mathsf{toMerge}'$
        $(\mathscr{K}^{(4)}, \mathsf{cstrts}^{(4)}, \mathsf{toMerge}^{(4)}) \leftarrow (\mathscr{K}^{(n)}, \mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)})$
(8) return $(\mathscr{K}^{(4)}, \mathsf{cstrts}^{(4)}, \mathsf{toMerge}^{(4)})$
```

Algorithm 15: Propagating the determinacy of an EC within the heads of affected ECs

```
PropagateHeadSubst ($\mathscr{K}, [a] : EC_H, T : \mathcal{T}_{EC}$, headSubst : $\mathscr{P}(T_H)$) :
        ($\mathscr{K}'$, cstrts : $\mathscr{P}(\mathcal{C})$, toMerge : $\mathscr{P}(\binom{EC_H}{2})$, dets : $\mathscr{P}(EC_H)$)
    Precondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}) \wedge [a] \in \mathscr{K} \wedge \mathcal{Q}\texttt{-Find}(\mathcal{Q}, [a]) = [a] \wedge$
        $\varsigma(T){\downarrow} \wedge \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, [a]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, T) \wedge$
        $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T)$
    Precondition: headSubst $\# \operatorname{dom}(T_R) \wedge$ headSubst $\subseteq \operatorname{dom}(\Theta) \wedge$
        $\forall \tilde{h} \in$ headSubst. $\neg\mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(\tilde{h})) \wedge \mathcal{T}_{EC}\texttt{-InHead}(\mathcal{Q}, [a], \Theta(\tilde{h}))$
    Postcondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}') \wedge \varsigma = \varsigma' \wedge \mathcal{Q} = \mathcal{Q}' \wedge T_R = T'_R \wedge G_{\preceq} = G'_{\preceq}$
    Postcondition: $\operatorname{dom}(\Theta') \cup$ headSubst $= \operatorname{dom}(\Theta) \wedge$
        $\forall \tilde{h} \in \operatorname{dom}(\Theta) \setminus$ headSubst. $\Theta(\tilde{h}) = \Theta'(\tilde{h})$
    Postcondition: $\operatorname{dom}(\mathcal{D}) \uplus$ dets $\subseteq \operatorname{dom}(\mathcal{D}')$
    Postcondition: $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge$ cstrts $\curlywedge M(\mathscr{K}'$, toMerge)
    $\mathscr{K}^{(1)} \leftarrow \mathscr{K}$
    headSubst$^{(1)} \leftarrow$ headSubst
    cstrts, toMerge, dets $\leftarrow \emptyset$
    Remark: We define headSubst$^{(1),c}$ as an alias for headSubst $\setminus$ headSubst$^{(1)}$.
    Loop Invariant: $\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(1)}) \wedge \varsigma = \varsigma^{(1)} \wedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)} \wedge$
        $G_{\preceq} = G_{\preceq}^{(1)}$
    Loop Invariant: $\operatorname{dom}(\Theta^{(1)}) \cup$ headSubst$^{(1),c} = \operatorname{dom}(\Theta) \wedge$
        $\forall \tilde{h} \in \operatorname{dom}(\Theta) \setminus$ headSubst$^{(1),c}$. $\Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h}) \wedge$
        headSubst$^{(1)} \subseteq \operatorname{dom}(\Theta^{(1)})$
    Loop Invariant: $\operatorname{dom}(\mathcal{D}) \uplus$ dets $\subseteq \operatorname{dom}(\mathcal{D}^{(1)})$
    Loop Invariant: $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge$ cstrts $\curlywedge M(\mathscr{K}^{(1)}$, toMerge)
(1)     while $\exists h \in$ headSubst$^{(1)}$ do
            headSubst$^{(n)} \leftarrow$ headSubst$^{(1)} \setminus \{h\}$
(2)         if $\mathcal{T}_{EC}\texttt{-IsDNF}(\Theta^{(1)}(h))$ then
                Note that we do not work with $[\![ [a] \mapsto T ]\!] \Theta^{(1)}(h)$ as we are only interested in substituting the $[a]$'s
                  appearing in head positions, not all occurrences of $[a]$.
(2a)            $S^{(1)} \leftarrow \mathcal{T}_{EC}\texttt{-ApplyHeadSubstitution}(\mathscr{K}^{(1)}, \Theta^{(1)}(h), [a], T)$
                It is not clear whether $S^{(1)}$ always satisfy the invariant or not. If not, we just skip.
(2b)            if $\neg\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S^{(1)})$ then
                    headSubst$^{(1)} \leftarrow$ headSubst$^{(n)}$
                    continue
(2c)            $S^{(2)} \leftarrow \mathcal{T}_{EC}\texttt{-SimplifyDNF}(\mathscr{K}^{(1)}, S^{(1)})$
(2d)            if $S^{(2)} \in EC_H$ then
                    The simplification yielded a simple $EC_H$; that is, $S^{(2)}$ is of the form $[x]$. In that case, we
                      restrain ourselves from updating $h$ with $S^{(2)}$: it is better to merge $\mathcal{R}^{(1)}(h)$ with $S^{(2)}$. We also
                      ensure that we do not try to merge a class with itself to keep toMerge well-formed by being a
                      set of unordered pairs.
(2d.i)              if $\mathcal{Q}\texttt{-Find}(\mathscr{K}^{(1)}, S^{(2)}) \neq \mathcal{R}^{(1)}(h)$ then
                        toMerge$^{(n)} \leftarrow$ toMerge $\cup \{S^{(2)}, \mathcal{R}^{(1)}(h)\}$
                        toMerge $\leftarrow$ toMerge$^{(n)}$
                    Otherwise, we give up and continue with the next $h$.
(2e)            else if $\mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}^{(1)}, S^{(2)})$ then
                    ($\mathscr{K}^{(n)}$, cstrts', toMerge', dets') $\leftarrow \texttt{UpdateMemberDetermined}(\mathscr{K}^{(1)}, h, S^{(2)})$
                    (cstrts$^{(n)}$, toMerge$^{(n)}$, dets$^{(n)}$) $\leftarrow$ (cstrts $\cup$ cstrts', toMerge $\cup$ toMerge', dets $\cup$ dets')
                    ($\mathscr{K}^{(1)}$, cstrts, toMerge, dets) $\leftarrow$ ($\mathscr{K}^{(n)}$, cstrts$^{(n)}$, toMerge$^{(n)}$, dets$^{(n)}$)
(2f)            else
                    $\mathscr{K}^{(n)} \leftarrow \texttt{UpdateMember}(\mathscr{K}^{(1)}, h, S^{(2)})$
                    $\mathscr{K}^{(1)} \leftarrow \mathscr{K}^{(n)}$
```

```
(3)         else
                $S \leftarrow \mathcal{T}_{EC}$-ApplyHeadSubstitution($\mathcal{K}^{(1)}, \Theta^{(1)}(h), [a], T$)
                if $\mathcal{T}_{EC}$-IsDet($\mathcal{K}^{(1)}, S$) then
                    $(\mathcal{K}^{(n)}, \mathsf{cstrts}', \mathsf{toMerge}', \mathsf{dets}') \leftarrow$ UpdateMemberDetermined($\mathcal{K}^{(1)}, h, S$)
                    $\mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)}, \mathsf{dets}^{(n)} \leftarrow \mathsf{cstrts} \cup \mathsf{cstrts}', \mathsf{toMerge} \cup \mathsf{toMerge}', \mathsf{dets} \cup \mathsf{dets}'$
                    $(\mathcal{K}^{(1)}, \mathsf{cstrts}, \mathsf{toMerge}, \mathsf{dets}) \leftarrow (\mathcal{K}^{(n)}, \mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)}, \mathsf{dets}^{(n)})$
                else
                    $\mathcal{K}^{(n)} \leftarrow$ UpdateMember($\mathcal{K}^{(1)}, h, S$)
                    $\mathcal{K}^{(1)} \leftarrow \mathcal{K}^{(n)}$
            $\mathsf{headSubst}^{(1)} \leftarrow \mathsf{headSubst}^{(n)}$
(4)     return $(\mathcal{K}^{(1)}, \mathsf{cstrts}, \mathsf{toMerge}, \mathsf{dets})$
```

Algorithm 16: Propagating the determinacy of an EC within DNFs

```
    PropagateDNFRefresh ($\mathcal{K}$, refreshDNF : $\mathcal{P}(T_H)$) :
                        ($\mathcal{K}'$, cstrts : $\mathcal{P}(\mathcal{C})$, toMerge : $\mathcal{P}(\binom{EC_H}{2})$, dets : $\mathcal{P}(EC_H)$)
        Precondition: $\mathcal{K}$-Valid($\mathcal{K}$) $\wedge$ refreshDNF $\#$ Im($T_R$) $\wedge$ refreshDNF $\subseteq$ dom($\Theta$) $\wedge$
            $\forall \tilde{h} \in$ refreshDNF. $\mathcal{T}_{EC}$-IsDNF($\Theta(\tilde{h})$)
        Postcondition: $\mathcal{K}$-Valid($\mathcal{K}'$) $\wedge \varsigma = \varsigma' \wedge \mathcal{Q} = \mathcal{Q}' \wedge T_R = T'_R \wedge G_{\preceq} = G'_{\preceq}$
        Postcondition: dom($\Theta'$) $\cup$ refreshDNF $=$ dom($\Theta$) $\wedge$
            $\forall \tilde{h} \in$ dom($\Theta$) $\setminus$ refreshDNF. $\Theta(\tilde{h}) = \Theta'(\tilde{h})$
        Postcondition: dom($\mathcal{D}$) $\uplus$ dets $\subseteq$ dom($\mathcal{D}'$)
        Postcondition: $\mathcal{K} \Vdash \mathcal{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathcal{K}', \mathsf{toMerge})$
        $\mathcal{K}^{(1)} \leftarrow \mathcal{K}$
        $\mathsf{refreshDNF}^{(1)} \leftarrow \mathsf{refreshDNF}$
        $\mathsf{cstrts}, \mathsf{toMerge}, \mathsf{dets} \leftarrow \emptyset$
        Remark: We define $\mathsf{refreshDNF}^{(1),c}$ as an alias for $\mathsf{refreshDNF} \setminus \mathsf{refreshDNF}^{(1)}$.
        Loop Invariant: $\mathcal{K}$-Valid($\mathcal{K}^{(1)}$) $\wedge \varsigma = \varsigma^{(1)} \wedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)} \wedge$
            $G_{\preceq} = G_{\preceq}^{(1)}$
        Loop Invariant: dom($\Theta^{(1)}$) $\cup$ $\mathsf{refreshDNF}^{(1),c}$ $=$ dom($\Theta$) $\wedge$
            $\forall \tilde{h} \in$ dom($\Theta$) $\setminus$ $\mathsf{refreshDNF}^{(1),c}$. $\Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h})$ $\wedge$
            $\mathsf{refreshDNF}^{(1)} \subseteq$ dom($\Theta^{(1)}$)
        Loop Invariant: dom($\mathcal{D}$) $\uplus$ dets $\subseteq$ dom($\mathcal{D}^{(1)}$)
        Loop Invariant: $\mathcal{K} \Vdash \mathcal{K}^{(1)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathcal{K}^{(1)}, \mathsf{toMerge})$
(1)     while $\exists h \in \mathsf{refreshDNF}^{(1)}$ do
            $\mathsf{refreshDNF}^{(n)} \leftarrow \mathsf{refreshDNF}^{(1)} \setminus \{h\}$
            assert $\mathcal{T}_{EC}$-IsDNF($\Theta^{(1)}(h)$)
(2)         $S \leftarrow \mathcal{T}_{EC}$-SimplifyDNF($\mathcal{K}^{(1)}, \Theta^{(1)}(h)$)
(3)         if $S \in EC_H$ then
                if $\mathcal{Q}$-Find($\mathcal{K}^{(1)}, S$) $\neq \mathcal{R}^{(1)}(h)$ then
                    $\mathsf{toMerge}^{(n)} \leftarrow \mathsf{toMerge} \cup \{S, \mathcal{R}^{(1)}(h)\}$
                    $\mathsf{toMerge} \leftarrow \mathsf{toMerge}^{(n)}$
(4)         else if $\mathcal{T}_{EC}$-IsDet($\mathcal{K}^{(1)}, S$) then
                $(\mathcal{K}^{(n)}, \mathsf{cstrts}', \mathsf{toMerge}', \mathsf{dets}') \leftarrow$ UpdateMemberDetermined($\mathcal{K}^{(1)}, h, S$)
                $(\mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)}, \mathsf{dets}^{(n)}) \leftarrow \mathsf{cstrts} \cup \mathsf{cstrts}', \mathsf{toMerge} \cup \mathsf{toMerge}', \mathsf{dets} \cup \mathsf{dets}'$
                $(\mathcal{K}^{(1)}, \mathsf{cstrts}, \mathsf{toMerge}, \mathsf{dets}) \leftarrow (\mathcal{K}^{(n)}, \mathsf{cstrts}^{(n)}, \mathsf{toMerge}^{(n)}, \mathsf{dets}^{(n)})$
```

```
(5)      else
             $\mathscr{K}^{(n)} \leftarrow$ UpdateMember($\mathscr{K}^{(1)}, h, S$)
             $\mathscr{K}^{(1)} \leftarrow \mathscr{K}^{(n)}$
         refreshDNF$^{(1)} \leftarrow$ refreshDNF$^{(n)}$
(6)  return ($\mathscr{K}^{(1)}$, cstrts, toMerge, dets)
```

Algorithm 17: Propagating the determinacy of an EC within potentially affected ECs

```
PropagateTrySubst ($\mathscr{K}$, trySubst : $T_H \rightharpoonup \mathscr{P}(\mathcal{T}_{EC}), T : \mathcal{T}_{EC}$) :
        ($\mathscr{K}'$, cstrts : $\mathscr{P}(\mathcal{C})$, toMerge : $\mathscr{P}(\binom{EC_H}{2})$, dets : $\mathscr{P}(EC_H)$)
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$) $\wedge \varsigma(T)\downarrow \wedge \mathcal{T}_{EC}$-in-$\Theta$-Inv($T$)
    Precondition: dom(trySubst) $\mathrel{\#}$ Im($T_R$) $\wedge$ dom(trySubst) $\subseteq$ dom($\Theta$)
        $\forall \tilde{h} \in$ dom(trySubst). $\mathcal{T}_{EC}$-IsAbsAppTycon($\Theta(\tilde{h})$) $\wedge$
        $\forall U \in \bigcup$Im(trySubst). [$\varsigma(U)\downarrow \wedge \mathcal{T}_{EC}$-IsAbsAppTycon($U$) $\wedge \mathcal{T}_{EC}$-in-$\Theta$-Inv($U$) $\wedge$
            $\mathcal{T}_{EC}$-kind($\mathscr{K}, U$) $= \mathcal{T}_{EC}$-kind($\mathscr{K}, T$)]
    Postcondition: $\mathscr{K}$-Valid($\mathscr{K}'$) $\wedge \varsigma = \varsigma' \wedge \mathcal{Q} = \mathcal{Q}' \wedge T_R = T'_R \wedge G_{\preceq} = G'_{\preceq}$
    Postcondition: dom($\Theta'$) $\cup$ dom(trySubst) = dom($\Theta$) $\wedge$
        $\forall \tilde{h} \in$ dom($\Theta$) $\setminus$ dom(trySubst). $\Theta(\tilde{h}) = \Theta'(\tilde{h})$
    Postcondition: dom($\mathcal{D}$) $\uplus$ dets $\subseteq$ dom($\mathcal{D}'$)
    Postcondition: $\mathscr{K} \Vdash \mathscr{K}' \curlywedge \bigcurlywedge$ cstrts $\curlywedge M(\mathscr{K}'$, toMerge)
    $\mathscr{K}^{(1)} \leftarrow \mathscr{K}$
    trySubst$^{(1)} \leftarrow$ trySubst
    cstrts, toMerge, dets $\leftarrow \emptyset$
    Remark: We define trySubst$^{(1),c}$ as an alias for trySubst $\setminus$ trySubst$^{(1)}$.
    Loop Invariant: $\mathscr{K}$-Valid($\mathscr{K}^{(1)}$) $\wedge \varsigma = \varsigma^{(1)} \wedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)} \wedge$
        $G_{\preceq} = G_{\preceq}^{(1)}$
    Loop Invariant: dom($\Theta^{(1)}$) $\cup$ dom(trySubst$^{(1),c}$) = dom($\Theta$) $\wedge$
        $\forall \tilde{h} \in$ dom($\Theta$) $\setminus$ dom(trySubst$^{(1),c}$). $\Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h})$ $\wedge$
        dom(trySubst$^{(1)}$) $\subseteq$ dom($\Theta^{(1)}$)
    Loop Invariant: dom($\mathcal{D}$) $\uplus$ dets $\subseteq$ dom($\mathcal{D}^{(1)}$)
    Loop Invariant: $\mathscr{K} \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge$ cstrts $\curlywedge M(\mathscr{K}^{(1)}$, toMerge)
(1)  while $\exists (h, \bar{U}) \in$ trySubst$^{(1)}$ do
         trySubst$^{(n)} \leftarrow$ trySubst$^{(1)} \setminus \{(h, \bar{U})\}$
(2)      for $U \in \bar{U}$ do
(3)          $S \leftarrow \mathcal{T}_{EC}$-TryApplyHeadSubstitution($\mathscr{K}^{(1)}, \Theta^{(1)}(h), U, T$)
(4)          if $S \neq NIL \wedge \mathcal{T}_{EC}$-IsDet($\mathscr{K}^{(1)}, S$) then
(4a)             ($\mathscr{K}^{(n)}$, cstrts$'$, toMerge$'$, dets$'$) $\leftarrow$ UpdateMemberDetermined($\mathscr{K}^{(1)}, h, S$)
                 cstrts$^{(n)}$, toMerge$^{(n)}$, dets$^{(n)} \leftarrow$ cstrts $\cup$ cstrts$'$, toMerge $\cup$ toMerge$'$, dets $\cup$ dets$'$
                 $\mathscr{K}^{(1)}$, cstrts, toMerge, dets $\leftarrow \mathscr{K}^{(n)}$, cstrts$^{(n)}$, toMerge$^{(n)}$, dets$^{(n)}$
                 break
         trySubst$^{(1)} \leftarrow$ trySubst$^{(n)}$
(5)  return $\mathscr{K}^{(1)}$, cstrts, toMerge, dets
```

Algorithm 18: Collecting all ECs containing the determined EC in a head position

```
GatherAffected ($\mathscr{K}$, $[b] : EC_H$, $\mathsf{processedECs} : \mathscr{P}(EC_H)$) :
        ($\mathsf{headSubst} : \mathscr{P}(T_H)$, $\mathsf{refreshDNF} : \mathscr{P}(T_H)$, $\mathsf{processedECs}' : \mathscr{P}(EC_H)$)
    Precondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}) \wedge \{[b]\} \cup \mathsf{processedECs} \subseteq \mathrm{dom}(\mathcal{M}) \wedge \mathcal{Q}\texttt{-Find}(\mathcal{Q}, [b]) = [b]$
    Postcondition: $\mathsf{headSubst} \# \mathrm{Im}(T_R) \wedge \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta) \wedge$
        $\forall \tilde{h} \in \mathsf{headSubst}.\ \neg \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(\tilde{h})) \wedge \mathcal{T}_{EC}\texttt{-InHead}(\mathcal{Q}, [b], \Theta(\tilde{h}))$
    Postcondition: $\mathsf{refreshDNF} \# \mathrm{Im}(T_R) \wedge \mathsf{refreshDNF} \subseteq \mathrm{dom}(\Theta) \wedge$
        $\forall \tilde{h} \in \mathsf{refreshDNF}.\ \mathcal{T}_{EC}\texttt{-IsDNF}(\Theta(\tilde{h}))$
    Postcondition: $\mathsf{processedECs} \subseteq \mathsf{processedECs}' \subseteq \mathrm{dom}(\mathcal{M})$
(1) if $[b] \in \mathsf{processedECs}$ then
        return $(\emptyset, \emptyset, \mathsf{processedECs})$
    $\mathsf{headSubst}, \mathsf{refreshDNF} \leftarrow \emptyset$
    $\mathsf{processedECs}^{(1)} \leftarrow \mathsf{processedECs}$
    Iterate over all $h$ where $[b]$ appears.
    To ease the proof correctness, we skip type representatives. We argue that correctness is ensured even if we
      did not skip these but it is just easier to add that extra filter.
    Loop Invariant: $\mathsf{headSubst} \# \mathrm{Im}(T_R) \wedge \mathsf{headSubst} \subseteq \mathscr{K}$
        $\forall \tilde{h} \in \mathsf{headSubst}.\ \neg \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(\tilde{h})) \wedge \mathcal{T}_{EC}\texttt{-InHead}(\mathcal{Q}, [b], \Theta(\tilde{h}))$
    Loop Invariant: $\mathsf{refreshDNF} \# \mathrm{Im}(T_R) \wedge \mathsf{refreshDNF} \subseteq \mathscr{K}$
        $\forall \tilde{h} \in \mathsf{refreshDNF}.\ \mathcal{T}_{EC}\texttt{-IsDNF}(\Theta(\tilde{h}))$
    Loop Invariant: $\mathsf{processedECs} \subseteq \mathsf{processedECs}^{(1)} \subseteq \mathrm{dom}(\mathcal{M})$
(2) for $h \in \{h : ([b], h) \in E_{EC}\} \setminus \mathrm{Im}(T_R)$ do
        $\mathsf{headSubst}^{(n)}, \mathsf{refreshDNF}^{(n)}, \mathsf{processedECs}^{(n)} \leftarrow \mathsf{headSubst}, \mathsf{refreshDNF}, \mathsf{processedECs}^{(1)}$
        That is, if $h$ is the determined type of its EC, where $[b]$ appears. We propagate with $NH$, as $[b]$ must
          appear in non-head position since $h$ was already the determined member before we started the
          propagation process.
(2a)    if $\mathcal{D}(\mathcal{R}(h)) = h$ then
            Ignoring the returned $\mathsf{headSubst}'$ because $[b]$ is not in a head position in these $h'$.
            $(\_, \mathsf{refreshDNF}', \mathsf{processedECs}') \leftarrow$
                $\texttt{GatherAffected}(\mathscr{K}, \mathcal{R}(h), NH, \mathsf{processedECs}^{(1)} \cup \{[b]\})$
            $\mathsf{refreshDNF}^{(n)} \leftarrow \mathsf{refreshDNF} \cup \mathsf{refreshDNF}'$
            $\mathsf{processedECs}^{(n)} \leftarrow \mathsf{processedECs}^{(1)} \cup \mathsf{processedECs}'$
(2b)    else if $\neg \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(h))$ then
(2b.i)      if $\mathcal{T}_{EC}\texttt{-InHead}(\mathcal{Q}, [b], \Theta(h))$ then
                $\mathsf{headSubst}^{(n)} \leftarrow \mathsf{headSubst} \cup \{h\}$
(2b.ii)     else if $\mathcal{T}_{EC}\texttt{-IsDNF}(\Theta(h))$ then
                $\mathsf{refreshDNF}^{(n)} \leftarrow \mathsf{refreshDNF} \cup \{h\}$
        $\mathsf{headSubst}, \mathsf{refreshDNF}, \mathsf{processedECs}^{(1)} \leftarrow \mathsf{headSubst}^{(n)}, \mathsf{refreshDNF}^{(n)}, \mathsf{processedECs}^{(n)}$
(3) return $(\mathsf{headSubst}, \mathsf{refreshDNF}, \mathsf{processedECs})$
```

Algorithm 19: Collecting all ECs that may contain the determined EC in a head position

```
      GatherPotentiallyAffected ($\mathscr{K}$, $[a]$)
          Precondition: $\mathscr{K}$-Valid($\mathscr{K}$) $\land$ $[a] \in \mathscr{K}$
          Postcondition: $\mathrm{dom}(\mathsf{trySubst}) \# \mathrm{Im}(T_R) \land \mathrm{dom}(\mathsf{trySubst}) \subseteq \mathrm{dom}(\Theta)$
              $\forall \tilde{h} \in \mathrm{dom}(\mathsf{trySubst}).$ $\mathcal{T}_{EC}$-IsAbsAppTycon($\Theta(\tilde{h})$) $\land$
              $\forall U \in \bigcup \mathrm{Im}(\mathsf{trySubst}).$ $[\varsigma(U)\downarrow \land$ $\mathcal{T}_{EC}$-IsAbsAppTycon($U$) $\land$ $\mathcal{T}_{EC}$-in-$\Theta$-Inv($U$) $\land$
                  $\mathcal{T}_{EC}$-kind($\mathscr{K}, U$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}, [a]$)$]$
          trySubst $\leftarrow \emptyset$
(1)       Loop Invariant: Analogous to postcondition
          for $h \in \mathcal{M}([a])$ do
              match $\Theta(h)$ :
                  Note that we have $\mathcal{T}_{EC}$-IsAbsAppTycon($\Theta(h)$). We pattern match on $\Theta(h)$ in order to extract the
                   head symbol $F$.
(2)               case $[\vec{v}\vec{X} \lhd B]$ =>> $F[\vec{S}]$ or $F[\vec{S}]$ :
                      trySubst' $\leftarrow$ trySubst
(2a)                  Here, all $h'$ are abstract type constructor application (i.e. applied $F$) by validity of $\mathscr{K}$.
                      for $h' \in \{h' : (F, h') \in E_{\mathcal{S}}\} \setminus (\mathrm{Im}(T_R) \cup \mathcal{M}([a]))$ do
(2a.i)                    if $h' \in \mathrm{dom}(\mathsf{trySubst})$ then
                              trySubst' $\leftarrow$ trySubst'$[h' \mapsto \{\Theta(h)\} \cup \mathsf{trySubst}'(h')]$
(2a.ii)                   else
                              trySubst' $\leftarrow$ trySubst'$[h' \mapsto \{\Theta(h)\}]$
                      trySubst$^{(n)}$ $\leftarrow$ trySubst'
                  otherwise :
                      pass
              trySubst $\leftarrow$ trySubst$^{(n)}$
(3)       return trySubst
```

5.3.5.6 Compaction phase for the running examples

We give three examples showcasing the compaction phase for listings 5-7 based on the runs of 5.3.1.1, 5.3.1.2 and 5.3.1.3. We omit the details of the compaction phase for the run 5.3.1.4 because the example covering the run 5.3.1.3 is quite extensive.

Example 5.3.5.1 (Compactions for listing 5)**.** `Compact` is called twice, sequentially. One call is done to compact $X \preceq$ `Int` and another one to compact `Int` $\preceq X$. Assuming $X \preceq$ `Int` is assimilated into $\mathscr{K}$ first, we get:

1. We create the ECs $[x]$ and $[i]$ for X and `Int` respectively. The $[x]$ and $[i]$ ECs respectively contain X and `Int` as their sole members. Furthermore, `Int` is set as the determined type for $[i]$.
2. Through `TryAddInequality`, we add an edge from $[x]$ to $[i]$ in $G_{\preceq}$; the returned cstrts and toMerge are empty.
3. The merge loop is not entered; we return without any new constraints.

Compacting `Int` $\preceq X$ gives:

1. $\mathcal{T}$-`FindOrCreateEC` manages to retrieve the EC for `Int` and X created earlier. In particular, it calls $\mathcal{T}_{EC}$-`FindOrCreateEC` which finds the ECs at line (1a.i).
2. We attempt to form a cycle by adding an edge from $[i]$ to $[x]$. As such, `TryAddInequality` does not update $G_{\preceq}$ and returns $\{\{[i], [x]\}\}$ for toMerge.
3. The merge loop picks $\{[i], [x]\}$ from toMerge and calls `Merge`. A propagation of determinacy is done for $[x]$ with $T =$ `Int` but does not yield anything new because $[x]$ does not appear anywhere. The

merge of $[i]$ and $[x]$ proceeds with the call to `MergeHelper` in a straightforward manner. There are no other ECs to merge and no new constraints.

4. We return from compaction with no new constraints.

Example 5.3.5.2 (Compactions for listing 6)**.** We assume $\texttt{Inv}[pX]\&\texttt{Inv}[\texttt{String}] \preceq \texttt{Inv}[X]\&Y$ is assimilated into $\mathcal{K}$ first:

1. The EC creation for $\texttt{Inv}[pX]$ & $\texttt{Inv}[\texttt{String}]$ proceeds as follows:
   (a) In $\mathcal{T}$-`FindOrCreateEC`, case (5) is matched. $\texttt{Inv}[pX]$ and $\texttt{Inv}[\texttt{String}]$ both match case (5a.i) resulting in the creation of the ECs for pX and `String` which we refer to as $[px]$ and $[s]$ respectively.
   (b) The $\mathcal{T}_{EC}$-`SimplifyDNF` call at (5b) yields the identity.
   (c) We call $\mathcal{T}_{EC}$-`FindOrCreateEC` with $\texttt{Inv}\big[[px]\big]$ & $\texttt{Inv}\big[[s]\big]$ as argument.
   (d) $\mathcal{T}_{EC}$-`FindOrCreateEC` does not find any EC for the given type and calls $\mathcal{T}_{EC}$-`CreateEC` to create an EC containing $\texttt{Inv}\big[[px]\big]\&\texttt{Inv}\big[[s]\big]$. We refer to the created EC as $[a]$. Due to $\texttt{Inv}\big[[px]\big]\&\texttt{Inv}\big[[s]\big]$ not being determined, $[a]$ does not have an associated determined type in $\mathcal{D}$.
   (e) $\mathcal{T}$-`FindOrCreateEC` eventually returns with the updated $\mathcal{K}$ and $[a]$.
2. The EC creation for $\texttt{Inv}[X]$ & Y proceeds similarly. We refer to its associated EC as $[b]$.
3. We add an edge from $[a]$ to $[b]$ in $G_{\preceq}$; the returned cstrts and $\mathsf{toMerge}$ are empty.
4. We do not enter the merge loop and return with an empty set of constraints.

The compaction of $\texttt{Inv}[X]$ & Y and $\texttt{Inv}[pX]$ & $\texttt{Inv}[\texttt{String}]$ is similar to the previous example, where adding an edge from $[b]$ to $[a]$ would cause a cycle. The merging of $[a]$ and $[b]$ proceeds similarly as well (except that there are no propagation of determinacy since $[a]$ and $[b]$ are not determined).

We however discuss the retrieval of the EC of $\texttt{Inv}[pX]$ & $\texttt{Inv}[\texttt{String}]$ by $\mathcal{T}$-`FindOrCreateEC`:

1. As in the first call, case (5) in $\mathcal{T}$-`FindOrCreateEC` is matched with $\texttt{Inv}[pX]$ and $\texttt{Inv}[\texttt{String}]$ both matching case (5a.i). $\mathcal{T}_{EC}$-`FindOrCreateEC` retrieves the EC $[px]$ and $[s]$ for pX and `String` respectively through the loop at (1a).
2. The $\mathcal{T}_{EC}$-`SimplifyDNF` call at (5b) yields the identity as well.
3. We call $\mathcal{T}_{EC}$-`FindOrCreateEC` with $\texttt{Inv}\big[[px]\big]$ & $\texttt{Inv}\big[[s]\big]$, which finds $[a]$ through the loop at (1a) too.
4. $\mathcal{T}$-`FindOrCreateEC` returns $\mathcal{K}$ unchanged and $[a]$.

$\mathcal{T}$-`FindOrCreateEC` finds the EC of $\texttt{Inv}[X]$ & Y analogously.

Example 5.3.5.3 (Compactions for listing 7, run (i))**.** We follow the calls to `Compact` in the order they appear in example 5.3.1.3.

Compaction of $[Z] \texttt{=>>} F[Z] \preceq [Z] \texttt{=>>} \mathit{Inv2}[Z, Y]$ & X**.**
The compaction proceeds as follows:

1. We first need to create an EC containing $[Z] \texttt{=>>} F[Z]$ with $\mathcal{T}$-`FindOrCreateEC`:
   (a) We match case (6). Because the bounds are trivial, we get $B' = \{Z \mapsto (\bot, \top)\}$ at (6a) We then continue with the call to $\mathcal{T}_{EC}$-`FindOrCreateEC`.
   (b) $\mathcal{T}_{EC}$-`FindOrCreateEC` delegates the EC creation to $\mathcal{T}_{EC}$-`CreateEC`.
   (c) An EC is created containing $[Z] \texttt{=>>} F[Z]$, which is not determined. We refer to this EC as $[f]$.
   (d) $\mathcal{T}$-`FindOrCreateEC` returns an updated $\mathcal{K}$ and $[f]$.
2. We then create an EC for $[Z] \texttt{=>>} \texttt{Inv2}[Z, Y]$ & X:
   (a) We match case (7). We similarly get $B' = \{Z \mapsto (\bot, \top)\}$ and keep on with the recursive call at (7b).

        i. $\texttt{Inv2}[Z, Y] \,\&\, X$ matches the case (5). Within the loop at (5a), $\texttt{Inv2}[Z, Y]$ matches (5a.i) and X matches (5a.ii). Starting with $\texttt{Inv2}[Z, Y]$, we recursively call $\mathcal{T}$`-FindOrCreateECVec` with Z and Y. Because Z is bound to the enclosing scope, we match (1) and return Z. For Y, an EC which we refer to as $[y]$ is created.
        We similarly create an EC for X, referred to as $[x]$.
        ii. The DNF simplification at (5b) gives the identity.
        iii. Because we are in a head position and under an enclosing scope, we match (5d) and return $\texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]$.
    (b) The recursive call yields $\texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]$. We go on with the call to $\mathcal{T}_{EC}$`-FindOrCreateEC` with $[Z] \texttt{=>>} \texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]$ and empty bounds as arguments.
    (c) $\mathcal{T}_{EC}$`-FindOrCreateEC` delegates the EC creation to $\mathcal{T}_{EC}$`-CreateEC`, creating an EC $[a]$ containing $[Z] \texttt{=>>} \texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]$.
    (d) $\mathcal{T}$`-FindOrCreateEC` returns an updated $\mathscr{K}$ and $[a]$.
3. At this point, we have four ECs:

$$\begin{array}{ll} [a] : \{[Z] \texttt{=>>} \texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]\} & [x] : \{X\} \\ [f] : \{[Z] \texttt{=>>} F[Z]\} & [y] : \{Y\} \end{array}$$

   None of these ECs are determined.
4. We add an edge from $[f]$ to $[a]$ in $G_{\preceq}$; the returned cstrts and toMerge are empty.
5. We do not enter the merge loop and return with an empty set of constraints.

Compaction of $[Z] \texttt{=>>} \texttt{Inv2}[Z, pY] \ \&\ \texttt{Inv}[pY] \preceq [Z] \texttt{=>>} pF[Z]$**.**
The compaction proceeds similarly and is thus omitted. We now have the following ECs with their corresponding members:

$$\begin{array}{ll} [a] : \{[Z] \texttt{=>>} \texttt{Inv2}\big[Z, [y]\big] \ \&\ [x]\} & [b] : \{[Z] \texttt{=>>} \texttt{Inv2}\big[Z, [py]\big] \ \&\ \texttt{Inv}\big[[py]\big]\} \\ [f] : \{[Z] \texttt{=>>} F[Z]\} & [py] : \{pY\} \\ [x] : \{X\} & [pf] : \{[Z] \texttt{=>>} pF[Z]\} \\ [y] : \{Y\} & \end{array}$$

with the edges $([f], [a])$ and $([b], [pf])$ in $G_{\preceq}$.

Compaction of $X \asymp \texttt{Inv}[pX]$**.**
Assuming we are tasked to compact $X \preceq \texttt{Inv}[pX]$ first, we get:

1. $\mathcal{T}$`-FindOrCreateEC` manages to retrieve the EC of X which is $[x]$.
2. $\texttt{Inv}[pX]$ does not have an associated EC: $\mathcal{T}$`-FindOrCreateEC` creates an EC for pX, $[px]$, and creates an EC $[ipx]$ containing $\texttt{Inv}\big[[px]\big]$. $\texttt{Inv}\big[[px]\big]$ is set as the determined type of $[ipx]$.
3. We add an edge from $[x]$ to $[ipx]$ in $G_{\preceq}$; the returned cstrts and toMerge are empty.
4. We do not enter the merge loop and return with no new constraints.

For $\texttt{Inv}[pX] \preceq X$, we obtain:

1. $\mathcal{T}$`-FindOrCreateEC` retrieves the ECs of $\texttt{Inv}[pX]$ and X which are $[ipx]$ and $[x]$ respectively.
2. We attempt to form a cycle by adding an edge from $[ipx]$ to $[x]$. `TryAddInequality` therefore does not update $G_{\preceq}$ and returns $\big\{\{[ipx], [x]\}\big\}$ for toMerge.
3. The merge loop picks $\{[ipx], [x]\}$ from toMerge and calls `Merge`.
4. Because $[ipx]$ is determined and $[x]$ not, a propagation of determinacy is done for $[x]$ with $T = \texttt{Inv}\big[[px]\big]$:

   (a) $[x]$ only appears in $[a]$. The propagation then proceeds by calling `PropagateHeadSubst` with the type handle of the sole member of $[a]$. $[x]$ gets substituted to $\texttt{Inv}\big[[px]\big]$, yielding $[Z] \mathrel{\texttt{=>>}} \texttt{Inv2}\big[Z,[y]\big] \mathbin{\&} \texttt{Inv}\big[[px]\big]$. Then, `UpdateMemberDetermined` update $[a]$ to contain this new type and set that type to be the determined type of $[a]$.
   (b) Since $[a]$ became determined, a propagation of determinacy is done for $[a]$ with $T = [Z] \mathrel{\texttt{=>>}} \texttt{Inv2}\big[Z,[y]\big] \mathbin{\&} \texttt{Inv}\big[[px]\big]$. The propagation is trivial because $[a]$ does not appear anywhere.

5. We resume the merge of $[ipx]$ and $[x]$ and call `MergeHelper`. We assume that $\mathcal{Q}$-`Union` of $[ipx]$ and $[x]$ choses $[x]$ as the representative for the union of these two partitions.
6. We exit the merge loop and return with no new constraints.

We have the following ECs:

$$\begin{array}{ll}
[a] : \{[Z] \mathrel{\texttt{=>>}} \texttt{Inv2}\big[Z,[y]\big] \mathbin{\&} \texttt{Inv}\big[[px]\big]\} & [b] : \{[Z] \mathrel{\texttt{=>>}} \texttt{Inv2}\big[Z,[py]\big] \mathbin{\&} \texttt{Inv}\big[[py]\big]\} \\
[f] : \{[Z] \mathrel{\texttt{=>>}} F[Z]\} & [px] : \{pX\} \\
[x] : \{\texttt{Inv}\big[[px]\big]\} & [py] : \{pY\} \\
[y] : \{Y\} & [pf] : \{[Z] \mathrel{\texttt{=>>}} pF[Z]\}
\end{array}$$

with the edges $([f],[a])$ and $([b],[pf])$ in $G_{\preceq}$. Furthermore, $[a]$, $[b]$ and $[x]$ are determined.

Compaction of $F \asymp pF$***.***

Assuming we first compact $F \preceq pF$, we get:

1. We expand F and pF into $[Z] \mathrel{\texttt{=>>}} F[Z]$ and $[Z] \mathrel{\texttt{=>>}} pF[Z]$ respectively.
2. $\mathcal{T}$-`FindOrCreateEC` retrieves their ECs $[f]$ and $[pf]$.
3. We add an edge from $[f]$ to $[pf]$ in $G_{\preceq}$; the returned `cstrts` and `toMerge` are empty.
4. We do not enter the merge loop and return with no new constraints.

$G_{\preceq}$ records the inequalities $[f] \preceq [a]$, $[b] \preceq [pf]$ and $[f] \preceq [pf]$.

For $pF \preceq F$, we have:

1. $\mathcal{T}$-`FindOrCreateEC` retrieves the ECs $[pf]$ and $[f]$.
2. We attempt to form a cycle by adding an edge from $[pf]$ to $[f]$. `TryAddInequality` does not update $G_{\preceq}$ and returns $\big\{\{[pf],[f]\}\big\}$ for `toMerge`.
3. The merge loop picks $\{[pf],[f]\}$ from `toMerge` and calls `Merge`, which in turn calls `MergeHelper`. We assume that the $\mathcal{Q}$-`Union` (line (1a)) of $[pf]$ and $[f]$ picks $[f]$ as the representative for the union.
   The update made to the $G_{\preceq}$ graph at (6e) (caused by fusing the node $[f]$ and $[pf]$ into $[f]$) creates a path between $[b]$ and $[a]$. Because $[b]$ and $[a]$ are determined, we generate the constraint $[Z] \mathrel{\texttt{=>>}} \texttt{Inv2}[Z,pY] \mathbin{\&} \texttt{Inv}[pY] \preceq [Z] \mathrel{\texttt{=>>}} \texttt{Inv2}[Z,Y] \mathbin{\&} \texttt{Inv}[pX]$ at line (11) [8].
4. We exit the merge loop and return with the above constraint.

Compaction of $Y \asymp pY \curlywedge pX \asymp pY$***.***

The assimilation of $Y \asymp pY \curlywedge pX \asymp pY$ results in four calls to `Compact`. We omit them because they do not produce anything new of interest.

---

[8] $[y]$, $[px]$ and $[px]$ have been substituted to Y, pX and pY respectively through ς leveraging the type representatives from T_R. We did not explicitly track these as it would considerably increase the example length.

5.4 Caveats

In section 2.3, we have mentioned that the rule (Path-&) introduces unsoundness in presence of implicit `null` and explicit casts.

This unsoundness affects the proposed algorithm as well. We start with the `null` problem.

Suppose we are given the following problem:

```scala
// Note: does not compile under Scala 3.0.0
class Box[T](a: T)
class Inv2[S, T](a: Box[S] & Box[T])

def patmat[X, Y](s: Inv2[X, Y], x: X): Y = s match {
    case p: Inv[xy] => x
}
```

We have $s.a : \texttt{Box}[X] \,\&\, \texttt{Box}[Y]$ and would deduce that X and Y are equal. We would thus authorize the given snippet.

However, if pass `patmat new Inv(null)`, we cause unsoundness if we instantiate X and Y to different types as in the following example:

```scala
// Would cause a ClassCastException.
val got: Int = patmat[String, Int](new Inv2(null), "hello")
```

We remark that explicits nulls would forbid the above snippet from compiling and should resolve the problem.

The other issue involves `asInstanceOf` casts. Consider the following snippet:

```scala
// Note: compiles and produces a ClassCastException under Scala 3.0.0
class Inv[T]

def patmat[X, Y](s: Inv[X] & Inv[Y], x: X): Y = s match {
    case p: Inv[xy] => x
}

val inv = new Inv[Any].asInstanceOf[Inv[String] & Inv[Int]]
val got: Int = patmat[String, Int](inv, "hello")
```

We deduce that X and Y are equal as well.

This problem is more general and affects Scala 3.0.0 as well. The underlying cause of this unsoundness is the presence of *phantom types*. A similar issue is tracked under ticket 8430 [9].

5.5 Constraint order-sensitivity

Before concluding this chapter, it is important to point out that the proposed algorithm is sensitive to the order in which the constraints are processed. It is in some sense non-deterministic.

As an example, let us consider a constraint $C_1 = p : R_1$ (where R_1 is a refinement), and another constraint $C_2 = R_1 \preceq R_2$ for some refinement R_2. The subtyping constraint C_2 will be passed to `DeductionIneq`, which will look for a p' in $\mathcal{I}$ that inhabits R_1. If we process C_1 first, `DeductionIneq` will retrieve p, resume the deduction, and potentially derive new constraints. Otherwise, if C_1 is processed after C_2, `DeductionIneq` will give up and abandon the constraint C_2.

We believe that an enhanced version of the algorithm taking care of performing a path typing propagation will still likely be sensitive to the constraint order, as various “give-up” cases may be triggered or avoided depending on the constraints ordering.

---

[9] `https://github.com/lampepfl/dotty/issues/8430`

Chapter 6

Related work

The following two works analyze the GADT problem in Scala's context.

Parreaux and Boruch-Gruszecki [7] have examined foundations for GADTs in Scala and have shown that GADTs can be explained and understood in terms of already present, simpler features such as type members.

Waśko [16] has analyzed the requirements of encoding GADTs within a calculus. He then presented an encoding of GADTs into the pDOT calculus as well as the necessary steps to formally prove the validity of the encoding.

The constraint language presented in chapter 3 has been largely inspired by the constraint language introduced by Pottier and Rémy [11]. They have presented a variant of HM(X) (covered next) where the constraints are interpreted within a model; conjunction and existential quantification are given their usual meaning. They have defined a constraint solver for HM(=) leveraging a standard first-order unification algorithm.

HM(X) has been introduced by Odersky, Sulzmann and Wehr [6]. It is a general framework for Hindley-Milner alike type systems with constraints and is parameterized by a constraint system X. When instantiated to the trivial constraint system – with X set to = – one obtains the Hindley-Milner system. The authors prove that type systems in HM(X) are sound for standard untyped compositional semantics and give a generic type inference algorithm.

The following works consider GADT inference in a different context than ours but are nonetheless interesting in how their authors have approached the problem.

Simonet and Pottier [15] have studied HMG(X) which extends the constraint-based type system HM(X) with (among other features) GADTs. Their settings allow arbitrary constraints, ranging from deep patterns to subtyping. They prove HMG(X) sound and show that the type inference problem can be reduced to constraint solving. Due to the parameterized nature of HMG(X), Simonet and Pottier do not provide a constraint solver. They argue however that any constraint solver is expected to be computationally expensive, especially due to the presence of the implication connective within constraints.

Peyton Jones, Vytiniotis, Weirich and Washburn [8] have specified a language supporting GADTs and user-supplied type annotations. They introduced the key concept of *wobbly types*, allowing to express the uncertainty of incremental type inference algorithms. They proved the type system sound and that it is a conservative extension of Hindley-Milner. Furthermore, the authors have implemented GADTs type inference for the Glasgow Haskell Compiler. A particular point of interest is the fact that wobbly types allow the type inference algorithm to be insensitive to the AST traversal order.

Kennedy and Russo [4] have presented a generalization of the type constraints mechanism of Java and $C^\sharp$ to avoid the need for user-supplied explicit casts. They have formalized the extension for a subset of $C^\sharp$ and proved its soundness.

Chapter 7

Conclusion

In this work, we have presented (without proving) an extension of the pDOT calculus with nominal subtyping and higher-kinded types. This extension allowed us to reason about some constructs that are not present in pDOT, such as class inheritance and higher-kinded abstractions.

Based on the work of François Pottier and Didier Rémy, we have developed a simple constraint language enabling formal reasoning about the GADT inference problem. With this abstraction, we have not only derived useful laws that guided the design of the proposed algorithm but also found some counter-examples of accepted programs that should be rejected [1].

We have presented an algorithm that accumulates constraints into a structure with an assimilation process that may result in discovering further information about the type variables. We have also proved the soundness of the algorithm and provided an incomplete proof of its termination. We have seen some examples where the proposed algorithm is capable of inferring interesting properties that the Scala 3.0.0 compiler does not intercept.

7.1 Future work

As we have seen throughout this report, there is still room for improvement.

On the formalization side, some definitions remain wobbly and not-so-formal. Another point of concern is basing a substantial amount of our work on the assumptions of the subtyping rules as stated in section 2.3. In particular, the rule (PATH-&) is questionable: we have seen two unsound issues in section 5.4 – though we deem these "reasonable".

On the algorithmic side, some constructs are not used to their full potential. For instance, path-dependent types are not entirely leveraged. Furthermore, the support of term variables is quite lackluster – especially for bound term variables introduced by methods. The first step towards this goal would be to introduce proper equivalence classes for term variables – akin to equivalence classes of type variables. These equivalence classes would also allow abstracting over term variables, just like higher-kinded equivalence classes. However, such a feature would likely introduce a considerable amount of complexity and may not be worthwhile.

Another important point to discuss is the fact that we strip all refinements before handing them over to the compaction phase. To improve the support of refinements, we would need to, in particular, adapt the family of the EC processing functions ($\mathcal{T}$-`FindOrCreateEC`, $\mathcal{T}_{EC}$-`FindOrCreateEC`, and so on). Creating and finding ECs of refinements may need implementing ECs for bound term variables as well to be satisfactory.

Besides, there are still some low-hanging fruits. We discuss two of them here. The first enhancement is to propagate a path inhabitation constraint $p : T$ to the upper bounds of T. The other relatively straightforward enhancement is to propagate subtyping relationship to other equivalence classes. For instance, if we are asked to add a subtyping relationship between two ECs $[a]$ and $[b]$, we should look in $G_{\preceq}$ for other ECs that could potentially end up in a subtyping relation due to $[a]$ becoming a subtype of $[b]$.

---

[1] See issues #11103, #11545, #11565 in Dotty repository: `https://github.com/lampepfl/dotty`.

At last but not least, an implementation of the proposed algorithm would help in deciding what parts could help the Scala 3 compiler in accepting a wider range of correct GADT programs – and potentially correct some of the remaining unsoundness holes.

Bibliography

[1] James Cheney and Ralf Hinze. *First-Class Phantom Types.* Tech. rep. Cornell University, 2003.

[2] Bernard A. Galler and Michael J. Fisher. “An Improved Equivalence Algorithm”. In: *Commun. ACM* 7.5 (May 1964), pp. 301–303. ISSN: 0001-0782. DOI: 10.1145/364099.364331. URL: https://doi.org/10.1145/364099.364331.

[3] Paolo G. Giarrusso. “Open GADTs and Declaration-Site Variance: A Problem Statement”. In: *Proceedings of the 4th Workshop on Scala.* SCALA ’13. Montpellier, France: Association for Computing Machinery, 2013. ISBN: 9781450320641. DOI: 10.1145/2489837.2489842. URL: https://doi.org/10.1145/2489837.2489842.

[4] Andrew Kennedy and Claudio Russo. “Generalized Algebraic Data Types and Object-Oriented Programming”. In: *OOPSLA ’05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA.* ACM New York, NY, USA, Oct. 2005, pp. 21–40. ISBN: 1-59593-031-0. URL: https://www.microsoft.com/en-us/research/publication/generalized-algebraic-data-types-and-object-oriented-programming/.

[5] S. C. Kleene. “On notation for ordinal numbers”. In: *Journal of Symbolic Logic* 3.4 (1938), pp. 150–155. DOI: 10.2307/2267778.

[6] Martin Odersky, Martin Sulzmann, and Martin Wehr. “Type Inference with Constrained Types”. In: *Theory and Practice of Object Systems* 5.1 (Jan. 1999), pp. 35–55. ISSN: 1074-3227. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1%3C35::AID-TAPO4%3E3.0.CO;2-4. URL: https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1%3C35::AID-TAPO4%3E3.0.CO;2-4.

[7] Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. “Towards Improved GADT Reasoning in Scala”. In: *Scala’19: Proceedings Of The 10Th Acm Sigplan International Symposium On Scala* (2019), pp. 12–16. DOI: 10.1145/3337932.3338813. URL: http://infoscience.epfl.ch/record/279651.

[8] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. “Simple Unification-Based Type Inference for GADTs”. In: *SIGPLAN Not.* 41.9 (Sept. 2006), pp. 50–61. ISSN: 0362-1340. DOI: 10.1145/1160074.1159811. URL: https://doi.org/10.1145/1160074.1159811.

[9] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, Dec. 2004. ISBN: 9780262281591. DOI: 10.7551/mitpress/1104.001.0001. URL: https://doi.org/10.7551/mitpress/1104.001.0001.

[10] Benjamin C. Pierce. *Types and Programming Languages.* 1st. The MIT Press, 2002. ISBN: 0262162091.

[11] François Pottier and Didier Rémy. “The Essence of ML Type Inference”. In: *Advanced Topics in Types and Programming Languages.* Ed. by Benjamin C. Pierce. The MIT Press, Dec. 2004. ISBN: 9780262281591. DOI: 10.7551/mitpress/1104.003.0016. eprint: https://direct.mit.edu/book/chapter-pdf/186369/9780262281591_caj.pdf. URL: https://doi.org/10.7551/mitpress/1104.003.0016.

[12] Marianna Rapoport and Ondřej Lhoták. “A Path to DOT: Formalizing Fully Path-Dependent Types”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360571. URL: https://doi.org/10.1145/3360571.

[13] Tiark Rompf and Nada Amin. “Type Soundness for Dependent Object Types (DOT)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 624–641. ISBN: 9781450344449. DOI: `10.1145/2983990.2984008`. URL: `https://doi.org/10.1145/2983990.2984008`.

[14] Tim Sheard and Emir Pasalic. “Meta-programming With Built-in Type Equality”. In: *Electronic Notes in Theoretical Computer Science* 199 (2008). Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004), pp. 49–65. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2007.11.012`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066108000789`.

[15] Vincent Simonet and François Pottier. “A Constraint-Based Approach to Guarded Algebraic Data Types”. In: *ACM Trans. Program. Lang. Syst.* 29.1 (Jan. 2007), 1–es. ISSN: 0164-0925. DOI: `10.1145/1180475.1180476`. URL: `https://doi.org/10.1145/1180475.1180476`.

[16] Radosław Waśko. “Formal foundations for GADTs in Scala”. In: (2020). URL: `http://infoscience.epfl.ch/record/277075`.

[17] Hongwei Xi, Chiyan Chen, and Gang Chen. “Guarded Recursive Datatype Constructors”. In: *SIGPLAN Not.* 38.1 (Jan. 2003), pp. 224–235. ISSN: 0362-1340. DOI: `10.1145/640128.604150`. URL: `https://doi.org/10.1145/640128.604150`.

Appendix A

Core functions proofs

A.1 Outline

This appendix section is dedicated to the proof of the core parts of the algorithm that we presented in 5.3.

Before diving in, we state some simple lemmas with respect to $\mathscr{K}$.

A.2 Useful lemmas

Lemma A.2.1. Let $\mathscr{K}$ be a valid knowledge structure. Then, for all $\bar{h} \in \mathrm{Im}(\mathcal{M})$, $h_1, h_2 \in \bar{h}$, the entailment $\mathscr{K} \Vdash \varsigma(\Theta(h_1)) \asymp \varsigma(\Theta(h_2))$ holds.

Proof. Straightforward use of the definition $\mathscr{K}$`-to-`$\mathcal{C}$ and lemma 3.3.8. ☐

Lemma A.2.2. Let $\mathscr{K}$ be a valid knowledge structure and $[a]$ contained in $\mathscr{K}$. Then, for all $\bar{h} \in \mathcal{M}([a])$, $h \in \bar{h}$, the entailment $\mathscr{K} \Vdash \varsigma([a]) \asymp \varsigma(\Theta(h))$ holds.

Proof. Straightforward use of the definition $\mathscr{K}$`-to-`$\mathcal{C}$. ☐

Lemma A.2.3. Let $\mathscr{K}$ be a valid knowledge structure. Then, for all $([a], [b]) \in E_{\preceq}$, the entailment $\mathscr{K} \Vdash \varsigma([a]) \preceq \varsigma([b])$ holds.

Proof. Straightforward use of the definition $\mathscr{K}$`-to-`$\mathcal{C}$. ☐

Corollary. Let $\mathscr{K}$ be a valid knowledge structure. Then, if $[a]$ and $[b]$ are in $\mathscr{K}$ and that there is a chain between $[a]$ and $[b]$ with respect to $G_{\preceq}$, the entailment $\mathscr{K} \Vdash \varsigma([a]) \preceq \varsigma([b])$ holds.

A.3 Simplification loop (partial correctness)

We are interested in proving partial correctness of the $\mathcal{C}$`-Simplify` function. While we do not provide a formal proof of $\mathcal{C}$`-Simplify` termination, we sketch one in A.10.

Proof. The proof revolves around the loops at (1) and (1d).

Outer loop (1).

It is straightforward to check that the loop invariants for the outer loop hold before the first iteration.

Now that the base case is established, we proceed by analyzing each statement.

For the return at (1a), we have by the loop invariant hypothesis (LIH) $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash$ `false` since the considered $C =$ `false` is in `cstrts`. (1b) is straightforward.

The call to `Deduction` at (1c) is well-formed. By the LIH, $\mathscr{K}^{(1)}$ is valid. Its postconditions state we have $\mathscr{K}^{(1)} \curlywedge C \Vdash \bigcurlywedge_j^m D_j$ with each D_j being `true`, `false`, $S_j \preceq T_j$ with S_j and T_j free of any refinement or $p_j : U_j$.

From the LIH and these postconditions, we can furthermore deduce $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \bigcurlywedge_j^m D_j$. Indeed, by the LIH, we have $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge \mathsf{cstrts}$ and because C is in $\bigcurlywedge \mathsf{cstrts}$, we have $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(1)} \curlywedge C$. Combining this with the postconditions of `Deduction` we get $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(1)} \curlywedge C \Vdash \bigcurlywedge_j^m D_j$ (result (⧫)).

Inner loop (1d).

The invariants hold before the first iteration thanks to the LIH of the outer loop.

Case (1d.i) is straightforward: we just need to apply the inner LIH. Case (1d.ii) is a straightforward application of the above observation. Case (1d.iv) is a straightforward application of the inner LIH and the definition of $\mathscr{K}$`-to-`$\mathcal{C}$.

For (1d.iii), we should first ensure that the requirements of the call to `Compact` are met: by the inner LIH, $\mathscr{K}^{(n)}$ is valid and `Deduction` ensures that S_j and T_j do not contain any refinement. `Compact` states that $\mathscr{K}'$ is valid and that $\mathscr{K}^{(n)} \curlywedge S_j \preceq T_j \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}'$.

Then, we are interested in proving that the inner LIH holds at the end of the iteration. The first one obviously holds. For the second one, we are interested in showing:

$$\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}^{(n)} \curlywedge \bigcurlywedge \mathsf{cstrts}'$$

From the inner LIH, we have $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(n)}$. With (⧫) and lemma 3.3.6, we get $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(n)} \curlywedge C \Vdash \bigcurlywedge_j^m D_j$. As such, thanks to `Compact`, we have $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}'$.

Using the inner LIH again, we have $\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathsf{cstrts}^{(n)}$. We then apply lemma 3.3.6 once again to glue all pieces together.

End of the outer loop (1).

The first outer LIH holds by the inner LI. For the second one, the goal is to show:

$$\mathscr{K} \curlywedge \bigcurlywedge_i^n C_i \Vdash \mathscr{K}^{(n)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(n)} \setminus \{C\}$$

which holds thanks to the inner LI.

Returned result (2).

Provided that the outer loop terminates, the postconditions of $\mathcal{C}$`-Simplify` are respected by the LI of the outer loop.

□

A.4 Deduction phase

We are interested in showing that `Deduction`, `DeductionIneq` and `DeductionTypedPath` terminate and hold their claims.

A.4.1 `DeductionIneq`

We start by giving a measure for `DeductionIneq` to help us prove termination. Next, we show that `DeductionIneq` yields constraints entailed by its argument. To do so, the induction hypothesis will employ the same measure as the termination.

The measure is in fact quite trivial. We just need to be careful about the case (5c) where the size of the arguments increases. A simple solution consists in having a single bit denoting whether the heads of the argument are the same or not. More formally, we define $m : \mathcal{T} \times \mathcal{T} \to \mathbb{N} \times \{0, 1\} \times \mathbb{N}$ with $m(T_1, T_2) = (\text{size}(T_2), \texttt{diffHead}, \text{size}(T_1))$ where `diffHead` is 1 if T_1 is a (possibly trivial) conjunction $\&_i^n Cls[S_{1,i}]$ and T_2 is of the form $Cls[S_2]$ for some types $S_{1,i}$, S_2 and some class symbol Cls. Otherwise, `diffHead` is 1. Then, the relation $(\mathcal{T} \times \mathcal{T}, \{(t, s) : m(t) < m(s),\ t, s \in \mathcal{T} \times \mathcal{T}\})$ is well-founded where $<$ is the lexicographic order on $\mathbb{N} \times \{0, 1\} \times \mathbb{N}$. As such, we can employ usual tools to prove termination and correctness.

Next, we need to show that each recursive call strictly decreases the measure.

Proof. By case analysis.

Cases (1)-(4), (6)-(15).

Straightforward. For case (13), we need to unfold the definition of `DeductionIneqDNF` to unveil the calls to `DeductionIneq` and note that the measure decreases as well.

Case (5).

For case (5b), it suffices to unfold the definition of `DeductionIneqVec` to note that the measure decreases as well. Case (5c) is the *raison-d'être* of this measure. We note that we have $m(T_1, T_2) = (\text{size}(T_2), 1, \text{size}(T_1))$ because $Cls_1 \neq Cls_2$. The measure for the recursive call is $m(U_1, U_2) = (\text{size}(U_2), 0, \text{size}(U_1)) = (\text{size}(T_2), 0, \text{size}(U_1)) < m(T_1, T_2)$ where $U_1 = \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S_1})]$ and $U_2 = Cls_2[\vec{S_2}] = T_2$.

□

We now would like to show that the constraints returned by `DeductionIneq` are entailed by $\mathscr{K}$ and $T_1 \preceq T_2$, or in other words, that $\mathscr{K} \curlywedge T_1 \preceq T_2 \Vdash$ `DeductionIneq`$(\mathscr{K}, T_1, T_2)$.

Furthermore, we would like to prove that the types contained within each constraints must be free of any refinement.

Proof. By induction on $m(T_1, T_2)$. We assume that the property holds for all S_1, S_2 for which we have $m(S_1, S_2) < m(T_1, T_2)$. We proceed by a case analysis.

Case (1).

Subcase (1b) is trivial, since all constraints entails `true`. For subcase (1a), we have $\Gamma \nvdash T_1 <: T_2$. T_1 and T_2 are closed: they are therefore idempotent element w.r.t. $(\phi, \gamma)(\cdot)$. We thus have $\Gamma \nvdash (\phi, \gamma)T_1 <: (\phi, \gamma)T_2$ for all ϕ, γ. Because T_1 and T_2 are closed, having Γ mapped through ϕ, γ does not alter the judgement, so we also have $(\phi, \gamma)\Gamma \nvdash (\phi, \gamma)T_1 <: (\phi, \gamma)T_2$ for all ϕ, γ. The contrapositive of the inversion lemma tells us that $\phi, \gamma \not\models T_1 \preceq T_2$ (for all ϕ, γ), so we indeed have $T_1 \preceq T_2 \equiv$ `false`.

Cases (2)-(4).

Trivial.

Case (5).

For (5a), we just need to apply lemma 3.4.5. (5b) is a straightforward application of the induction hypothesis and the corollary of lemma 3.4.3, after having unfolded the definition of `DeductionIneqVec`.

For (5c), applying lemma 3.4.4 gives us:

$$Cls_1[\vec{S_1}] \preceq Cls_2[\vec{S_2}] \Vdash \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S_1})] \preceq Cls_2[\vec{S_2}]$$

.

Lemma 3.3.6 allows us to add $\mathscr{K}$ to both side of the entailment, giving:

$$\mathscr{K} \curlywedge Cls_1[\vec{S_1}] \preceq Cls_2[\vec{S_2}] \Vdash \mathscr{K} \curlywedge \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S_1})] \preceq Cls_2[\vec{S_2}]$$

The IH yields:

$$\mathscr{K} \curlywedge \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S_1})] \preceq Cls_2[\vec{S_2}] \Vdash \texttt{DeductionIneq}(\mathscr{K}, \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S_1})], Cls_2[\vec{S_2}])$$

which also corresponds to the result of `DeductionIneq`$(\mathscr{K}, T_1, T_2)$ since it is returned.

Combining these two observations with the transitivity of entailment, we get the desired result.

Finally, the requirement of having constraints free of refinements is guaranteed by the IH.

Case (6).

Let us verify that the calls to $\mathcal{B}_{EC}$-`Subsumes` and $\mathcal{B}_{EC}$-`BoundsEntailed` are well-formed. Both functions require the given bounds to be $\mathcal{B}_{EC}$ bounds. Since B_1 and B_2 are $\mathcal{B}$ bounds, this condition is equivalent to requiring B_1 and B_2 to be free of any refinements, which is ensured by the guard on the case pattern.

For $\mathcal{B}_{EC}$-`Subsumes`, the domain of B_1 and B_2 are equal by the assumptions of well-formed subtyping constraints. As stated by the comment for this case, we assume that the domain of B_1 and B_2 – $\bar{X}$ – is disjoint from the free type variables of $\mathcal{K}$. It is always possible to satisfy this condition by a suitable α-renaming. These observations allow to conclude that the calls to $\mathcal{B}_{EC}$`-Subsumes` and $\mathcal{B}_{EC}$`-BoundsEntailed` are well-formed.

Let us now treat (6a) and (6b): (6b) is split in two subcases on whether T_1 and T_2 are free of refinements in order to satisfy the free-of-refinements requirement. For (6b.i), it simply returns the assumed $T_1 \preceq T_2$ without changing anything, so we naturally have $\mathcal{K} \curlywedge T_1 \preceq T_2 \Vdash T_1 \preceq T_2$. (6b.ii) is trivial.

For (6a), for all assignments γ, ϕ satisfying $\mathcal{K}$, there is a $\vec{U} \in \mathcal{T}^{|\vec{X}|}$ such that:

$$\phi[\vec{X} \mapsto \vec{U}], \gamma \models B_1 \wedge \phi[\vec{X} \mapsto \vec{U}], \gamma \not\models B_2$$

By applying this result to lemma 3.4.6, we get that $\phi, \gamma \not\models T_1 \preceq T_2$. By lemma 3.3.7, we have $\mathcal{K} \curlywedge T_1 \preceq T_2 \Vdash$ `false`.

Let us now turn our attention on the loop. Since the loop is simple, we skip the details on establishing a loop invariant. We proceed for each subcase. We omit the `otherwise` subcase since it is trivial.

Subcase (6c).

Let ϕ, γ be any assignments satisfying $\mathcal{K} \curlywedge T_1 \preceq T_2$. We need to show that $\phi, \gamma \models U_1 \preceq U_2$. The requirement of having U_1 and U_2 free of refinements is guaranteed by the IH.

We observe the following points:

- With `entls` being `true`, the postcondition of $\mathcal{B}_{EC}$`-BoundsEntailed` tells us that we have:

$$\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_1$$

- With the ($\Longrightarrow$) direction of lemma 3.4.6, we have:

$$\begin{gathered}\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_1 \implies \\ \phi[\vec{X} \mapsto \vec{A}], \gamma \models B_2 \curlywedge S_1 \preceq S_2.\end{gathered}$$

Let $\vec{A}$ be any element of $(\mathcal{T}^{\text{cl}})^{|\vec{X}|}$ with the same kind as $\vec{X}$. By combining these two points together, we get $\phi[\vec{X} \mapsto \vec{A}], \gamma \models S_1 \preceq S_2$.

Since $\bar{X} \mathbin{\#} \text{ftv}(\mathcal{K})$, we have by lemma 3.3.4 $\phi[\vec{X} \mapsto \vec{A}], \gamma \models \mathcal{K}$.

Then, by the IH, we have $\phi[\vec{X} \mapsto \vec{A}], \gamma \models \bigcurlywedge_i^m D_i$. Since $U_1 \preceq U_2 \in \bigcurlywedge_i^m D_i$, we also have $\phi[\vec{X} \mapsto \vec{A}], \gamma \models U_1 \preceq U_2$. We remark that the pattern guard guarantees that no type variable in $\bar{X}$ appears free in U_1 and U_2. As such, we can apply lemma 3.3.4 yielding the desired result $\phi, \gamma \models U_1 \preceq U_2$.

Subcase (6d).

Let ϕ, γ be any assignments satisfying $\mathcal{K} \curlywedge T_1 \preceq T_2$. We would like to show that the assignments ϕ, γ satisfy $[\vec{v}\vec{X} \lhd B_1]$ `=>>` $U_1 \preceq [\vec{v}\vec{X} \lhd B_2]$ `=>>` U_2 and that the entailed constraint is free of refinement. By the IH, U_1 and U_2 do not contain any refinement and the pattern guard guarantees that B_1, B_2 are free of refinement.

Let $\vec{A}$ be any element in $(\mathcal{T}^{\text{cl}})^{|\vec{X}|}$ with the same kind as $\vec{X}$ such that $\phi[\vec{X} \mapsto \vec{A}], \gamma \models B_1$. Due to `Subsumes` returning `true`, we have $\phi[\vec{X} \mapsto \vec{A}], \gamma \models B_2$. If we manage to show that $\phi[\vec{X} \mapsto \vec{A}], \gamma \models U_1 \preceq U_2$, we can apply the ($\Longleftarrow$) direction of lemma 3.4.6 to derive the desired result.

With the ($\Longrightarrow$) direction of lemma 3.4.6 and the assumptions about $\vec{A}$, we get $\phi[\vec{X} \mapsto \vec{A}], \gamma \models S_1 \preceq S_2$. As for the previous subcase, we can apply lemma 3.3.4 to get $\phi[\vec{X} \mapsto \vec{A}], \gamma \models \mathcal{K}$ since $\bar{X}$ is disjoint from $\text{ftv}(\mathcal{K})$. Applying the IH yields $\phi[\vec{X} \mapsto \vec{A}], \gamma \models \bigcurlywedge_i^m D_i$. Because $U_1 \preceq U_2$ is in $\bigcurlywedge_i^m D_i$, we obtain $\phi[\vec{X} \mapsto \vec{A}], \gamma \models U_1 \preceq U_2$.

Subcase (6e).

Let ϕ, γ be any assignments satisfying $\mathcal{K}$ Our goal is to prove that ϕ, γ cannot satisfy $T_1 \preceq T_2$. Once the goal proved, it is sufficient to apply lemma 3.3.7.

Let $\vec{A}$ be any element of $(\mathcal{T}^{\text{cl}})^{|\vec{X}|}$ with the same kind as $\vec{X}$. By applying a similar reasoning to (6c), we get $\phi[\vec{X} \mapsto \vec{A}], \gamma \models \mathcal{K} \curlywedge B_1 \curlywedge B_2 \curlywedge S_1 \preceq S_2$. By the IH, we get that $\phi[\vec{X} \mapsto \vec{A}], \gamma$ do not actually satisfy

$\mathscr{K} \curlywedge B_1 \curlywedge B_2 \curlywedge S_1 \preceq S_2$. Applying the contrapositive of the ($\Longrightarrow$) direction of lemma 3.4.6 gives the desired result.

Case (7).
Straightforward application of lemma 3.4.11

Case (8).
We first remark that the returned `false` at (8a) holds the postconditions by lemma 3.4.10. We then note that if we do not satisfy the condition at (8b), the returned value trivially satisfies the postconditions. From now on, we assume the existence of a p such that (p, R_1) is in $\mathcal{I}$. It is important to point out that p may not be a closed path.

We split the case analysis in three parts according to the deduction with respects to fields (8c), type members (8d) and methods (8e).

Fields (8c).
Let ϕ, γ be any assignments satisfying $\mathscr{K} \curlywedge R_1 \preceq R_2$. We would like to prove that ϕ, γ satisfy D. We first remark that, by $\mathscr{K}$`-to-`$\mathcal{C}$, ϕ, γ also satisfy $p : R_1$.

Because $\phi, \gamma \models p : R_1$, we also have $\phi, \gamma \models \gamma(p) : R_1$.

Furthermore, the IH tells us that, for any ϕ', γ' satisfying $\mathscr{K} \curlywedge [z \mapsto p]F_{1,i} \preceq [z \mapsto p]F_{2,i}$, the constraint D is satisfied by ϕ', γ'.

Since $\phi, \gamma \models R_1 \preceq R_2 \curlywedge \gamma(p) : R_1$, lemma 3.4.9 gives us $\phi, \gamma[z \mapsto \gamma(p)] \models F_{1,i} \preceq F_{2,i}$. By applying lemma 3.3.2, we get $\phi, \gamma \models [z \mapsto \gamma(p)]F_{1,i} \preceq [z \mapsto \gamma(p)]F_{2,i}$, which implies $\phi, \gamma \models [z \mapsto p]F_{1,i} \preceq [z \mapsto p]F_{2,i}$ (due to $\gamma(p)$ being closed). It suffices to apply the IH to conclude.

Bounds (8d).
Analogous to the previous case, with the extra step of applying lemma 3.4.2.

Methods (8e).

Argument types (8e.ii).
Let ϕ, γ be any assignments satisfying $\mathscr{K} \curlywedge R_1 \preceq R_2$. We are interested in showing $\phi, \gamma \models D_l$ assuming we have succeeded the check at (8e.i) and that $\bar{x} \mathrel{\#} \operatorname{ftmv}(D_l)$. We also note that z does not appear in D_l since it is substituted to p.

Similarly to (8c), the successful check at (8b) guarantees us the existence of p such that $\phi, \gamma \models p : R_1$ holds. Remarking that $\gamma(p)$ is closed and that we have $\phi, \gamma \models \gamma(p) : R_1$ as in the previous case, the application of lemma 3.4.9 gives us:

$$\begin{aligned}\phi, \gamma[z \mapsto \gamma(p)] \models \texttt{def}\ \ & m_i[\vec{Y_i} \lhd B_{Y,1,i}](\vec{x}_i : \vec{U}_{1,i}) : V_{1,i} \preceq \\ \texttt{def}\ \ & m_i[\vec{Y_i} \lhd B_{Y,2,i}](\vec{x}_i : \vec{U}_{2,i}) : V_{2,i}\end{aligned}$$

We get from lemma 3.4.8, for all $\vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{Y}|}$ and $\vec{q} \in (\mathcal{P}^{\mathrm{cl}})^{|\vec{x}|}$:

$$\begin{gathered}\phi[\vec{Y_i} \mapsto \vec{A}], \gamma[z \mapsto \gamma(p), \vec{x} \mapsto \vec{q}] \models B_{1,i} \curlywedge \vec{x} : \vec{U}_{1,i} \\ \Longrightarrow \\ \phi[\vec{Y_i} \mapsto \vec{A}], \gamma[z \mapsto \gamma(p), \vec{x} \mapsto \vec{q}] \models B_{2,i} \curlywedge U_{2,i,j} \preceq U_{1,i,j}\end{gathered}$$

By the oracle check at (8e.i), we know that $\phi', \gamma' \models \vec{x} : \vec{U}_{1,i}$ for all assignments ϕ', γ'. We therefore have $\gamma[z \mapsto \gamma(p), \vec{x} \mapsto \vec{q}] \models \vec{x} : \vec{U}_{1,i}$ as well.

Applying the ($\Longleftarrow$) direction of lemma 3.4.6 yields:

$$\phi, \gamma[z \mapsto \gamma(p), \vec{x} \mapsto \vec{q}] \models [\vec{v}\vec{Y_i} \lhd B_{Y,1,i}] \,\texttt{=>>}\, U_{2,i,j} \preceq [\vec{v}\vec{Y_i} \lhd B_{Y,2,i}] \,\texttt{=>>}\, U_{1,i,j}$$

Due to $\gamma(p)$ be closed, the above implies:

$$\phi, \gamma[\vec{x} \mapsto \vec{q}] \models [z \mapsto p]\big([\vec{v}\vec{Y_i} \lhd B_{Y,1,i}] \,\texttt{=>>}\, U_{2,i,j} \preceq [\vec{v}\vec{Y_i} \lhd B_{Y,2,i}] \,\texttt{=>>}\, U_{1,i,j}\big)$$

Then, we use the IH to get $\phi, \gamma[\vec{x} \mapsto \vec{q}] \models D_l$. Since $(\{z\} \cup \bar{x}) \mathrel{\#} \mathrm{ftmv}(D_l)$, we can apply lemma 3.3.5 to get the desired result $\phi, \gamma \models D_l$.

Return type (8e.iii).

Analogous to the previous case.

Argument types (8e.iv).

Similar to (8e.ii), except that we do not apply the lemma 3.4.6. We note that, since $\vec{Y_i}$ is empty, an extension of ϕ such as $\phi[\vec{Y_i} \mapsto \vec{A}]$ results in ϕ.

Return type (8e.v).

Analogous to the previous case.

Cases (9)-(13).

Straightforward application of the IH and lemma 3.4.12, alongside the property of `ApproxDisjunction`.

Cases (14)-(15).

Trivial.

□

A.4.2 `DeductionTypedPath`

Since termination is trivial, we focus on showing that the claimed postcondition is correct.

Proof. Case (1) is trivial. An appropriate invariant for the loop at (2) is $\mathscr{K} \curlywedge p : T \Vdash D$, which naturally holds before the first iteration. Assuming $\mathscr{K} \curlywedge p : T \Vdash D$, we proceed by a case analysis on the matched pattern and show that the loop invariant holds.

Case (2a).

$\mathcal{T}$`-CommonTypes` tells us that we have $p : Cls[\vec{A}] \;\&\; Cls[\vec{B}]$. By lemma 3.4.7 we have $\mathscr{K} \curlywedge p : Cls[\vec{A}] \& Cls[\vec{B}] \Vdash A_j \asymp B_j$ for all j such that $v_j = \pm$. We also have $\mathscr{K} \curlywedge A_j \preceq B_j \Vdash D_1$ and $\mathscr{K} \curlywedge B_j \preceq A_j \Vdash D_2$ where $D_1 = \texttt{DeductionIneq}(\mathscr{K}, A_j, B_j)$ and $D_2 = \texttt{DeductionIneq}(\mathscr{K}, B_j, A_j)$. By combining everything together, we indeed get $\mathscr{K} \curlywedge p : T \Vdash D \curlywedge D_1 \curlywedge D_2$.

Case (2b).

We have $p : Cls_1[\vec{A}] \& Cls_2[\vec{B}]$ with $Cls_1 \neq Cls_2$. Lemma 3.4.4 tells us that $Cls_1[\vec{A}] \& Cls_2[\vec{B}] \Vdash Cls_1[\vec{A}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{A})]$. Applying lemmas 3.4.12 and 3.4.13 yields $p : Cls_2[\boldsymbol{\sigma}_i(\vec{A})] \& Cls_2[\vec{B}]$ for $1 \leq i \leq N$. Applying lemma 3.4.7 for $1 \leq i \leq N$ and for each j such that $v_j = \pm$ yields $p : Cls_2[\boldsymbol{\sigma}_i(\vec{A})] \;\&\; Cls_2[\vec{B}] \Vdash \boldsymbol{\sigma}_i(\vec{A})_j \asymp B_j$. Noticing that we call `DeductionIneq` with $\boldsymbol{\sigma}_i(\vec{A})_j$ and B_j, we can apply the same reasoning as in the previous case to conclude.

□

A.4.3 `Deduction`

Proof. By case analysis. Case (2) follows from `DeductionIneq`. For (1), $\mathcal{T}$`-InhabitedTypes` guarantees us that $x : T \Vdash p : S$ for all iterated p, S and that the returned set is finite. With lemma 3.3.6, we can add $\mathscr{K}$ to the antecedents, yielding $\mathscr{K} \curlywedge x : T \Vdash p : S$. We then get $\mathscr{K} \curlywedge x : T \Vdash \texttt{DeductionTypedPath}(\mathscr{K}, p, S)$ for all iterated p, S. The accumulation of entailments with disjunctions maintains the invariant $\mathscr{K} \curlywedge x : T \Vdash D$, as stated by lemma 3.3.6.

□

A.5 Compaction entry point

We are interested in proving the claims affirmed by the `Compact` function.

Before going on, we remind the used notation:

$$[a] \in \mathscr{K} \triangleq [a] \in \mathcal{Q}\texttt{-AllMembers}(\mathscr{K})$$

$$h \in \mathscr{K} \triangleq h \in \mathrm{dom}(\Theta)$$

$$\varsigma \triangleq EC_H\texttt{-Subst}(\mathscr{K})$$

$$M(\mathscr{K}, \mathsf{toMerge}) \triangleq \big\{\varsigma([x]) \asymp \varsigma([y]) : \{[x],[y]\} \in \mathsf{toMerge}\big\}$$

$$I(\mathscr{K}, \mathsf{ineqs}) \triangleq \big\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in \mathsf{ineqs}\big\}$$

$$L(\mathscr{K}, \mathsf{toMerge}) \triangleq \sum_{\{[a],[b]\} \in \mathsf{toMerge}} 1 \text{ if } ([a],[b]),([b],[a]) \notin E_{\mathcal{S}} \ 0 \text{ otherwise}$$

Proof. We proceed line by line.

Statements (1), (2).

The calls are naturally well-formed. We deduce the following points:

1. By Q-FEC1, $\mathscr{K}^{(1)}$ is valid. Furthermore $\mathscr{K}$ and $\mathscr{K}^{(1)}$ agree on $\mathcal{M}$, Θ, $\mathcal{R}$, $\mathcal{D}$, $\mathcal{Q}$ and T_R for entries defined at $\mathscr{K}$.
2. By Q-FEC5, $[s]$ is not NIL. Because we pass an empty set for the bound variables, Q-FEC3 guarantees that $[s]$ an EC_H.
3. By Q-FEC2, S and $[s]$ have the same kind under $\mathscr{K}^{(1)}$.
4. By Q-FEC6: $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(1)}$.
5. We furthermore have $\mathscr{K} \Vdash S \asymp \varsigma^{(1)}([s])$ by Q-FEC7 and because $\varsigma^{(1)}(S) = S$ (S does not contain any EC_H as it is a $\mathcal{T}$ type).

We deduce similar points for (2). We note that, since S and T have the same kind (thanks to the well-formedness of constraints), so do $[s]$ and $[t]$ under $\mathscr{K}^{(2)}$ (Q-FEC1 guarantees that the kindness for EC is preserved across "updates" of $\mathscr{K}$'s).

Before going on, let us show that $\mathscr{K} \curlywedge S \preceq T \Vdash \varsigma^{(2)}([s]) \preceq \varsigma^{(2)}([t])$ ($\blacklozenge$). We have at our disposal $\mathscr{K} \Vdash S \asymp \varsigma^{(1)}([s])$ and $\mathscr{K}^{(1)} \Vdash T \asymp \varsigma^{(2)}([t])$. Due to $[s]$ belonging to $\mathscr{K}^{(1)}$, we apply Q-FEC1 to deduce that $\varsigma^{(1)}([s]) = \varsigma^{(2)}([s])$, leading us to $\mathscr{K} \curlywedge S \preceq T \Vdash S \asymp \varsigma^{(2)}([s]) \curlywedge T \asymp \varsigma^{(2)}([t])$. We then apply lemma 3.3.8 to conclude.

Statement (3).

It is straightforward to check that the call to `TryAddInequality` is well-formed.

We deduce:

1. $\mathscr{K}^{(3)}$ is valid. All ECs appearing in $\mathsf{toMerge}$ are contained in $\mathscr{K}^{(3)}$; furthermore the members of the unordered pairs in $\mathsf{toMerge}$ have the same kind.
2. We have $\mathscr{K}^{(2)} \curlywedge \varsigma^{(2)}([s]) \preceq \varsigma^{(2)}([t]) \Vdash \mathscr{K}^{(3)} \curlywedge \mathop{\Large\curlywedge} \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(3)}, \mathsf{toMerge})$ and from ($\blacklozenge$), we have $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(3)} \curlywedge \mathop{\Large\curlywedge} \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(3)}, \mathsf{toMerge})$

Loop (4).

The loop invariants holds before the first iteration thanks to the postconditions of `TryAddInequality`. However, the termination proof is not straightforward. We first prove that the loop terminates and proceed to show that the invariants hold at the end of each iteration.

Loop (4): termination.

To prove termination, we employ the measure $m(\mathscr{K}, \mathsf{toMerge}) = (|\mathrm{dom}(\mathcal{M})|, L(\mathscr{K}, \mathsf{toMerge}), |\mathsf{toMerge}|)$. We show that $m(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(n)}) < m(\mathscr{K}^{(4)}, \mathsf{toMerge})$. Case (4a) is straightforward.

For case (4b), we employ Q-MG3 of `Merge`. We perform a case analysis on:

$$([a],[b]),([b],[a]) \notin E_{\preceq}^{(4)} \wedge \texttt{ExistUndirChain}(G_{\preceq}^{(4)}, [a], [b])$$

If it is false, then we have $|\mathrm{dom}(\mathcal{M}^{(n)})| < |\mathrm{dom}(\mathcal{M}^{(4)})|$, so the measure decreases.

Otherwise, we have $\mathscr{K}^{(4)} = \mathscr{K}^{(n)}$, $L(\mathscr{K}^{(4)}, \mathsf{toMerge}') = L(\mathscr{K}^{(n)}, \mathsf{toMerge}') = 0$ and furthermore $([a],[b])$, $([b],[a]) \notin E_{\preceq}$. We remark that $L(\mathscr{K}^{(4)}, \{[a],[b]\}) = L(\mathscr{K}^{(n)}, \{[a],[b]\}) = 1$. As such, we get:

$$\begin{aligned}
L(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(n)}) &= L(\mathscr{K}^{(n)}, (\mathsf{toMerge} \cup \mathsf{toMerge}') \setminus \{[a],[b]\}) \\
&= L(\mathscr{K}^{(n)}, \mathsf{toMerge} \setminus \{[a],[b]\}) + L(\mathscr{K}^{(n)}, \mathsf{toMerge}' \setminus \{[a],[b]\}) \\
&= L(\mathscr{K}^{(n)}, \mathsf{toMerge}) - L(\mathscr{K}^{(n)}, \{[a],[b]\}) \\
&= L(\mathscr{K}^{(n)}, \mathsf{toMerge}) - 1 \\
&= L(\mathscr{K}^{(4)}, \mathsf{toMerge}) - 1 \\
&< L(\mathscr{K}^{(4)}, \mathsf{toMerge})
\end{aligned}$$

In the third equality, we have used the fact that $\{[a],[b]\} \in \mathsf{toMerge}$ to transform the set removal into a subtraction. The term $L(\mathscr{K}^{(n)}, \mathsf{toMerge}' \setminus \{[a],[b]\})$ is simplified because $L(\mathscr{K}^{(n)}, \mathsf{toMerge}') = 0$.

Since $\mathscr{K}^{(4)} = \mathscr{K}^{(n)}$, we have $|\mathrm{dom}(\mathcal{M}^{(4)})| = |\mathrm{dom}(\mathcal{M}^{(n)})|$, and the measure decreases due to the decrease of the second component of m.

Loop (4): correctness.

We show that the invariants hold at the end of each iteration. The branch (4a) trivially maintains the LI.

For branch (4b), we first have to ensure that the precondition of `Merge` are satisfied. These require to have $[a]$ and $[b]$ distinct, be the representatives of their equivalence class and to have the same kind, which we naturally have thanks to the previous calls to `𝒬-Find`, the check for equality and the LIH respectively. The validity of $\mathscr{K}^{(4)}$ is guaranteed thanks to the LIH.

Starting with the first loop invariant, `Merge` states that $\mathscr{K}^{(n)}$ is valid and that the set of EC_H of $\mathscr{K}^{(3)}$ and $\mathscr{K}^{(n)}$ are identical [1]. By Q-MG2 and this observation, $\mathsf{toMerge}' \subseteq \mathscr{K}^{(n)}$ and by the LIH, $\mathsf{toMerge}^{(n)} \subseteq \mathscr{K}^{(n)}$. By Q-MG2, the pairs in $\mathsf{toMerge}'$ have the same kind.
To prove that the pairs in $\mathsf{toMerge}$ have the same kind under $\mathscr{K}^{(n)}$, we apply Q-MG1 to deduce that each $[x] \in \bigcup \mathsf{toMerge}$ keeps the same kind under $\mathscr{K}^{(n)}$ and apply the LIH to deduce that the members of the pairs have the same kind.

It remains to show the second loop invariant. It is beneficial to show that $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(4)} \curlywedge \varsigma^{(4)}([a]) \asymp \varsigma^{(4)}([b])$. Thanks to the LIH, we have $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(4)} \curlywedge M(\mathscr{K}^{(4)}, \mathsf{toMerge})$. By using the definition of M and because $\{[a],[b]\}$ is in $\mathsf{toMerge}$, we have $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(4)} \curlywedge \varsigma^{(4)}([a]) \asymp \varsigma^{(4)}([b])$.

We break the entailment of the second loop invariant into sub-entailments and prove that these hold.

1. $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(n)}$:
   Straightforward.
2. $\mathscr{K} \curlywedge S \preceq T \Vdash \bigcurlywedge \mathsf{cstrts}^{(n)}$:
   Straightforward application of the LIH with Q-MG4 and noting that we have "access" to the entailed conjuncts thanks to having $\mathscr{K} \curlywedge S \preceq T \Vdash \mathscr{K}^{(4)} \curlywedge \varsigma^{(4)}([a]) \asymp \varsigma^{(4)}([b])$.
3. $\mathscr{K} \curlywedge S \preceq T \Vdash M(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(n)})$:
   We note that showing $\mathscr{K} \curlywedge S \preceq T \Vdash M(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(n)} \cup \{[a],[b]\})$ proves the desired result as well; it is less cumbersome to prove this claim instead. We note that $[a]$ and $[b]$ are still contained in $\mathscr{K}^{(n)}$, so any operation applied to them remains valid.
   We can split $M(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(n)} \cup \{[a],[b]\})$ into $M(\mathscr{K}^{(n)}, \mathsf{toMerge}') \curlywedge M(\mathscr{K}^{(n)}, \mathsf{toMerge}^{(4)})$. The entailment $\mathscr{K} \curlywedge S \preceq T \Vdash M(\mathscr{K}^{(n)}, \mathsf{toMerge}')$ is straightforward.
   Showing $\mathscr{K} \curlywedge S \preceq T \Vdash M(\mathscr{K}^{(n)}, \mathsf{toMerge})$ is a bit tricky because we do not have $\varsigma^{(4)} = \varsigma^{(n)}$. Instead, `Merge` gives the weaker property that:

$$\mathscr{K}^{(4)} \curlywedge \varsigma^{(4)}([a]) \asymp \varsigma^{(4)}([b]) \Vdash \bigcurlywedge \{\varsigma^{(4)}([x]) \asymp \varsigma^{(n)}([x]), [x] \in \mathscr{K}^{(4)}\}$$

---

[1] Note that `Merge` reduces by one the number of *equivalence classes*, not the number of *equivalence classes handles*: these are simply regrouped into the same equivalence class.

Unfolding M, using the LIH and applying lemma 3.3.8 yields:

$$\begin{gathered}
\mathscr{K} \curlywedge S \preceq T \\
\Vdash \mathscr{K}^{(4)} \curlywedge \varsigma^{(4)}([a]) \asymp \varsigma^{(4)}([b]) \\
\Vdash \bigcurlywedge\{\varsigma^{(4)}([x]) \asymp \varsigma^{(n)}([x]), [x] \in \mathscr{K}^{(4)}\} \curlywedge \bigcurlywedge\left\{\varsigma^{(4)}([x]) \asymp \varsigma^{(4)}([y]) : \{[x],[y]\} \in \mathsf{toMerge}\right\} \\
\Vdash \bigcurlywedge\left\{\varsigma^{(n)}([x]) \asymp \varsigma^{(n)}([y]) : \{[x],[y]\} \in \mathsf{toMerge}\right\} \triangleq M(\mathscr{K}^{(n)}, \mathsf{toMerge})
\end{gathered}$$

Returned result (5).
The postconditions of `Compact` are respected by the LI of (4).

□

A.6 ECs processing

The task of finding and creating ECs is divided into multiple functions. We prove them in the following order:

1. $\mathcal{T}$-`FindOrCreateEC`
2. $\mathcal{T}_{EC}$-`FindOrCreateEC`
3. $\mathcal{T}_{EC}$-`TryFindApplied`
4. $\mathcal{T}_{EC}$-`CreateEC`

A.6.1 $\mathcal{T}$-`FindOrCreateEC`

Proof. By induction on the size of T with a case analysis on the shape of T. $\mathscr{K}$, B_X and $\vec{v}_X$ are held abstract.

$\mathcal{T}$-`FindOrCreateECVec` and $\mathcal{B}$-`FindOrCreateEC` contain recursive calls to $\mathcal{T}$-`FindOrCreateEC`. They are meant to be unfolded. It is then straightforward to check that all recursive calls strictly decrease the size of the argument.

Case (1).
We remark that $\varsigma(X) = X$, taking care of the definedness and equivalence requirements present in Q-FEC2 and Q-FEC7 respectively. Q-FEC3 is vacuous. Q-FEC4 is straightforward.

Case (2).
We first need to unfold the definition of $\mathcal{T}$-`FindOrCreateECVec` at (2a), which unveils a recursive call within a loop.

If we return at (2b), the postconditions are held (in that case, we must have $\mathsf{create} = \mathtt{false}$ by the IH). Therefore, we focus on (2c).

Returning back to the unfolding of $\mathcal{T}$-`FindOrCreateECVec`, by the IH, all the $\mathscr{K}'_i$ within the loop satisfy Q-FEC1 and Q-FEC6. We thus have a chain of entailment $\mathscr{K} \Vdash \mathscr{K}'_1 \Vdash \mathscr{K}'_2 \Vdash ... \Vdash \mathscr{K}^{(1)}$ and $\mathscr{K}, \mathscr{K}'_1, \mathscr{K}'_2, ..., \mathscr{K}^{(1)}$ all agree on common domains. Combining this observation with the IH, $\varsigma^{(1)}$ is defined for all $S'_i \in \vec{S}'$. Furthermore, $\vec{S}$ and $\vec{S}'$ have the same kind under $\mathscr{K}^{(1)}$. We also deduce that $\mathscr{K}$ entails $\varsigma^{(1)}(\vec{S}) \asymp \varsigma^{(1)}(\vec{S}')$, therefore it entails $\varsigma^{(1)}(F[\vec{S}]) \asymp \varsigma^{(1)}(F[\vec{S}'])$ as well.

It remains to show Q-FEC3 and Q-FEC4. We remark that the former is vacuous. By the IH, all S'_i satisfy the property stated by Q-FEC3 and Q-FEC4 with $\mathsf{inHead} = \mathtt{false}$. All the possible forms for each S'_i satisfy $\mathcal{T}_{EC}$-$\mathtt{in}$-Θ-$\mathtt{Inv}(S'_i, \mathtt{false}, \mathrm{dom}(B_X))$. Therefore, $\mathcal{T}_{EC}$-$\mathtt{in}$-Θ-$\mathtt{Inv}(F[\vec{S}'], \mathtt{true}, \mathrm{dom}(B_X))$ holds.

Case (3). All but two preconditions for $\mathcal{T}_{EC}$-`FindOrCreateEC` are directly implied by those of $\mathcal{T}$-`FindOrCreateEC`. We note that X and $p.Q$ are idempotent under ς (because they do not contain any EC_H), as such, $\varsigma(T)$ is defined. It is straightforward to check that $\mathcal{T}_{EC}$-$\mathtt{in}$-Θ-$\mathtt{Inv}(T, \mathtt{true}, \mathrm{dom}(B_X))$ holds. The second precondition, which restrains the form of T, is guaranteed by having filtered these cases out with (1) and (2).

The postconditions of $\mathcal{T}\texttt{-FindOrCreateEC}$ are ensured by $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$.

Case (4).

Using a similar reasoning as (2), we get to conclude that all S'_i satisfy the property stated by Q-FEC3 and Q-FEC4, as well as the assertion $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S'_i, \texttt{false}, \mathrm{dom}(B_X))$.

We only consider (4d) since (4b) is trivial and (4c) straightforward.

In (4d), we call $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$ which has two extra sets of requirements that is not directly covered by the assumptions of $\mathcal{T}\texttt{-FindOrCreateECVec}$. We start with the first one.

Because $\varsigma^{(1)}(\vec{S'})$ is defined, $\varsigma^{(1)}(TyCon[\vec{S'}])$ is defined as well. For $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(TyCon[\vec{S'}], \texttt{true}, \mathrm{dom}(B_X))$ [2], we need to prove that $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S'_i, \texttt{false}, \mathrm{dom}(B_X))$ is true for all $S'_i \in \vec{S'}$, which we already have. Furthermore, if $TyCon$ is a bound abstract type constructor, the argument `inHead` of $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ must be set to `true`, which is the case by assumption.

The second set of requirements has already been taken care of at (2).

Now that we have ensured the well-formedness of the call to $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$, we can turn our attention on proving the claims Q-FEC1 through Q-FEC7.

$\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$ guarantees the same set of properties as $\mathcal{T}\texttt{-FindOrCreateEC}$ for $\mathcal{K}^{(2)}$ and T', so the conclusion is rather straightforward. $\mathcal{K}$, $\mathcal{K}^{(1)}$ and $\mathcal{K}^{(1)}$, $\mathcal{K}^{(2)}$ agree on common domains, so $\mathcal{K}$, $\mathcal{K}^{(2)}$ agree as well, satisfying Q-FEC1. Q-FEC2 stems from $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$ and the fact that T has the same kind as $TyCon[\vec{S'}]$. Q-FEC5 is straightforward. Q-FEC6 is straightforward as well because $\mathcal{K} \Vdash \mathcal{K}^{(1)} \Vdash \mathcal{K}^{(2)}$.

The properties Q-FEC3 and Q-FEC4 are guaranteed by $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$.

It remains to prove Q-FEC7. We have at our disposal:

$$\mathcal{K} \Vdash \varsigma^{(1)}(\vec{S}) \asymp \varsigma^{(1)}(\vec{S'}) \qquad \text{From (4a)}$$

$$\mathcal{K}^{(1)} \Vdash \varsigma^{(2)}(TyCon[\vec{S'}]) \asymp \varsigma^{(2)}(T') \qquad \text{From (4d)}$$

Since $T = TyCon[\vec{S}]$, we also have $\mathcal{K} \Vdash \varsigma^{(1)}(T) \asymp \varsigma^{(1)}(TyCon[\vec{S'}])$.

Because $\mathcal{K}$, $\mathcal{K}^{(1)}$ and $\mathcal{K}^{(2)}$ agree on common domains, we get that:

$$\mathcal{K} \Vdash \varsigma^{(2)}(T) \asymp \varsigma^{(2)}(TyCon[\vec{S'}])$$

$$\mathcal{K}^{(1)} \Vdash \varsigma^{(2)}(TyCon[\vec{S'}]) \asymp \varsigma^{(2)}(T')$$

With lemmas 3.3.6 and 3.3.8, we have:

$$\mathcal{K}^{(1)} \curlywedge \varsigma^{(2)}(T) \asymp \varsigma^{(2)}(TyCon[\vec{S'}]) \Vdash \varsigma^{(2)}(T) \asymp \varsigma^{(2)}(T')$$

Since $\mathcal{K} \Vdash \mathcal{K}^{(1)} \curlywedge \varsigma^{(2)}(T) \asymp \varsigma^{(2)}(TyCon[\vec{S'}])$, we have $\mathcal{K} \Vdash \varsigma^{(2)}(T) \asymp \varsigma^{(2)}(T')$ which is the desired result.

Case (5).

By the IH, $\mathcal{K}^{(1)}$ and each $S'_{i,j}$ satisfy the properties stated by Q-FEC1 through Q-FEC7. As such, $\varsigma^{(1)}(\big|_i^n \&_j^{m_i} S'_{i,j})$ is defined. Moreover, each $S'_{i,j}$ is either an applied type constructor of a class, of a bound abstract type constructor , or of the form $[s]$ or $[s][\vec{X}]$ with $\vec{X} = \mathrm{dom}(B_X)$. Then, the DNF $\big|_i^n \&_j^{m_i} S'_{i,j}$ satisfies the $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ predicate in a head position with bound variables $\mathrm{dom}(B_X)$ (⧫).

These observations allows us to conclude that the call to $\mathcal{T}_{EC}\texttt{-SimplifyDNF}$ is well-formed. If S' is an EC_H (branch (5c)), the properties Q-FEC3 and Q-FEC4 are trivially satisfied. On the other hand, if we reach (5d), the conclusion is straightforward.

It remains (5e). This case is similar to (4) except for proving that the preconditions of $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$ hold. In particular, we need to show that $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S', \texttt{true}, \mathrm{dom}(B_X))$. We first remark that $\mathcal{T}_{EC}\texttt{-SimplifyDNF}$ may only remove terms in the DNF. Then, if S' remains a DNF, the predicate is satisfied as point out by (⧫). Otherwise, if S' is not a DNF, it must satisfy $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S', \texttt{true}, \mathrm{dom}(B_X))$ by Q-FEC4 of the IH because S' cannot be an EC_H (such case has been filtered out by (5c)).

---

[2] $\mathcal{T}_{EC}\texttt{-FindOrCreateEC}$ requires $TyCon[\vec{S'}]$ to satisfy the predicate in a head position even if we are in a non-head context.

Now that we have ensured that that the call to $\mathcal{T}_{EC}$-`FindOrCreateEC` is well-formed, we apply the same reasoning as for (4) to glue all pieces together.

Case (6).

As stated by the associated comment, this case is a specialized version of (7). As such, we prove its correctness by showing (7).

Case (7).

We first unfold the definition of $\mathcal{B}$-`FindOrCreateEC` at (7a), revealing two recursive calls.

We should ensure that the recursive calls are well-formed. By assumptions, $\mathscr{K}$ is valid and by the IH, $\mathscr{K}'$ ($\mathscr{K}^{(1)}$ in the body of the considered case) keeps its validity throughout the updates. It is straightforward to check that B_{tmp} satisfies P-FEC2.

We now analyze B'_Y. If L'_i and U'_i are not NIL, then their shapes are described by properties Q-FEC3 and Q-FEC4 of the IH in a head position. For all possible shapes, L'_i and U'_i satisfy $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ in a head position under the type variables $\bar{X} \cup \bar{Y}$. With proper prior α-renaming of $\bar{Y}$ (as stated by the comment on the analyzed case (7)), we have $\bar{Y} \# \bar{X}$: as such, we have $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B'_Y, \bar{X})$. By P-FEC2, we have $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_X, \emptyset)$; it is therefore straightforward to check that we have $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_X B_Y, \emptyset)$.

We are also interested in showing that $\varsigma^{(1)}(B'_Y)$ is defined. We need to ensure that $\varsigma^{(1)}(L'_i)$ and $\varsigma^{(1)}(U'_i)$ are defined, for all $Y_i \in \vec{Y}$ with $(L'_i, U'_i) = B'_Y(Y_i)$. For each iterated Y_i, the associated $\varsigma^{(a)}$ is defined for L'_i. Similarly, the associated $\varsigma^{(n)}$ is defined for U'_i. By Q-FEC1 of the IH, ς, and all the $\varsigma^{(a)}$ and $\varsigma^{(n)}$ within the loop agree on common domains. As such, for all Y_i, $\varsigma^{(1)}(L'_i)$ and $\varsigma^{(1)}(U'_i)$ must be defined because the domain of $\varsigma^{(1)}$ extends the domain of the iterated $\varsigma^{(a)}$ and $\varsigma^{(n)}$.

These observations allow us to conclude that the recursive call to $\mathcal{T}$-`FindOrCreateEC` at (7c) is well-formed. Therefore, by the IH, $\mathscr{K}^{(2)}$ and S' satisfy Q-FEC1 through Q-FEC7. Because the return at (7d) is straightforward, we focus on the `else` branch at (7e).

$\mathcal{T}_{EC}$-`FindOrCreateEC` requirements are all straightforward except for:

$$\varsigma^{(2)}([\vec{v}_Y\vec{Y} \triangleleft B'_Y] \mathtt{=>>} S')\downarrow \equiv \varsigma^{(2)}(B'_Y)\downarrow \wedge \varsigma^{(2)}(S')\downarrow$$

$$\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}([\vec{v}_Y\vec{Y} \triangleleft B'_Y] \mathtt{=>>} S', \mathtt{true}, B_X, \vec{v}_X)$$

By the IH, $\varsigma^{(2)}(S')$ is defined. By the IH, $\varsigma^{(1)}$ and $\varsigma^{(2)}$ agree on common domain, so $\varsigma^{(2)}(B'_Y)$ is defined as well.

For the second requirement, we need to show:

$$\begin{gathered}\mathrm{dom}(B'_Y) = \bar{Y} \wedge \mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B'_Y, \bar{X}) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S', \mathtt{true}, \bar{X} \cup \bar{Y})\end{gathered}$$

The first conjunct is straightforward. We already have the second from an earlier observations. For the third conjunct, we employ the facts about the shape of S', which must obey Q-FEC4.

We have ensured that the call to $\mathcal{T}_{EC}$-`FindOrCreateEC` situated in the branch at (7e) is well-formed. $\mathcal{T}_{EC}$-`FindOrCreateEC` states that $\mathscr{K}^{(3)}$ and T' satisfy Q-FEC1-Q-FEC7.

Using a similar reasoning as for the analysis of (4), we can conclude Q-FEC1 through Q-FEC6.

Unsurprisingly, showing Q-FEC7 is different from (4). We are looking to prove the following:

$$\begin{gathered}\mathscr{K} \Vdash \varsigma^{(3)}([\vec{v}_Y\vec{Y} \triangleleft B_Y] \mathtt{=>>} S) \preceq \varsigma^{(3)}(T') \\ \equiv \\ \mathscr{K} \Vdash [\vec{v}_Y\vec{Y} \triangleleft B_Y] \mathtt{=>>} S \asymp \varsigma^{(3)}(T')\end{gathered}$$

where we have removed $\varsigma^{(3)}(\cdot)$ due to its idempotence on $\mathcal{T}$ types.

From $\mathcal{T}_{EC}$-`FindOrCreateEC` and using the fact that $\mathscr{K} \Vdash \mathscr{K}^{(2)}$, we have:

$$\begin{gathered}\mathscr{K} \Vdash \varsigma^{(3)}([\vec{v}_Y\vec{Y} \triangleleft B'_Y] \mathtt{=>>} S') \asymp \varsigma^{(3)}(T') \\ \equiv \\ \mathscr{K} \Vdash [\vec{v}_Y\vec{Y} \triangleleft \varsigma^{(3)}(B'_Y)] \mathtt{=>>} \varsigma^{(3)}(S') \asymp \varsigma^{(3)}(T')\end{gathered}$$

where we have "pushed" $\varsigma^{(3)}(\cdot)$ within the HK abstraction in the left member. As stated by the comment of the branch, it is possible to α-rename $\vec{Y}$ to have $\bar{Y}$ fresh, ensuring that the domain of $\varsigma^{(3)}$ is disjoint from $\bar{Y}$.

Our goal is then to show the following:

$$\mathscr{K} \Vdash [\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} S \asymp [\vec{v}_Y \vec{Y} \lhd \varsigma^{(3)}(B'_Y)] \texttt{=>>} \varsigma^{(3)}(S')$$

as we can connect $[\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} S$ and $\varsigma^{(3)}(T')$ together in an equality with lemma 3.3.8.

To do so, we expand the equality into two inequalities and employ the ($\Longleftarrow$) direction of lemma 3.4.6. We need to resort to using the "low-level" concept of entailment with ϕ, γ.

Let ϕ, γ be any assignments satisfying $\mathscr{K}$. Then, we are striving to show:

$$\begin{gathered}
\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_Y \implies \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(3)}(B'_Y) \curlywedge S \preceq \varsigma^{(3)}(S') \\
\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(3)}(B'_Y) \implies \phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_Y \curlywedge \varsigma^{(3)}(S') \preceq S
\end{gathered}$$

where we have expanded the $\asymp$ constraint into two $\preceq$ constraints.

It is *sufficient* to show:

$$\begin{gathered}
\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}.\ (\phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_Y \iff \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(3)}(B'_Y)) \\
\forall \vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma \models S \asymp \varsigma^{(3)}(S')
\end{gathered}$$

Let $\vec{A}$ be any element of $(\mathcal{T}^{\text{cl}})^{|\vec{Y}|}$. Before continuing, we remark that, by assumptions, we have $\bar{Y} \mathbin{\#} \text{ftv}(\mathscr{K})$. This allows us to apply lemma 3.3.4 to conclude that he assignments $\phi[\vec{Y} \mapsto \vec{A}], \gamma$ satisfy $\mathscr{K}$ as well.

As usual, we start with the easy part, that is:

$$\phi[\vec{Y} \mapsto \vec{A}], \gamma \models S \asymp \varsigma^{(3)}(S')$$

From the recursive call at (7c), we deduce:

$$\begin{gathered}
\phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(2)}(S) \asymp \varsigma^{(2)}(S') \\
\equiv \\
\phi[\vec{Y} \mapsto \vec{A}], \gamma \models S \asymp \varsigma^{(3)}(S')
\end{gathered}$$

where we have remove $\varsigma^{(2)}(\cdot)$ on the left-handside member due to being an $\mathcal{T}$ (that is, not containing any EC_H) and have used the fact that $\varsigma^{(3)} \upharpoonright \mathscr{K}^{(2)} = \varsigma^{(2)}$ (by Q-FEC1).

To conclude the case and the proof for $\mathcal{T}$-`FindOrCreateEC`, it remains to show:

$$\phi[\vec{Y} \mapsto \vec{A}], \gamma \models B_Y \iff \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(3)}(B'_Y)$$

The bounds constraint B_Y expands into:

$$\curlywedge \{L_i \preceq Y_i \curlywedge Y_i \preceq U_i : (Y_i, (L_i, U_i)) \in B_Y\}$$

while $\varsigma^{(3)}(B'_Y)$ expands into (with some rewriting):

$$\curlywedge \{\varsigma^{(3)}(L'_i) \preceq Y_i \curlywedge Y_i \preceq \varsigma^{(3)}(U'_i) : (Y_i, (L'_i, U'_i)) \in B'_Y\}$$

The Y_i is idempotent under $\varsigma^{(3)}(\cdot)$ because $\bar{Y}$ is fresh and thus free in $\mathscr{K}$.

From the unfolding of $\mathcal{B}$-`FindOrCreateEC` at (7a), we have for every $Y_i \in \vec{Y}$:

$$\begin{gathered}
\phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma^{(3)}(L_i) \asymp \varsigma^{(3)}(L'_i) \curlywedge \varsigma^{(3)}(U_i) \asymp \varsigma^{(3)}(U'_i) \\
\equiv \\
\phi[\vec{Y} \mapsto \vec{A}], \gamma \models L_i \asymp \varsigma^{(3)}(L'_i) \curlywedge U_i \asymp \varsigma^{(3)}(U'_i)
\end{gathered}$$

where $(L_i, U_i) = B_Y(Y_i)$ and $(L'_i, U'_i) = B'_Y(Y_i)$. We have again used the fact that $\varsigma^{(3)}$, the $\varsigma^{(a)}$ and $\varsigma^{(n)}$ within the loop all agree on common domains.

Then, it is a matter of assembling the pieces together with lemma 3.3.8.

□

A.6.2 $\mathcal{T}_{EC}$-`FindOrCreateEC`

Proof. By a case analysis on the form of T.

Case (1).

We are interested in showing that the returned values at (1a.i), (1c), (1d) and (1e) satisfy the postconditions of $\mathcal{T}_{EC}$-`FindOrCreateEC`.

Of these, only the return at (1a.i) is of interest, as the other ones are straightforward.

$\mathcal{T}_{EC}$-`TryFindApplied` and $\mathcal{T}_{EC}$-`CreateEC` requirements are trivially satisfied and their postconditions correspond to $\mathcal{T}_{EC}$-`FindOrCreateEC`.

Let us then show that the return at (1a.i) satisfies the postconditions. Since $\mathscr{K}$ is left unchanged, Q-FEC1, Q-FEC6 are trivially satisfied.

We now look at Q-FEC2. We have $\varsigma(\mathcal{R}(h)) = T_R(\mathcal{R}(h))$ which is defined, so $\varsigma(\mathcal{R}(h))\downarrow$. We remark that $T_R(\mathcal{R}(h))$ and h are contained in $\mathcal{M}(\mathcal{R}(h))$ as stated by K-INV5. As such, their underlying type must have the same kind (by K-INV8), therefore, since S has simple kind, so must $T_R(\mathcal{R}(h)) = \varsigma(\mathcal{R}(h))$.

Q-FEC3 and Q-FEC4 are straightforward.

It remains Q-FEC7: we are interested in showing $\mathscr{K} \Vdash \varsigma(T) \asymp \varsigma(\mathcal{R}(h))$. $\mathcal{T}_{EC}$-`Equiv` states we have $\mathscr{K} \Vdash \varsigma(T) \asymp \varsigma(\Theta(h))$. By K-INV5 and lemma A.2.2, we have $\mathscr{K} \Vdash \varsigma(\mathcal{R}(h)) \asymp \varsigma(\Theta(h))$. Applying lemma 3.3.8 concludes this subcase.

Case (2).

We would like to show that all exit points (2a.i), (2b) and (2c) hold the claim of $\mathcal{T}_{EC}$-`FindOrCreateEC` Since (2b) and (2c) are straightforward, we focus on (2a.i).

The postconditions Q-FEC1, Q-FEC5, Q-FEC6 are trivially satisfied.

For Q-FEC2, we first remark that $\Theta(h)$ and T must have the same kind under $\mathscr{K}$, otherwise, we would not have matched (2a). We deduce that $\mathcal{R}(h)$ and $\Theta(h)$ have the same kind under $\mathscr{K}$, so $\mathcal{R}(h)$ and T have the same kind under $\mathscr{K}$ as well.

Q-FEC3 and Q-FEC4 are straightforward.

We are once again left with Q-FEC7, for which we would like to prove

$$\begin{gathered}\mathscr{K} \Vdash \varsigma([\vec{v}\vec{Y} \lhd B_1] \texttt{=>>} S_1) \asymp \varsigma(\mathcal{R}(h)) \\ \equiv \\ \mathscr{K} \Vdash [\vec{v}\vec{Y} \lhd \varsigma(B_1)] \texttt{=>>} \varsigma(S_1) \asymp \varsigma(\mathcal{R}(h))\end{gathered} \tag{$\blacklozenge$}$$

We have "pushed" $\varsigma(\cdot)$ within the HK abstraction in the left member. This is possible thanks to the freshness of $\bar{Y}$.

By definition of ς, we have $\varsigma(\mathcal{R}(h)) = \Theta(T_R(\mathcal{R}(h)))$. Let us analyze the shape of $\Theta(T_R(\mathcal{R}(h)))$.

Since $\mathcal{R}(h)$ is higher-kinded, $\Theta(T_R(\mathcal{R}(h)))$ is of the form $[\vec{v}\vec{Y} \lhd B_U] \texttt{=>>} U$ (up to α-renaming). Because h and $T_R(\mathcal{R}(h))$ both belong to $\mathcal{M}(\mathcal{R}(h))$, we have by lemma A.2.1:

$$\begin{gathered}\mathscr{K} \Vdash \varsigma(\Theta(h)) \asymp \varsigma(\Theta(T_R(\mathcal{R}(h)))) \\ \equiv \\ \mathscr{K} \Vdash \varsigma([\vec{v}\vec{Y} \lhd B_2] \texttt{=>>} S_2) \asymp \varsigma([\vec{v}\vec{Y} \lhd B_U] \texttt{=>>} U) \\ \equiv \\ \mathscr{K} \Vdash [\vec{v}\vec{Y} \lhd \varsigma(B_2)] \texttt{=>>} \varsigma(S_2) \asymp [\vec{v}\vec{Y} \lhd \varsigma(B_U)] \texttt{=>>} \varsigma(U)\end{gathered}$$

Using the primary goal ($\blacklozenge$), it is then sufficient to prove:

$$\mathscr{K} \Vdash [\vec{v}\vec{Y} \lhd \varsigma(B_1)] \texttt{=>>} \varsigma(S_1) \asymp [\vec{v}\vec{Y} \lhd \varsigma(B_2)] \texttt{=>>} \varsigma(S_2)$$

To do so, we need to employ the ($\Longleftarrow$) direction of lemma 3.4.6. Then, the goal becomes:

$$\begin{gathered}\forall \vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_1) \implies \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_2) \Yup \varsigma(S_1) \preceq \varsigma(S_2) \\ \forall \vec{A} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_2) \implies \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_1) \Yup \varsigma(S_2) \preceq \varsigma(S_1)\end{gathered}$$

for all ϕ, γ satisfying $\mathscr{K}$, where the quantified $\vec{A}$ have the same kind as $\vec{Y}$.

We prove the first subgoal; the second one is analogous.

Let ϕ, γ be any assignments satisfying $\mathscr{K}$, and let any $\vec{A} \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}$ with the same kind as $\vec{Y}$. Then, since $\bar{Y}$ is fresh, $\phi[\vec{Y} \mapsto \vec{A}], \gamma$ satisfy $\mathscr{K}$ as well (by lemma 3.3.4).

From $\mathcal{T}_{EC}$-`Equiv`, we obtain $\phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(S_1) \preceq \varsigma(S_2)$.

Similarly, since $\text{dom}(B_1) = \text{dom}(B_2) = \bar{Y}$, we get from $\mathcal{B}_{EC}$-`Equiv`:

$$\phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_1) \implies \phi[\vec{Y} \mapsto \vec{A}], \gamma \models \varsigma(B_2)$$

thus concluding this subcase.

□

A.6.3 $\mathcal{T}_{EC}$-`TryFindApplied`

Proof. By analyzing all points of return. Returning NIL trivially satisfies the claim. We are therefore interested in the returns at (5b) and (6).

It is straightforward to check that the calls to $\mathcal{T}_{EC}$-`TryMatch` and $\mathcal{B}_{EC}$-`Satisified` at (1) and (3) are well-formed; the expression at (1) is meant to extend σ to satisfy the requirements of $\mathcal{B}_{EC}$-`Satisified`. The call to $\mathcal{T}_{EC}$-`TryFindECOfApplied` at (5a) requires applied to be closed and to have $\varsigma(\text{applied})$ defined. The first requirement is ensured by the `if` guarding this case. For the second requirement, we observe that the head $\varsigma(\mathcal{R}(h))$ is defined by validity of $\mathscr{K}$ and the definedness of $\varsigma(\vec{A})$ is guaranteed by $\mathcal{T}_{EC}$-`TryMatch`.

We can now go back on analyzing (5b) and (6).

We start with (5b). Q-FEC2, Q-FEC3 and Q-FEC4 are straightforward. The property Q-FEC7 is ensured by $\mathcal{T}_{EC}$-`TryFindECOfApplied`.

The return at (6) demands a bit more work. For Q-FEC2, ς is defined for applied as observed. Furthermore, applied is of simple kind like T. Q-FEC3 is vacuous by the guarding `if`.

For Q-FEC4, we need to ensure that $\mathcal{T}_{EC}$-`in-`Θ`-Inv`$(\text{applied}, \texttt{true}, \text{dom}(B_X)))$ holds – which is equivalent to showing that all $A_i \in \vec{A}$ satisfy the $\mathcal{T}_{EC}$-`in-`Θ`-Inv` predicate in a non-head position and bound variables $\text{dom}(B_X)$. By assumptions, T satisfies the predicate in a head position and bound variables $\text{dom}(B_X)$. Since T has a simple kind and that we have filtered out the cases where it is of the form $[a]$, X or $p.Q$, it must be a DNF or an applied type constructor. If T is an applied type constructor, all A_i must therefore appear in an argument position, thus satisfying $\mathcal{T}_{EC}$-`in-`Θ`-Inv`$(A_i, \texttt{false}, \text{dom}(B_X)))$. Otherwise, (if T is a DNF), then an A_i appearing in an argument position within a head of T satisfies the requirement. Indeed, if A_i is a head, it must be either of the form $[a]$, or of the form $[a]\left[\vec{V}\right]$, or of the form $Cls[\vec{S}]$ or $p.F[\vec{S}]$ (such that $\mathcal{T}_{EC}$-`in-`Θ`-Inv`$(\vec{S}, \texttt{false}, \text{dom}(B_X))$ holds). All these forms satisfy $\mathcal{T}_{EC}$-`in-`Θ`-Inv`$(A_i, \texttt{false}, \text{dom}(B_X))$.

Unsurprisingly, we are left with Q-FEC7. The goal is to prove:

$$\mathscr{K} \Vdash \varsigma(T) \asymp \varsigma(\mathcal{R}(h)[\vec{A}])$$

From $\mathcal{T}_{EC}$-`TryMatch`, we know that:

$$\mathscr{K} \Vdash \varsigma([\vec{Y} \mapsto \vec{A}]S) \asymp \varsigma(T)$$

We start by having a look at $\mathcal{R}(h)$. We remark that $T_R(\mathcal{R}(h))$ and h are contained in $\mathcal{M}(\mathcal{R}(h))$ as stated by K-INV5. As such, their underlying type must have the same kind (by K-INV8), therefore, $\Theta(T_R(\mathcal{R}(h)))$ is of the form $[\vec{v}_Y \vec{Y} \triangleleft B_U]$ `=>>` U (up to α-renaming of $\vec{Y}$).

These observations allows us to expand $\varsigma(\mathcal{R}(h)[\vec{A}])$, using the definition of EC_H-`SubstApply`:

$$\varsigma(\mathcal{R}(h)[\vec{A}]) = [\vec{Y} \mapsto \varsigma(\vec{A})]U$$

Then, our goal becomes:

$$\mathscr{K} \Vdash \varsigma([\vec{Y} \mapsto \vec{A}]S) \asymp [\vec{Y} \mapsto \varsigma(\vec{A})]U$$

as lemma 3.3.8 allows us to connect $\varsigma(T)$ and $\varsigma([\vec{Y} \mapsto \vec{A}]U)$ together.

We first remark that $\varsigma(U) = U$ because U does not contain any EC_H (by K-INV7).

Next, we claim the following equalities:

$$\varsigma([\vec{Y} \mapsto \vec{A}]S) = \varsigma([\vec{Y} \mapsto \varsigma(\vec{A})]S) = [\vec{Y} \mapsto \varsigma(\vec{A})]\varsigma(S)$$

The first equality comes from the fact that ς is idempotent under types with no EC_H, which is the case for $\varsigma(\vec{A})$. For the second equality, we observe that the codomain of $[\vec{Y} \mapsto \varsigma(\vec{A})]$ and the domain of ς are disjoint because the domain of ς is a subset of EC_H whereas $\varsigma(\vec{A})$ does not contain any EC_H. On the other hand, the domain of $[\vec{Y} \mapsto \varsigma(\vec{A})]$ and codomain of ς are subsets of $\mathcal{T}$. Thanks to the freshness of $\vec{Y}$, $\vec{Y}$ appears free in the codomain of ς. As such, we can swap ς and $[\vec{Y} \mapsto \varsigma(\vec{A})]$.

Our goal is then refined to:

$$\mathscr{K} \Vdash [\vec{Y} \mapsto \varsigma(\vec{A})]\varsigma(S) \asymp [\vec{Y} \mapsto \varsigma(\vec{A})]U$$

We then remember that $\Theta(h)$ is $[\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} S$, and that $T_R(\mathcal{R}(h))$ and h belong both to $\mathcal{M}(\mathcal{R}(h))$. Applying lemma A.2.1 gives us:

$$\begin{aligned} &\mathscr{K} \Vdash \varsigma([\vec{v}_Y \vec{Y} \lhd B_Y] \texttt{=>>} S) \asymp \varsigma([\vec{v}\vec{Y} \lhd B_U] \texttt{=>>} U) \\ &\equiv [\vec{v}_Y \vec{Y} \lhd \varsigma(B_Y)] \texttt{=>>} \varsigma(S) \asymp [\vec{v}_Y \vec{Y} \lhd \varsigma(B_U)] \texttt{=>>} U \end{aligned}$$

We now need to employ the lower-level concepts of assignments. Let ϕ, γ be any assignments satisfying $\mathscr{K}$.

Using the $\preceq$ part of the above equality with the ($\implies$) direction of the lemma 3.4.6 gives us:

$$\forall \vec{A}' \in (\mathcal{T}^{\text{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}'], \gamma \models \varsigma(B_Y) \implies \phi[\vec{Y} \mapsto \vec{A}'], \gamma \models \varsigma(B_U) \curlywedge \varsigma(S) \preceq U \qquad (\blacklozenge)$$

By $\mathcal{B}_{EC}$-`Satisified`, $\mathscr{K} \Vdash \varsigma(\vec{A}) \lhd \varsigma(B_Y)$. With some rewriting, the entailed constraints expands into:

$$\bigcurlywedge\{[\vec{Y} \mapsto \varsigma(\vec{A})]\varsigma(L_i) \preceq \varsigma(A_i) \curlywedge \varsigma(A_i) \preceq [\vec{Y} \mapsto \varsigma(\vec{A})]\varsigma(U_i) : (Y_i, (L_i, U_i)) \in B_Y\}$$

Since ϕ, γ satisfy $\mathscr{K}$, they also satisfy the expanded entailed constraint.

Before continuing, it is important to point out that $\varsigma(\vec{A})$ is not closed, so we cannot instantiate the $\vec{A}'$ in $(\blacklozenge)$ to $\varsigma(\vec{A})$. On the other hand, $(\phi, \gamma)\varsigma(\vec{A})$ is closed. Because it is closed and by composition of ϕ, γ, the assignments ϕ, γ satisfy the following as well:

$$\begin{aligned} \bigcurlywedge\{&[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]\varsigma(L_i) \preceq (\phi, \gamma)\varsigma(\vec{A}) \curlywedge \\ &(\phi, \gamma)\varsigma(\vec{A}) \preceq [\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]\varsigma(U_i) : (Y_i, (L_i, U_i)) \in B_Y\} \end{aligned}$$

Then, with lemma 3.3.2, we have:

$$\begin{gathered} \phi[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})], \gamma \models \bigcurlywedge\{\varsigma(L_i) \preceq Y_i \curlywedge Y_i \preceq \varsigma(U_i) : (Y_i, (L_i, U_i)) \in B_Y\} \\ \equiv \\ \phi[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})], \gamma \models \varsigma(B_Y) \end{gathered}$$

We can instantiate the quantified $\vec{A}'$ to $(\phi, \gamma)\varsigma(\vec{A})$ from $(\blacklozenge)$ and eliminate the implication, yielding:

$$\phi[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})], \gamma \models \varsigma(B_U) \curlywedge \varsigma(S) \preceq U$$

With lemma 3.3.2, we almost get the $\preceq$ part of the goal:

$$\phi, \gamma \models [\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]\varsigma(S) \preceq [\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]U$$

Because $(\phi, \gamma)\varsigma(\vec{A})$ is closed and by composition of ϕ, γ, the assignments ϕ, γ also satisfy:

$$\phi, \gamma \models [\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]\varsigma(S) \preceq [\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})]U$$

corresponding to the $\preceq$ part of the goal.

We now employ the $\succeq$ part of the equality and apply lemma 3.4.6 again:

$$\forall \vec{A}' \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{Y}|}.\ \phi[\vec{Y} \mapsto \vec{A}'], \gamma \models \varsigma(B_U) \implies \phi[\vec{Y} \mapsto \vec{A}'], \gamma \models \varsigma(B_Y) \curlywedge U \preceq \varsigma(S) \tag{$\bigstar$}$$

We can instantiate the quantified $\vec{A}'$ to $(\phi, \gamma)\varsigma(\vec{A})$ from ($\bigstar$). We note that we have obtained $\phi[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})], \gamma \models \varsigma(B_U)$ from the $\preceq$ derivation, enabling us to eliminate the implication:

$$\phi[\vec{Y} \mapsto (\phi, \gamma)\varsigma(\vec{A})], \gamma \models U \preceq \varsigma(S)$$

Applying lemma 3.3.2 and peeling off $(\phi, \gamma)(\cdot)$ from $\varsigma(\vec{A})$ conclude this case.

□

A.6.4 $\mathcal{T}_{EC}$-`CreateEC`

We organize the proof as follows. First, we argue about the validity of the constructed $\tilde{T}$. Next, we show that $\mathcal{K}^{(1)}$, $\mathcal{K}^{(2)}$, $\mathcal{K}^{(3)}$, $\mathcal{K}^{(4)}$ and $\mathcal{K}^{(5)}$ are all valid. We then show that they form an entailment chain and that they are all entailed by $\mathcal{K}$. We subsequently present some properties about the returned value. Finally, we gather all results and show that the postconditions are satisfied.

A.6.4.1 Validity of $\tilde{T}$

We are interested in showing that $\varsigma(\tilde{T})$ is defined and that $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(\tilde{T})$ holds. We remind that the last two arguments of $\mathcal{T}_{EC}$`-in-`Θ`-Inv` default to `true` (for the head position) and $\emptyset$ (for the set of bound type variables).

For branches (2a.i) and (2b.i) where $\mathrm{ftv}(T) \mathrel{\#} \bar{X}$ and $\tilde{T} = T$, $\varsigma(\tilde{T})$ is defined thanks to the preconditions. For the predicate satisfaction, no free type variable in T appear in $\bar{X}$, as such, $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(\tilde{T}, \texttt{true}, \bar{X})$ implies $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(\tilde{T}, \texttt{true}, \emptyset)$

Otherwise, the form of $\tilde{T}$ depends on the kind of T.

Starting with (2a.ii), we have $\tilde{T} = [\vec{v}_X\vec{X} \triangleleft B_X]\texttt{=>>}T$. For the definedness, we get:

$$\begin{gathered}\varsigma\big([\vec{v}_X\vec{X} \triangleleft B_X]\texttt{=>>}T\big)\downarrow \\ \equiv \\ \varsigma(B_X)\downarrow \wedge \varsigma(T)\downarrow\end{gathered}$$

which holds thanks to the preconditions.

To show $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(\tilde{T}) \equiv \mathcal{T}_{EC}$`-in-`$\Theta$`-Inv`$(\tilde{T}, \texttt{true}, \emptyset)$, we unfold the definition for the HK case:

$$\begin{gathered}\mathrm{dom}(B_X) = \bar{X} \wedge \mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_X, \emptyset) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T, \texttt{true}, \bar{X})\end{gathered}$$

$\mathrm{dom}(B_X) = \bar{X}$ and $\mathcal{B}_{EC}$`-in-`Θ`-Inv`$(B_X, \emptyset)$ hold by P-FEC2. $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(T, \texttt{true}, \bar{X})$ holds by the preconditions.

For the branch at (2b.ii), we have $\tilde{T} = [\vec{v}_X\vec{v}_Y\vec{X}\vec{Y} \triangleleft B_X, B_Y]\texttt{=>>}S$. Showing $\varsigma(\tilde{T})\downarrow$ proceeds similarly:

$$\begin{gathered}\varsigma\big([\vec{v}_X\vec{v}_Y\vec{X}\vec{Y} \triangleleft B_X, B_Y]\texttt{=>>}S\big)\downarrow \\ \equiv \\ \varsigma(B_X)\downarrow \wedge \varsigma(B_Y)\downarrow \wedge \varsigma(S)\downarrow\end{gathered}$$

$\varsigma(B_X)\downarrow$ holds by P-FEC2. $\varsigma(B_Y)$ and $\varsigma(S)$ are defined because $\varsigma(T)$ is defined (from which B_Y and S come).

To conclude this part, it remains to show $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(\tilde{T})$. Unfolding the predicate for the HK case, we get:

$$\begin{gathered}\operatorname{dom}(B_X, B_Y) = \bar{X} \cup \bar{Y} \wedge \mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}((B_X, B_Y), \emptyset) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S, \texttt{true}, \bar{X} \cup \bar{Y})\end{gathered}$$

We have $\operatorname{dom}(B_X, B_Y) = \bar{X} \cup \bar{Y}$ by P-FEC2 and by $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T, \texttt{true}, \bar{X})$.

For $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}((B_X, B_Y), \emptyset)$, we have at our disposal (by the preconditions):

$$\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_X, \emptyset) \wedge \mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(B_Y, \bar{X})$$

Expanding this conjunction and remembering that $\bar{X} = \operatorname{dom}(B_X)$ gives:

$$\begin{gathered}\operatorname{dom}(B_X) \mathbin{\#} \emptyset \wedge \operatorname{dom}(B_Y) \mathbin{\#} \operatorname{dom}(B_X) \wedge \\ \forall (L, U) \in \operatorname{Im}(B_X).\ \big[\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(L, \texttt{true}, \operatorname{dom}(B_X)) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U, \texttt{true}, \operatorname{dom}(B_X))\big] \wedge \\ \forall (L, U) \in \operatorname{Im}(B_Y).\ \big[\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(L, \texttt{true}, \operatorname{dom}(B_X) \cup \operatorname{dom}(B_X)) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U, \texttt{true}, \operatorname{dom}(B_X) \cup \operatorname{dom}(B_Y))\big]\end{gathered}$$

We can almost fold this expression into $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}((B_X, B_Y), \emptyset)$. To do so, we need to show that:

$$\begin{gathered}\forall (L, U) \in \operatorname{Im}(B_X).\ \big[\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(L, \texttt{true}, \operatorname{dom}(B_X)) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U, \texttt{true}, \operatorname{dom}(B_X))\big]\end{gathered}$$

implies

$$\begin{gathered}\forall (L, U) \in \operatorname{Im}(B_X).\ \big[\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(L, \texttt{true}, \operatorname{dom}(B_X) \cup \operatorname{dom}(B_Y)) \wedge \\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U, \texttt{true}, \operatorname{dom}(B_X) \cup \operatorname{dom}(B_Y))\big]\end{gathered}$$

As stated by the comment in (2b), it is possible to α-rename $\vec{Y}$ to have $\bar{Y} \mathbin{\#} B_X$. Then, for all $(L, U) \in \operatorname{Im}(B_X)$, we have $\operatorname{ftv}(L, U) \mathbin{\#} \operatorname{dom}(B_Y)$ which implies the desired conclusion. We can then fold back the expression back into $\mathcal{B}_{EC}\texttt{-in-}\Theta\texttt{-Inv}((B_X, B_Y), \emptyset)$.

Finally, $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S, \texttt{true}, \bar{X} \cup \bar{Y})$, stems from the assumptions $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T, \texttt{true}, \bar{X})$ which unfolds to the HK case (8) (since $T = [\vec{v}_Y \vec{Y} \triangleleft B_Y] \texttt{=>>} S$).

A.6.4.2 Validity for constructed $\mathscr{K}$

We start with $\mathscr{K}^{(1)}$. All but K-INV3, K-INV8, K-INV6, K-INV7, and K-INV15 are straightforward.

For K-INV3, we remark that $\varsigma(\tilde{T})\downarrow$ is equivalent to $[x] \in \tilde{T} \implies [x] \in \mathcal{Q}\texttt{-AllMembers}(\mathcal{Q})$.

K-INV8 holds because $\varsigma(\tilde{T})$ and $\tilde{T}$ have the same kind.

K-INV6 holds as well; $\tilde{T}$ satisfies the $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ predicate with default arguments as shown earlier. It is not required for $\varsigma(\tilde{T})$ (picked as the type representative) to satisfy the predicate.

For K-INV7, we remark that $\Theta^{(1)}(h_R) = \varsigma(\tilde{T})$ does not contain any EC_H. Indeed, we have shown that $\varsigma(\tilde{T})$ is defined; as such, all EC_H are substituted into types not containing EC_H thanks to K-INV7 of $\mathscr{K}$.

For K-INV15 we first remark that $\varsigma^{(1)}$ is an extension of ς. Then, because ς is defined on all T contained in Θ, so must $\varsigma^{(1)}$. It remains to show that $\varsigma^{(1)}$ is defined for $\tilde{T}$ and $\varsigma(\tilde{T})$. We note that $\varsigma(\tilde{T})$ does not contain any EC_H, so $\varsigma(\tilde{T})$ is idempotent under $\varsigma^{(1)}$. Because $[a]$ is fresh, it cannot appear in $\tilde{T}$ (which is built from T). Therefore, $\varsigma^{(1)}(\tilde{T}) = \varsigma(\tilde{T})$ for which we previously have shown definedness.

We now move on with $\mathscr{K}^{(2)}$, on which we only need to show that $G_{\mathcal{S}}$ satisfies K-INV1 and K-INV12. If we match (5a), these hold thanks to $\mathcal{T}_{EC}\texttt{-Composition}$. Otherwise, they trivially hold by validity of $\mathscr{K}^{(1)}$.

$\mathscr{K}^{(3)}$ validity is straightforward: we do not require anything special from $G_{EC}^{(3)}$ excepts that it may only refer to T_H and EC_H contained in $\mathrm{dom}(\Theta)$ and $\mathrm{dom}(\mathcal{M})$, which is ensured by $\mathcal{T}_{EC}$-`Composition`.

The validity of $\mathscr{K}^{(4)}$ is ensured by $\mathcal{T}_{EC}$-`Composition` as well. Finally, the validity of $\mathscr{K}^{(5)}$ is straightforward.

A.6.4.3 Entailment of $\mathscr{K}$

We start with the easy part. It is straightforward to check that we have $\mathscr{K}^{(1)} \Vdash \mathscr{K}^{(2)} \Vdash \mathscr{K}^{(3)} \Vdash \mathscr{K}^{(4)} \Vdash \mathscr{K}^{(5)}$. Indeed, the interpretation of $\mathscr{K}$ relies on $\mathcal{M}$, Θ, $\mathcal{Q}$, $\mathcal{I}$ and T_R and these structures are left untouched for the mentioned $\mathscr{K}$.

For $\mathscr{K} \Vdash \mathscr{K}^{(1)}$, we need to show $\mathscr{K}$-`to`-$\mathcal{C}(\mathscr{K}) \Vdash \mathscr{K}$-`to`-$\mathcal{C}(\mathscr{K}^{(1)})$, or in its expanded form:

$$\begin{gathered}
\underbrace{\curlywedge\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in E_{\preceq}\}}_{C_{\preceq}} \curlywedge \underbrace{\curlywedge\{\varsigma([r]) \asymp \varsigma(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M}, h \in \bar{h}\}}_{C_{\asymp}} \curlywedge \\
\underbrace{\curlywedge\{p : T : (p,T) \in \mathcal{I}\}}_{C_p} \\
\Vdash \\
\underbrace{\curlywedge\{\varsigma^{(1)}([x]) \preceq \varsigma^{(1)}([y]) : ([x],[y]) \in E_{\preceq}^{(1)}\}}_{C_{\preceq}^{(1)}} \curlywedge \underbrace{\curlywedge\{\varsigma^{(1)}([r]) \asymp \varsigma^{(1)}(\Theta^{(1)}(h)) : ([r],\bar{h}) \in \mathcal{M}^{(1)}, h \in \bar{h}\}}_{C_{\asymp}^{(1)}} \curlywedge \\
\underbrace{\curlywedge\{p : T : (p,T) \in \mathcal{I}^{(1)}\}}_{C_p^{(1)}}
\end{gathered}$$

Because $E_{\preceq} = E_{\preceq}^{(1)}$, and that $\varsigma^{(1)}$ and ς agree on common domain, the subtyping constraint set in the conclusion is trivially entailed by the antecedent. $\mathcal{I}$ is left untouched i.e. $\mathcal{I} = \mathcal{I}^{(1)}$. As such, C_p and $C_p^{(1)}$ cancel out.

Employing the definition of $\mathscr{K}^{(1)}$, we can rewrite $C_{\asymp}^{(1)}$ as follows:

$$\begin{gathered}
C_{\asymp}^{(1)} \\
\equiv \\
\underbrace{\curlywedge\{\varsigma([r]) \asymp \varsigma(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M}, h \in \bar{h}\}}_{=C_{\asymp}} \curlywedge \\
\curlywedge \{\varsigma^{(1)}([r]) \asymp \varsigma^{(1)}(\Theta^{(1)}(h)) : ([r],\bar{h}) \in \{([a],\{h_{\tilde{T}}, h_R\})\}, h \in \bar{h}\} \\
\equiv \\
C_{\asymp} \curlywedge \varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(\Theta^{(1)}(h_R)) \curlywedge \varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(\Theta^{(1)}(h_{\tilde{T}})) \\
\equiv \\
C_{\asymp} \curlywedge \Theta^{(1)}(h_R) \asymp \varsigma^{(1)}(\varsigma^{(1)}(\tilde{T})) \curlywedge \Theta^{(1)}(h_R) \asymp \varsigma^{(1)}(\tilde{T}) \\
\equiv \\
C_{\asymp} \curlywedge \varsigma(\tilde{T}) \asymp \varsigma(\tilde{T}) \curlywedge \varsigma(\tilde{T}) \asymp \varsigma(\tilde{T})
\end{gathered}$$

with $\varsigma^{(1)}(\tilde{T}) = \varsigma(\tilde{T})$ because $\tilde{T}$ cannot contain $[a]$ due to $[a]$'s freshness.

Since $C_{\asymp}$ is contained in the antecedent and that $\varsigma(\tilde{T}) \asymp \varsigma(\tilde{T}) \curlywedge \varsigma(\tilde{T}) \asymp \varsigma(\tilde{T})$ is a tautology, this concludes the proof for $\mathscr{K} \Vdash \mathscr{K}^{(1)}$.

A.6.4.4 Equivalence of T and the returned result T'

We are essentially interested in showing Q-FEC7: $\mathscr{K} \Vdash \varsigma^{(5)}(T) \asymp \varsigma^{(5)}(T')$. Because $\mathscr{K} \Vdash \mathscr{K}^{(5)}$, showing $\mathscr{K}^{(5)} \Vdash \varsigma^{(5)}(T) \asymp \varsigma^{(5)}(T')$ implies the primary goal.

We proceed by a case analysis analogous to the branching at (2).

Case $\mathrm{ftv}(T) \# \bar{X}$.

Then, $T' = [a]$ and the goal is to prove that $\mathscr{K}^{(5)} \Vdash \varsigma^{(5)}(T) \asymp \varsigma^{(5)}([a])$.

Since $[a] \notin T$, we have $\varsigma^{(5)}(T) = \varsigma(T)$. Furthermore, by definition, we have $\varsigma^{(5)}([a]) = \Theta^{(1)}(h_R) = \varsigma(\tilde{T})$. We also have $\tilde{T} = T$, thus concluding this case.

Case $\mathrm{ftv}(T) \cap \bar{X} \neq \emptyset \wedge \mathrm{kind}(T) = \star$.

Then we have $T' = [a]\big[\vec{X}\big]$. The goal is to prove that $\mathscr{K}^{(5)} \Vdash \varsigma^{(5)}(T) \asymp \varsigma^{(5)}([a]\big[\vec{X}\big])$.

As for the previous case, we have $\varsigma^{(5)}(T) = \varsigma(T)$. We also have:

$$\varsigma^{(5)}([a]) = \Theta^{(1)}(h_R) = [\vec{v}_X\vec{X} \lhd \varsigma(B_X)] \texttt{=>>} \varsigma(T)$$

As such, we get $\varsigma^{(5)}([a]\big[\vec{X}\big]) = [\vec{X} \mapsto \vec{X}]\varsigma(T) = \varsigma(T)$, concluding this case.

Case $\mathrm{ftv}(T) \cap \bar{X} \neq \emptyset \wedge \mathrm{kind}(T) = \kappa \Rightarrow \star$.

We have $T = [\vec{v}_Y\vec{Y} \lhd B_Y] \texttt{=>>} S$ and $T' = [\vec{v}_Y\vec{Y} \lhd B_Y] \texttt{=>>} [a]\big[\vec{X}, \vec{Y}\big]$. The goal is to prove:

$$\mathscr{K}^{(5)} \Vdash \varsigma^{(5)}([\vec{v}_Y\vec{Y} \lhd B_Y] \texttt{=>>} S) \asymp \varsigma^{(5)}([\vec{v}_Y\vec{Y} \lhd B_Y] \texttt{=>>} [a]\big[\vec{X}, \vec{Y}\big])$$

Due to the freshness of $[a]$, it cannot appear in B_Y or S. Then, it is equivalent to prove:

$$\mathscr{K}^{(5)} \Vdash [\vec{v}_Y\vec{Y} \lhd \varsigma(B_Y)] \texttt{=>>} \varsigma(S) \asymp [\vec{v}_Y\vec{Y} \lhd \varsigma(B_Y)] \texttt{=>>} \varsigma^{(5)}([a]\big[\vec{X}, \vec{Y}\big])$$

We have:

$$\varsigma^{(5)}([a]) = \Theta^{(1)}(h_R) = [\vec{v}_X\vec{v}_Y\vec{X}\vec{Y} \lhd \varsigma(B_X), \varsigma(B_Y)] \texttt{=>>} \varsigma(S)$$

and as such, $\varsigma^{(5)}([a]\big[\vec{X}, \vec{Y}\big]) = [\vec{X} \mapsto \vec{X}, \vec{Y} \mapsto \vec{Y}]\varsigma(S) = \varsigma(S)$ which concludes the case.

A.6.4.5 Postconditions

Q-FEC1 is straightforward because the substructures of $\mathscr{K}^{(5)}$ in question are all extension of $\mathscr{K}$. Validity has been shown in A.6.4.2. Q-FEC2, Q-FEC3 and Q-FEC4 are straightforward. Q-FEC6 and Q-FEC7 have been shown in A.6.4.3 and A.6.4.4 respectively.

A.7 Adding inequality

We would like to prove that `TryAddInequality` holds its claims.

Proof. Cases (1)-(3) are straightforward.

For (4), the only non-trivial points to prove are the validity of $\mathscr{K}'$ – boiling down to showing property K-INV10 – and the entailment $\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}$.

We start with a proof by contradiction that $G'_{\preceq}$ is forward-free: assuming the existence of a chain $[x_1], ..., [x_n]$ of length $x \geq 3$ in $G'_{\preceq}$ such that $([x_1], [x_n]) \in E'_{\preceq}$, we show that such chain cannot, in fact, exist.

We first remark that there must be an i such that $[x_i]$ and $[x_{i+1}]$ are equal to $[a]$ and $[b]$ respectively. Otherwise, the chain $[x_1], ..., [x_n]$ in $G'_{\preceq}$ is also a chain in $G_{\preceq}$ with $([x_1], [x_n])$ a forward edge, thus contradicting the forward-freeness of $G_{\preceq}$. Therefore, all $[x_j]$ with $j \leq i$ are contained in $\mathsf{allLower}$ and all $[x_k]$ with $k \geq i + 1$ are contained in $\mathsf{allUpper}$. In particular, $[x_1]$ is in $\mathsf{allLower}$ and $[x_n]$ in $\mathsf{allUpper}$. But $E'_{\preceq}$ removes all edges formed from the cross product between $\mathsf{allLower}$ and $\mathsf{allUpper}$, contradicting the existence of the chain $[x_1], ..., [x_n]$ with the forward edge $([x_1], [x_n])$.

We now proceed to prove the acyclicity by contradiction. Assuming there is a cycle in $G'_{\preceq}$, we show that there is a cycle in $G_{\preceq}$ – a contradiction.

Let $[x_1], ..., [x_n]$ be a cycle in $G'_{\preceq}$. We first remark that there cannot be cycles of length 1 due to $[a]$ and $[b]$ being distinct. Furthermore, there must be an i such that $[x_i]$ and $[x_{i+1}]$ are equal to $[a]$ and $[b]$; otherwise we would have a cycle in $G_{\preceq}$. Then, $[x_{i+1}], ..., [x_i]$ (equivalent to $[b], ..., [a]$) is chain in $G'_{\preceq}$. However, we have ensured with the check at (3) to filter out such cases.

It remains to prove $\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}$. To do so, we need to employ the interpretation of $\mathscr{K}$ as a constraint, defined with $\mathscr{K}\texttt{-to-}\mathcal{C}$, that is, we are interested in proving $\mathscr{K}\texttt{-to-}\mathcal{C}(\mathscr{K}) \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathscr{K}\texttt{-to-}\mathcal{C}(\mathscr{K}') \curlywedge \bigcurlywedge \mathsf{cstrts}$.

If we unfold the definitions, our goal is to show that the following entailment holds.

$$\begin{gathered}
\underbrace{\bigcurlywedge\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in E_{\preceq}\}}_{C_{\preceq}} \curlywedge \underbrace{\bigcurlywedge\{\varsigma([r]) \asymp \varsigma(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M}, h \in \bar{h}\}}_{C_{\asymp}} \curlywedge \\
\underbrace{\bigcurlywedge\{p : T : (p,T) \in \mathcal{I}\}}_{C_p} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \\
\Vdash \\
\underbrace{\bigcurlywedge\{\varsigma'([x]) \preceq \varsigma'([y]) : ([x],[y]) \in E'_{\preceq}\}}_{C'_{\preceq}} \curlywedge \underbrace{\bigcurlywedge\{\varsigma'([r]) \asymp \varsigma'(\Theta'(h)) : ([r],\bar{h}) \in \mathcal{M}', h \in \bar{h}\}}_{C'_{\asymp}} \curlywedge \\
\underbrace{\bigcurlywedge\{p : T : (p,T) \in \mathcal{I}'\}}_{C'_p} \curlywedge \underbrace{\{\varsigma(\Theta(\mathcal{D}([l]))) \preceq \varsigma(\Theta(\mathcal{D}([u]))) : [l] \in \mathsf{allLower}_{\mathsf{det}}, [u] \in \mathsf{allUpper}_{\mathsf{det}}\}}_{\mathsf{cstrts}}
\end{gathered}$$

Because all sub-structures except $G_{\preceq}$ are left untouched, we have $\varsigma = \varsigma'$ and $C_{\asymp}$, $C'_{\asymp}$ and C_p, C'_p are equal and cancel out.

To prove the entailment of $C'_{\preceq}$, it is sufficient to remark that $E'_{\preceq}$ is a subset of $E_{\preceq} \cup \{([a],[b])\}$; $C'_{\preceq}$ is therefore entailed by $C_{\preceq} \curlywedge \varsigma([a]) \asymp \varsigma([b])$.

For $\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathsf{cstrts}$, we claim it is sufficient to show:

$$\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \bigcurlywedge\{\varsigma([l]) \preceq \varsigma([u]) : [l] \in \mathsf{allLower}_{\mathsf{det}}, [u] \in \mathsf{allUpper}_{\mathsf{det}}\} \tag{$\blacklozenge$}$$

Indeed, for all such $[l]$ and $[u]$, we have $\mathscr{K} \Vdash \varsigma([l]) \asymp \varsigma(\Theta(\mathcal{D}([l]))) \curlywedge \varsigma([u]) \asymp \varsigma(\Theta(\mathcal{D}([u])))$ by lemma A.2.2 ($\mathcal{D}([l])$ and $\mathcal{D}([u])$ being defined, they are part of $\mathrm{dom}(\mathcal{M})$). Assuming the claim $\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \varsigma([l]) \preceq \varsigma([u])$ we get:

$$\begin{aligned}
\mathscr{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) &\Vdash \varsigma([l]) \preceq \varsigma([u]) \curlywedge \varsigma([l]) \asymp \varsigma(\Theta(\mathcal{D}([l]))) \curlywedge \varsigma([u]) \asymp \varsigma(\Theta(\mathcal{D}([u]))) \\
&\Vdash \varsigma(\Theta(\mathcal{D}([l]))) \preceq \varsigma(\Theta(\mathcal{D}([u])))
\end{aligned}$$

where the first entailment stems from combining the previous entailments using lemma 3.3.6 while the second is obtained by applying lemma 3.3.8.

Let us show ($\blacklozenge$). Let any $[l] \in \mathsf{allLower}_{\mathsf{det}}$ and $[u] \in \mathsf{allUpper}_{\mathsf{det}}$. From the definition of $\mathsf{allLower}_{\mathsf{det}}$ and $\mathsf{allUpper}_{\mathsf{det}}$, there is a chain between $[l]$ and $[a]$ and between $[b]$ and $[u]$ in $G_{\preceq}$. From the corollary of lemma A.2.3, we have $\mathscr{K} \Vdash \varsigma([l]) \preceq \varsigma([a]) \curlywedge \varsigma([b]) \preceq \varsigma([u])$ and therefore $\mathscr{K} \Vdash \varsigma([l]) \preceq \varsigma([u])$.

□

A.8 Merging

We proceed to prove the correctness of `Merge` and of `MergeHelper`.

A.8.1 `Merge`

Proof. We examine each statement, starting with the match at (1).

Case (1).

We claim that all execution paths reaching the end of the match expression without returning satisfy the following assertions ($\blacklozenge$):

1. $\mathscr{K}\text{-}\mathtt{Valid}(\mathscr{K}^{(1)}) \wedge \varsigma = \varsigma^{(1)} \curlywedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)}$
2. $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(1)}$

We note that 1. satisfy Q-MG1; for kind preservation, we remark that $\mathcal{T}_{EC}$-`kind` is based on ς which remains untouched. Points 2. implies Q-MG4.

Cases (1a) and (1c.i).

These cases trivially satisfy the above claim.

Case (1b).

We do not have to hold the claim because this case exits the function. Since $\mathscr{K} = \mathscr{K}'$ and that the returned constraint set is empty, it is sufficient to prove, for each postcondition set:

- Q-MG1: Nothing since $\mathscr{K} = \mathscr{K}'$.
- Q-MG2: $\bigcup \mathsf{toMerge} \subseteq \mathscr{K} \wedge \forall \{[x], [y]\} \in \mathsf{toMerge}.\ \mathcal{T}_{EC}\text{-}\mathtt{kind}(\mathscr{K}, [x]) = \mathcal{T}_{EC}\text{-}\mathtt{kind}(\mathscr{K}, [y])$
- Q-MG3: $([a], [b]), ([b], [a]) \notin E_{\preceq} \wedge \mathtt{ExistUndirChain}(G_{\preceq}, [a], [b]) \wedge L(\mathscr{K}, \mathsf{toMerge}) = 0$
- Q-MG4: $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash M(\mathscr{K}, \mathsf{toMerge})$

We proceed sequentially.

Postcondition Q-MG2.

We have $([a], [x_1]), ..., ([x_n], [b]) \in E_{\preceq}$; furthermore, all the EC_H of these pairs have the same kind (by K-INV10). They are all contained in $\mathscr{K}$ as well by K-INV2.

Postcondition Q-MG3.

As hinted earlier, it suffices to show $\mathscr{K}' = \mathscr{K} \wedge L(\mathscr{K}, \mathsf{toMerge}) = 0$. Indeed, we claim that:

$$([a], [b]), ([b], [a]) \notin E_{\preceq} \wedge \mathtt{ExistUndirChain}(G_{\preceq}, [a], [b])$$

holds in the analyzed case. Since we have matched a chain, $\mathtt{ExistUndirChain}(G_{\preceq}, [a], [b])$ is necessarily true. To show that $([a], [b]) \notin E_{\preceq}$ we use the "forward-free" property K-INV10 of the $G_{\preceq}$ graph. In our case, this invariant particularly specifies that $([a], [b]) \notin E_{\preceq}$. We also have $([b], [a]) \notin E_{\preceq}$: if it were the case, we would have a loop in $G_{\preceq}$, which is not possible thanks to K-INV10.

Finally, $L(\mathscr{K}, \mathsf{toMerge}) = 0$ results due to $\mathsf{toMerge}$ being an (unordered) chain.

Postcondition Q-MG4.

By lemma A.2.3, we have $\mathscr{K} \Vdash \varsigma([a]) \preceq \varsigma([x_1]) \curlywedge ... \curlywedge \varsigma([x_n]) \preceq \varsigma([b])$. The lemma 3.3.6 allows us to add $\varsigma([a]) \asymp \varsigma([b])$ to both side of the entailment, giving us:

$$\begin{aligned} \mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) &\Vdash \varsigma([a]) \asymp \varsigma([b]) \curlywedge \varsigma([a]) \preceq \varsigma([x_1]) \curlywedge ... \curlywedge \varsigma([x_n]) \preceq \varsigma([b]) \\ &\Vdash \varsigma([a]) \asymp \varsigma([x_1]) \curlywedge ... \curlywedge \varsigma([x_n]) \asymp \varsigma([b]) \\ &\Vdash M(\mathscr{K}, \mathsf{toMerge}) \end{aligned}$$

where we have employed the definition of $\asymp$ and applied lemma 3.3.6.

Case (1c.ii).

Analogous to (1b)

Case (1c.iii).

The call to `TryAddInequality` is well-formed by the preconditions of `Merge`.

We deduce the following:

1. $\mathcal{K}\texttt{-Valid}(\mathcal{K}^{(1)}) \wedge \varsigma = \varsigma^{(1)} \wedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)}$
2. $\mathcal{K} \curlywedge \varsigma([a]) \preceq \varsigma([b]) \Vdash \mathcal{K}^{(1)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(1)}$

Point 1. exactly matches the corresponding point of $(\blacklozenge)$. Point 2. implies the corresponding point of $(\blacklozenge)$.

Assertions (2) and (3).

The first assertion is useful because we only need to prove the second conjunct of Q-MG3. We proceed by proving that all execution paths from the case at (1) satisfy the assertion.

Cases (1a) and (1c.i) respectively witness the fact that $([a],[b]) \in E_{\preceq}$ and $([b],[a]) \in E_{\preceq}$. Cases (1b) and (1c.ii) exit and never reach the assertion. Case (1c.iii) witnesses that there are no undirected path between $[a]$ and $[b]$ under $G_{\preceq}$.

The second assertion is proven analogously and by noting that the call to `TryAddInequality` at (1c.iii) states that $([a],[b]) \in E_{\preceq}^{(1)}$ since there are no undirected paths between $[a]$ and $[b]$.

Match (4).

In the same vein as the match expression at (1), we claim the following hold at the end of the match $(\bigstar)$:

1. $\mathcal{K}\texttt{-Valid}(\mathcal{K}^{(2)}) \wedge \varsigma = \varsigma^{(2)} \wedge \mathcal{Q} = \mathcal{Q}^{(2)} \wedge T_R = T_R^{(2)} \wedge G_{\preceq}^{(1)} = G_{\preceq}^{(2)}$
2. $\bigcup \mathsf{toMerge}^{(2)} \subseteq \mathcal{K} \wedge \forall \{[x],[y]\} \in \mathsf{toMerge}^{(2)}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(2)},[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(2)},[y])$
3. $\mathcal{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathcal{K}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(2)} \curlywedge M(\mathcal{K}^{(2)}, \mathsf{toMerge}^{(2)})$
4. $\neg([a] \in \mathrm{dom}(\mathcal{D}^{(2)}) \wedge [b] \in \mathrm{dom}(\mathcal{D}^{(2)}))$

Case (4a).

We ensure first that the created constraint is well-formed: $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a])))$ and $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b])))$ must be defined and have the same kind. Since $[a]$ and $[b]$ are in $\mathcal{K}^{(1)}$, that $[a]$ and $[b]$ are the representatives of their class, that $[a]$ and $[b]$ are determined, and that $\mathcal{K}^{(1)}$ is valid, these expressions are well-defined. By K-INV5 and K-INV8, $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a])))$ has the same kind as $[a]$ (and similarly for $[b]$). Since $[a]$ and $[b]$ have by assumptions the same kind, so do $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a])))$ and $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b])))$.

We should now ensure that the call to `RemoveMember` is well-defined. $\mathcal{K}^{(2)}$ is naturally valid, and $\mathcal{D}^{(2)}([a]) \in \mathrm{dom}(\Theta^{(2)})$ by K-INV1. We have $\mathcal{D}^{(2)}([a]) \notin \mathrm{Im}(T_R^{(2)})$ by K-INV9.

Now that well-formedness is established, we show that the claim $(\bigstar)$ holds. The first point holds by `RemoveMember`. The second one holds as shown by the analysis for the well-formedness of $\mathsf{toMerge}^{(2)}$.

For the third one, `RemoveMember` tells us we have $\mathcal{K}^{(1)} \Vdash \mathcal{K}^{(2)}$. By applying lemma 3.3.6 to this entailment and to the facts $(\blacklozenge)$, we obtain $\mathcal{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathcal{K}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(1)}$. Since $\varsigma = \varsigma^{(2)}$, we have $M(\mathcal{K}^{(1)}, \mathsf{toMerge}^{(1)}) = M(\mathcal{K}^{(2)}, \mathsf{toMerge}^{(2)})$. It remains to show:

$$\mathcal{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a]))) \asymp \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b])))$$

We have:

$$\begin{aligned}
\mathcal{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) &\Vdash \mathcal{K}^{(1)} \\
&\Vdash \varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a]))) \curlywedge \\
&\quad\ \varsigma^{(1)}([b]) \asymp \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b]))) \\
&\Vdash \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a]))) \asymp \varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([b])))
\end{aligned}$$

The second entailment stems from the validity of $\mathcal{K}^{(1)}$ and lemma A.2.2.

To conclude this case, it remains to show point 4. From `RemoveMember`, we get to conclude that $[a] \notin \mathrm{dom}(\mathcal{D}^{(2)})$, a sufficient condition for point 4.

Case (4b).

We first ensure that the preconditions for `PropagateDeterminacy` are met. $\mathscr{K}^{(1)}$ is valid. $[b]$ is in $\mathscr{K}^{(1)}$ and the representative of its EC by the preconditions and the fact that $\mathcal{Q} = \mathcal{Q}^{(1)}$. $\varsigma^{(1)}(\Theta^{(1)}(\mathcal{D}^{(1)}([a])))$ is defined using a similar reasoning as for the previous case. By K-INV14, $\Theta^{(1)}(\mathcal{D}^{(1)}([a]))$ is determined under $\mathscr{K}^{(1)}$. From the previous observations, $\Theta^{(1)}(\mathcal{D}^{(1)}([a]))$ and $[b]$ have the same kind under $\mathscr{K}^{(1)}$. By K-INV9, $\mathcal{D}^{(1)}([a])$ is not contained in T_R and by K-INV6, $\Theta^{(1)}(\mathcal{D}^{(1)}([a]))$ satisfies $\mathcal{T}_{EC}$`-in-`Θ`-Inv`.

We obtain from `PropagateDeterminacy`:

1. $\mathscr{K}\texttt{-Valid}(\mathscr{K}') \wedge \mathcal{Q}^{(1)} = \mathcal{Q}' \wedge T_R^{(1)} = T_R' \wedge G_{\preceq}^{(1)} = G_{\preceq}'$
2. $\bigcup \mathsf{toMerge} \subseteq \mathscr{K} \wedge \forall \{[x],[y]\} \in \mathsf{toMerge}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}',[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}',[y])$
3. $\mathscr{K}^{(1)} \curlywedge \varsigma^{(1)}([b]) \asymp \varsigma^{(1)}(\mathcal{D}^{(1)}([a])) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}', \mathsf{toMerge}^{(2)})$

With the facts (♦), this in turns leads to (▲) [3]:

1. $\mathscr{K}\texttt{-Valid}(\mathscr{K}') \wedge \varsigma = \varsigma' \wedge \mathcal{Q} = \mathcal{Q}' \wedge T_R = T_R' \wedge G_{\preceq}^{(1)} = G_{\preceq}'$
2. $\bigcup \mathsf{toMerge}^{(2)} \subseteq \mathscr{K} \wedge \forall \{[x],[y]\} \in \mathsf{toMerge}^{(2)}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}',[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}',[y])$
3. $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts}' \curlywedge M(\mathscr{K}', \mathsf{toMerge}^{(2)})$

We now perform an analysis on the branches (4b.i) and (4b.ii) and show that the claim (★) holds.

Branch (4b.i).

By applying the same reasoning as in (4a) to the above observations, we get to show points 1-4.

Branch (4b.ii).

Points 1-3 are satisfied since $\mathscr{K}^{(2)} = \mathscr{K}'$. Point 4 is satisfied because $[b] \notin \mathrm{dom}(\mathcal{D}') = \mathrm{dom}(\mathcal{D}^{(2)})$.

Case (4c).

Analogous to (4b).

Case (4d).

Trivially holds thanks to the facts (▲).

Statement (5).

We should first ensure that the call to `MergeHelper` is well-defined. Since $\mathcal{Q} = \mathcal{Q}^{(2)}$, the first set of preconditions is held. We also have $\neg([a] \in \mathrm{dom}(\mathcal{D}^{(2)}) \wedge [b] \in \mathrm{dom}(\mathcal{D}^{(2)}))$ from the the facts (★). $([a],[b]) \in E_{\preceq}^{(2)} \vee ([b],[a]) \in E_{\preceq}^{(2)}$ holds as well as shown in the analysis of the assertion (3) and due to having $G_{\preceq}^{(1)} = G_{\preceq}^{(2)}$.

We obtain (■):

1. $\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(3)}) \wedge |\mathrm{dom}(\mathcal{M}^{(3)})| < |\mathrm{dom}(\mathcal{M}^{(2)})| \wedge ([x] \in \mathscr{K}^{(2)} \iff [x] \in \mathscr{K}^{(3)}) \wedge$
   $\forall [x] \in \mathscr{K}^{(2)}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(2)},[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(3)},[x])$
2. $\bigcup \mathsf{toMerge} \subseteq \mathscr{K}^{(2)} \wedge \forall \{[x],[y]\} \in \mathsf{toMerge}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(3)},[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(3)},[y])$
3. $\mathscr{K}^{(2)} \curlywedge \varsigma^{(2)}([a]) \asymp \varsigma^{(2)}([b]) \Vdash \mathscr{K}^{(3)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(3)}, \mathsf{toMerge}) \curlywedge \bigcurlywedge\{\varsigma^{(2)}([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(2)}\}$

Returned result (6).

We show that each postcondition set is satisfied.

Postcondition Q-MG1.

We remark that we have $\mathcal{Q} = \mathcal{Q}^{(2)}$. As such, $[x] \in \mathscr{K} \iff [x] \in \mathscr{K}^{(2)}$ is true, so we get $[x] \in \mathscr{K} \iff [x] \in \mathscr{K}^{(3)}$ as well. From (■), we know that the kinds of the EC are preserved from $\mathscr{K}^{(2)}$ to $\mathscr{K}^{(3)}$. Because $\varsigma = \varsigma^{(2)}$, the kinds are preserved from $\mathscr{K}$ to $\mathscr{K}^{(2)}$ and we therefore have $\forall [x] \in \mathscr{K}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K},[x]) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(3)},[x])$.

Postcondition Q-MG2.

Straightforward combination of the facts from (★), (■), and remembering that we have $[x] \in \mathscr{K} \iff [x] \in \mathscr{K}^{(3)}$.

---

[3] Notice that we do not necessarily have $G_{\preceq} = G_{\preceq}^{(1)}$, hence we simply report $G_{\preceq}^{(1)} = G_{\preceq}'$ back

Postcondition Q-MG3.
We only have to show $|\mathrm{dom}(\mathcal{M}^{(3)})| < |\mathrm{dom}(\mathcal{M})|$ because we have $\neg\big(([a],[b]),([b],[a]) \notin E_{\preceq}$, as stated by the proved assertion at (2). From (■), we have $|\mathrm{dom}(\mathcal{M}^{(3)})| < |\mathrm{dom}(\mathcal{M}^{(2)})|$. Since $\mathcal{Q} = \mathcal{Q}^{(2)}$, we have $\mathrm{dom}(\mathcal{M}) = \mathrm{dom}(\mathcal{M}^{(2)})$.

Postcondition Q-MG4.
Combining everything we have in our hands (and in particular exploiting the equality $\varsigma = \varsigma^{(2)}$), we have:

$$\begin{aligned}\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}^{(3)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(2)}, \mathsf{toMerge}^{(2)}) \curlywedge \\ M(\mathscr{K}^{(3)}, \mathsf{toMerge}) \curlywedge \bigcurlywedge\{\varsigma([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(3)}\}\end{aligned}$$

We almost have the desired expression: we have everything except $M(\mathscr{K}^{(3)}, \mathsf{toMerge}^{(2)})$. Expanding $M(\mathscr{K}^{(2)}, \mathsf{toMerge}^{(2)})$ (and using $\varsigma = \varsigma^{(2)}$ again), we have:

$$M(\mathscr{K}^{(2)}, \mathsf{toMerge}^{(2)}) \triangleq \bigcurlywedge \left\{\varsigma([x]) \asymp \varsigma([y]) : \{[x],[y]\} \in \mathsf{toMerge}^{(2)}\right\}$$

With lemma 3.3.8 and $\bigcurlywedge\{\varsigma([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(3)}\}$, we get:

$$\begin{aligned}\bigcurlywedge\{\varsigma([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(3)}\} \curlywedge \bigcurlywedge \left\{\varsigma([x]) \asymp \varsigma([y]) : \{[x],[y]\} \in \mathsf{toMerge}^{(2)}\right\} \\ \Vdash \bigcurlywedge \left\{\varsigma^{(3)}([x]) \asymp \varsigma^{(3)}([y]) : \{[x],[y]\} \in \mathsf{toMerge}^{(3)}\right\} \triangleq M(\mathscr{K}^{(3)}, \mathsf{toMerge}^{(2)})\end{aligned}$$

Putting back everything together:

$$\begin{aligned}\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}^{(3)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(2)}, \mathsf{toMerge}^{(2)}) \curlywedge \\ M(\mathscr{K}^{(3)}, \mathsf{toMerge}) \curlywedge \bigcurlywedge\{\varsigma([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(3)}\} \\ \Vdash \mathscr{K}^{(3)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(3)}, \mathsf{toMerge} \cup \mathsf{toMerge}^{(2)}) \curlywedge \\ \bigcurlywedge\{\varsigma([x]) \asymp \varsigma^{(3)}([x]) : [x] \in \mathscr{K}^{(3)}\}\end{aligned}$$

as desired.

□

A.8.2 `MergeHelper`

Proof. The proof is organized as follows. We first show that $\mathscr{K}^{(7)}$ is well-formed. We then show that it is entailed by $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b])$, and that it is valid. Then, we prove that $\mathscr{K}^{(8)}$ remains valid and entailed, and that the returned set of classes to merge is well-formed and entailed by $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b])$.

A.8.2.1 Well-formedness of $\mathscr{K}^{(7)}$

K-INV1. Straightforward. We remind that $\Theta^{(7)} = \Theta$ as it is left untouched.
K-INV2. Straightforward; we note that $\mathcal{Q}$`-AllMembers(`$\mathcal{Q}^{(7)}$`)` = $\mathcal{Q}$`-AllMembers(`$\mathcal{Q}^{(1)}$`)` = $\mathcal{Q}$`-AllMembers(`$\mathcal{Q}$`)` and that $\mathrm{dom}(\mathcal{M}^{(7)}) = \mathrm{dom}(\mathcal{M}^{(2)}) = (\mathrm{dom}(\mathcal{M}) \setminus \{[a],[b]\}) \cup \{[ab]\}$.
K-INV3. Trivially holds by the facts that $\Theta^{(7)} = \Theta$ and $\mathcal{Q}$`-AllMembers(`$\mathcal{Q}^{(7)}$`)` = $\mathcal{Q}$`-AllMembers(`$\mathcal{Q}$`)`.
K-INV4-K-INV7. Straightforward.
K-INV8. It is sufficient to prove that for any $T \in \Theta$, the kind of T under $\mathscr{K}$ and $\mathscr{K}^{(7)}$ are equal.
By inspecting $\mathcal{T}_{EC}$`-kind`, we remark that we only need to consider the case where T is of the form $[x]$.
Let $[r_1]$, $[r_2]$ be the representatives of $[x]$ under $\mathcal{Q}$ and $\mathcal{Q}^{(7)}$ respectively (i.e. $[r_1] = \mathcal{Q}$`-Find(`$\mathcal{Q}, [x]$`)` and $[r_2] = \mathcal{Q}$`-Find(`$\mathcal{Q}^{(7)}, [x]$`)`). Then, the goal is to show $\mathrm{kind}(\Theta(T_R([r_1]))) = \mathrm{kind}(\Theta(T_R^{(7)}([r_2])))$.
If $[r_2] = [ab]$, then $T_R^{(7)}([a]) = T_R([a])$. Furthermore, $[r_1]$ must be either $[a]$ or $[b]$. If $[r_1] = [a]$, we are done. Otherwise, we use the first set of precondition stating that $[a]$ and $[b]$ have the same kind under

$\mathscr{K}$. That is, we have $\mathrm{kind}(\varsigma([a])) = \mathrm{kind}(\varsigma([b]))$ which expands into $\mathrm{kind}(\Theta(T_R([a]))) = \mathrm{kind}(\Theta(T_R([b])))$. Combining this with $\mathrm{kind}(\Theta(T_R^{(7)}([a]))) = \mathrm{kind}(\Theta(T_R([a])))$ and the fact that $[ab]$ is $[a]$ or $[b]$ and that $[r_1] = [b]$ concludes this case.

Otherwise, $[r_2]$ must be different from $[a]$ and $[b]$. We must also have $[r_1] = [r_2]$ because the representatives for other EC members other than $[a]$ and $[b]$ remain unchained. As such, we have $T_R([r_1]) = T_R^{(7)}([r_1])$, concluding the proof for this invariant.

K-INV9. Straightforward.

K-INV10. The first part is straightforward. We show that $G_{\preceq}^{(7)}$ (which is equal to $G_{\preceq}^{(6)}$) is acyclic, forward-free and “kind-preserving”.

Acyclicity.

Without loss of generality, we assume that $[ab] = [a]$.

We proceed with a proof by contradiction: assuming that there is a cycle in $G_{\preceq}^{(7)}$, we show that there is a cycle or a forward edge in $G_{\preceq}$, a contradiction.

We observe three points:

1. There are cannot be cycles of length 1 in $G_{\preceq}^{(7)}$: by construction, there are no $[x]$ such that $([x],[x]) \in E_{\preceq}^{(7)}$.
2. All cycles in $G_{\preceq}^{(7)}$ must go through $[a]$: indeed, for any cycle $[x_1], ..., [x_n]$, there must be an i such that $([x_i],[x_{i+1}]) \in G_{\preceq}^{(7)}$ but $([x_i],[x_{i+1}]) \notin G_{\preceq}$. By the definition of $G_{\preceq}^{(7)}$, either $[x_i] = [a]$ or $[x_{i+1}] = [a]$.
3. All edges $([x],[y]) \in E_{\preceq}^{(7)}$ with $[x]$, $[y]$ different from $[a]$ and $[b]$ are in $E_{\preceq}$ as well.

With these observations, we deduce that there must exist a cycle C $[a], [x_1], ..., [x_n], [a]$ in $G_{\preceq}^{(7)}$ with $n \geq 1$ such that $([a],[x_1]), ([x_n],[a]) \in E_{\preceq}^{(7)}$ and $([x_i],[x_{i+1}]) \in E_{\preceq}^{(7)}$ with $1 \leq i < n$ and $[x_i] \neq [a]$, $1 \leq i \leq n$. Because $[b]$ does not appear in $G_{\preceq}^{(7)}$, point 3. of the above observation allows us to deduce that the edges $([x_i],[x_{i+1}])$ are in $E_{\preceq}$ as well.

We split the analysis in two parts on whether $([a],[b]) \in E_{\preceq}$ or $([b],[a]) \in E_{\preceq}$. The preconditions guarantees there is a direct link between $[a]$ and $[b]$. Then, for each part, we proceed by a case analysis on the origin of the connection between $([a],[x_1])$ and $([x_n],[a])$.

Case $([a],[b]) \in E_{\preceq}$***.***

Subcase $([a],[x_1]) \notin E_{\preceq} \wedge ([x_n],[a]) \notin E_{\preceq}$***.***

By construction of $E_{\preceq}^{(7)}$, we must have $([b],[x_1]) \in E_{\preceq}$ and $([x_n],[b]) \in E_{\preceq}$. Since the chain $[x_1], ..., [x_n]$ in $G_{\preceq}^{(7)}$ is also a chain in $G_{\preceq}$, $[b], [x_1], ..., [x_n], [b]$ is a chain and a cycle in $G_{\preceq}$.

Subcase $([a],[x_1]) \notin E_{\preceq} \wedge ([x_n],[a]) \in E_{\preceq}$***.***

Then we must have $([b],[x_1]) \in E_{\preceq}$. Because $([a],[b]) \in E_{\preceq}$, $[b], [x_1], ..., [x_n], [a], [b]$ is a chain and a cycle in $G_{\preceq}$.

Subcase $([a],[x_1]) \in E_{\preceq} \wedge ([x_n],[a]) \notin E_{\preceq}$***.***

Then we must have $([x_n],[b]) \in E_{\preceq}$. But $([a],[b]) \in E_{\preceq}$ actually constitutes a forward edge. Indeed, we have (in $G_{\preceq}$) the chain $[a], [x_1], ..., [x_n], [b]$ (with $n \geq 1$), so $([a],[b])$ is a forward edge.

Subcase $([a],[x_1]) \in E_{\preceq} \wedge ([x_n],[a]) \in E_{\preceq}$***.***

The cycle C in $G_{\preceq}^{(7)}$ is also a cycle in $G_{\preceq}$.

Case $([b],[a]) \in E_{\preceq}$***.***

Subcase $([a],[x_1]) \notin E_{\preceq} \wedge ([x_n],[a]) \notin E_{\preceq}$***.***

We must have $([b],[x_1]) \in E_{\preceq}$ and $([x_n],[b]) \in E_{\preceq}$. Then, $[b], [x_1], ..., [x_n], [b]$ is a chain and a cycle in $G_{\preceq}$.

Subcase $([a],[x_1]) \notin E_{\preceq} \wedge ([x_n],[a]) \in E_{\preceq}$***.***

Then we must have $([b],[x_1]) \in E_{\preceq}$. In that case, $([b],[a]) \in E_{\preceq}$ is a forward edge because $[b], [x_1], ..., [x_n], [a]$ is a chain in $G_{\preceq}$.

Subcase $([a],[x_1]) \in E_{\preceq} \wedge ([x_n],[a]) \notin E_{\preceq}$***.***

We then have $([x_n],[b]) \in E_{\preceq}$. Combined with the fact that $([b],[a]) \in E_{\preceq}$, $[a], [x_1], ..., [x_n], [b], [a]$ is a chain and a cycle in $G_{\preceq}$.

Subcase $([a], [x_1]) \in E_{\preceq} \wedge ([x_n], [a]) \in E_{\preceq}$**.**

The cycle C in $G_{\preceq}^{(7)}$ is also a cycle in $G_{\preceq}$.

Forward-free.

Without loss of generality, we assume that $[ab] = [a]$.

We proceed with a proof by contradiction: assuming that there is a chain $[x_1], ..., [x_n]$ of length $n \geq 3$ in $G_{\preceq}^{(7)}$ such that $([x_1], [x_n]) \in E_{\preceq}^{(7)}$, we show that such a chain cannot exist.

We observe there are two possible cases:

1. $([x_1], [x_n]) \notin E_{\preceq}$, so the forward edge has been explicitly added (i.e. $([x_1], [x_n])$ is in extra $\setminus$ forward).
2. $([x_1], [x_n]) \in E_{\preceq}$ but there is an i such that $([x_i], [x_{i+1}]) \notin E_{\preceq}$. Such an i must exist because $G_{\preceq}$ is forward-free. By the definition of $G_{\preceq}^{(7)}$, we have $[x_i] = [a]$ or $[x_{i+1}] = [a]$: in either case the chain passes through $[a]$.

Case $([x_1], [x_n]) \notin E_{\preceq}$**.**

Then, by construction of $G_{\preceq}^{(7)}$, the edge $([x_1], [x_n])$ must satisfy at least one of the following:

i. $([x_1], [x_n])$ is of the form $([x_1], [a])$ such that $([x_1], [a]), ([x_1], [b]) \in E_{\preceq}$.

ii. $([x_1], [x_n])$ is of the form $([a], [x_n])$ such that $([a], [x_n]), ([b], [x_n]) \in E_{\preceq}$.

We show that subcase i. leads to a contradiction. Subcase ii. is similar.

Because we either have $([a], [b]) \in E_{\preceq}$ or $([b], [a]) \in E_{\preceq}$ (by the preconditions of `MergeHelper`), we run into a contradiction.

Indeed, if $([a], [b]) \in E_{\preceq}$, then $[x_1], [a], [b]$ and $[x_1], [b]$ are chains in $G_{\preceq}$, and $([x_1], [b])$ constitutes a forward edge. Otherwise, $([b], [a]) \in E_{\preceq}$ implies that $[x_1], [b], [a]$ and $[x_1], [a]$ are chains in $G_{\preceq}$, and $([x_1], [a])$ constitutes a forward edge.

Case $([x_1], [x_n]) \in E_{\preceq}$**.**

We remark that $([x_1], [x_n])$ cannot be contained in the forward as it is removed from the built graph. As such, we must have $[x_1] \notin$ allLower or $[x_n] \notin$ allUpper.

We show that $[x_1] \notin$ allLower leads to a contradiction. The $[x_n] \notin$ allLower subcase is analogous.

By construction of the allLower set, $[x_1]$ cannot be $[a]$. Since we are in case 2, we also know that there is an $i > 1$ such that $[x_i] = [a]$. This in turn implies that $([x_{i-1}], [a]) \in E_{\preceq}$ or $([x_{i-1}], [b]) \in E_{\preceq}$ by the definition of $G_{\preceq}^{(7)}$. Then, the (possibly trivial) chain $[x_1], ...[x_{i-1}]$ in $G_{\preceq}^{(7)}$ is also a chain in $G_{\preceq}$, so $[x_1]$ is a transitive lower bound of $[a]$ or $[b]$, a contradiction.

Kind-preserving.

By the construction of $G_{\preceq}^{(7)}$, is it sufficient to show:

$$\forall [l] \in \mathsf{lower}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [l]) = \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [ab])$$
$$\forall [u] \in \mathsf{upper}.\ \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [u]) = \mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [ab])$$

where we exploit the fact that $\mathcal{T}_{EC}\texttt{-kind}$ relies on ς for retrieving the kind of an EC_H, which depends on Θ, T_R and $\mathcal{Q}$. These values are frozen at $\mathcal{K}^{(5)}$.

We show that the first point holds; the second one is proved analogously.

We note that for all $[l] \in$ lower, we have $\mathcal{Q}\texttt{-Find}(\mathcal{Q}^{(5)}, [l]) = [l]$ because $[l]$ is neither $[a]$ nor $[b]$ and that the representatives for ECs other than $[a]$ and $[b]$ are left untouched. As such, we have:

$$\mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [l]) = \mathrm{kind}(\Theta(T_R^{(5)}([l]))) = \mathrm{kind}(\Theta(T_R([l])))$$

For $\mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [ab])$, we have:

$$\mathcal{T}_{EC}\texttt{-kind}(\mathcal{K}^{(5)}, [ab]) = \mathrm{kind}(\Theta(T_R^{(5)}([ab]))) = \mathrm{kind}(\Theta(T_R([a])))$$

and by the assumptions, we furthermore get $\mathrm{kind}(\Theta(T_R([a]))) = \mathrm{kind}(\Theta(T_R([b])))$. To conclude the proof, we observe that we have $([l], [a]) \in E_{\preceq}$, in which case $\mathrm{kind}(\Theta(T_R([l]))) = \mathrm{kind}(\Theta(T_R([a])))$ or $([l], [b]) \in E_{\preceq}$, in which case $\mathrm{kind}(\Theta(T_R([l]))) = \mathrm{kind}(\Theta(T_R([b])))$.

K-INV11. Straightforward.

K-INV12-K-INV13. Trivially hold, as $G_{\mathcal{S}}$ and G_p are left untouched.

A.8.2.2 Entailment of $\mathscr{K}^{(7)}$

Next, we are interested in showing $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}^{(7)}$. That is, we would like to show $\mathscr{K}$-`to`-$\mathcal{C}(\mathscr{K}) \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}$-`to`-$\mathcal{C}(\mathscr{K}^{(7)})$, or in its expanded form:

$$
\begin{gathered}
\underbrace{\bigcurlywedge\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in E_{\preceq}\}}_{C_{\preceq}} \curlywedge \underbrace{\bigcurlywedge\{\varsigma([r]) \asymp \varsigma(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M}, h \in \bar{h}\}}_{C_{\asymp}} \curlywedge \\
\underbrace{\bigcurlywedge\{p : T : (p,T) \in \mathcal{I}\}}_{C_p} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \\
\Vdash \\
\underbrace{\bigcurlywedge\{\varsigma^{(7)}([x]) \preceq \varsigma^{(7)}([y]) : ([x],[y]) \in E_{\preceq}^{(7)}\}}_{C_{\preceq}^{(7)}} \curlywedge \underbrace{\bigcurlywedge\{\varsigma^{(7)}([r]) \asymp \varsigma^{(7)}(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M}^{(7)}, h \in \bar{h}\}}_{C_{\asymp}^{(7)}} \curlywedge \\
\underbrace{\bigcurlywedge\{p : T : (p,T) \in \mathcal{I}^{(7)}\}}_{C_p^{(7)}}
\end{gathered}
$$

$\mathcal{I}$ is left untouched, as such C_p and $C_p^{(7)}$ cancel out.

Before resuming, it is useful to remember that, in any valid $\mathscr{K}'$, $\varsigma'([x]) = \Theta(T'_R([x]))$ provided that $[x]$ is the representative of its EC (i.e. $\mathcal{Q}$-`Find`$(\mathcal{Q}', [x]) = [x]$). In our case, we have $\varsigma([a]) = \Theta(T_R([a]))$, $\varsigma([b]) = \Theta(T_R([b]))$ and $\varsigma^{(7)}([ab]) = \Theta(T_R^{(7)}([ab])) = \Theta(T_R([a]))$ (by construction of $T_R^{(7)} = T_R^{(5)}$ and because $\Theta = \Theta^{(7)}$).

We now proceed by showing that each of the conclusion is entailed by the antecedents.

Subtyping constraint set $C_{\preceq}^{(7)}$.

Employing the definition of $\mathscr{K}^{(7)}$, we can rewrite the subtyping constraint as follows:

$$
\begin{gathered}
C_{\preceq}^{(7)} \\
\equiv \\
\bigcurlywedge\{\varsigma([x]) \preceq \varsigma([y]) : ([x],[y]) \in E_{\preceq} \setminus (\mathsf{fwd} \cup \mathsf{abConns})\} \curlywedge \\
\bigcurlywedge\{\varsigma^{(7)}([x]) \preceq \varsigma^{(7)}([y]) : ([x],[y]) \in \mathsf{extra} \setminus \mathsf{fwd}\}
\end{gathered}
$$

For the first set of conjunction, we have used the fact that $\varsigma^{(7)}([x]) = \varsigma([x])$ (similar for $[y]$) because these $[x]$ and $[y]$ are different from $[a]$ and $[b]$, and are the representatives of their EC. We remark that the first set of conjunctions is trivially entailed by the antecedent.

For the second set of conjunctions, we proceed by showing $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \varsigma^{(7)}([x]) \asymp \varsigma^{(7)}([y])$ for each contained $([x],[y])$. We observe that each $([x],[y])$ must be of the form $([l],[ab])$ with $([l],[a]) \in E_{\preceq} \vee ([l],[b]) \in E_{\preceq}$, or of the form $([ab],[u])$ with $([a],[u]) \in E_{\preceq} \vee ([b],[u]) \in E_{\preceq}$.

We proceed by a case analysis on the form of $([x],[y])$. Since the analysis is similar for both cases, we only prove the first one.

Assuming $([x],[y]) = ([l],[ab])$, we are interested in showing $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \varsigma^{(7)}([l]) \preceq \varsigma^{(7)}([ab])$, or equivalently $\mathscr{K} \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b])) \Vdash \Theta(T_R([l])) \preceq \Theta(T_R([a]))$ as pointed out by the above remark, using the fact that $[l]$ is different from $[a]$ and $[b]$, and that $[l]$ is the representative of its EC under $\mathcal{Q}$ and $\mathcal{Q}^{(7)}$. By assumptions, we have $([l],[a]) \in E_{\preceq}$ or $([l],[b]) \in E_{\preceq}$, both of which are present in the antecedent in the form of a subtyping constraint.

If $([l],[a]) \in E_{\preceq}$, we get:

$$\begin{aligned}\mathscr{K} \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b])) &\Vdash \Theta(T_R([l])) \preceq \Theta(T_R([a])) \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b]))\\ &\Vdash \Theta(T_R([l])) \preceq \Theta(T_R([a]))\end{aligned}$$

On the other hand, if we have $([l],[b]) \in E_{\preceq}$, we obtain:

$$\begin{aligned}\mathscr{K} \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b])) &\Vdash \Theta(T_R([l])) \preceq \Theta(T_R([b])) \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b]))\\ &\Vdash \Theta(T_R([l])) \preceq \Theta(T_R([a]))\end{aligned}$$

Equality constraint set $C^{(7)}_{\asymp}$.
We similarly employ the definition of $\mathscr{K}^{(7)}$ to rewrite the equality constraint set:

$$\begin{gathered}C^{(7)}_{\asymp}\\ \equiv\\ \bigcurlywedge\big\{\underbrace{\varsigma^{(7)}([r])}_{=\varsigma([r])} \asymp \varsigma^{(7)}(\Theta(h)) : ([r],\bar{h}) \in \mathcal{M} \restriction (\mathrm{dom}(\mathcal{M}) \setminus \{[a],[b]\}), h \in \bar{h}\big\} \curlywedge\\ \bigcurlywedge\big\{\underbrace{\varsigma^{(7)}([ab])}_{=\varsigma([a])} \asymp \varsigma^{(7)}(\Theta(h)) : \bar{h} \in \mathcal{M}([a]), h \in \bar{h}\big\} \curlywedge\\ \bigcurlywedge\big\{\underbrace{\varsigma^{(7)}([ab])}_{=\varsigma([a])} \asymp \varsigma^{(7)}(\Theta(h)) : \bar{h} \in \mathcal{M}([b]), h \in \bar{h}\big\}\end{gathered}$$

We note that $\varsigma([r]) = \Theta(T_R([r]))$ and $\varsigma([a]) = \Theta(T_R([a]))$.

To keep going, we need to unfold the definition of $\varsigma^{(7)}$. We claim (without giving a proof) it is equivalent to $[\Theta(T_R([b])) \mapsto \Theta(T_R([a]))] \circ \varsigma$ where we have used composition of substitution instead of the usual extension operation. Then, for all $T : \mathcal{T}_{EC}$ such that $\varsigma^{(7)}(T)$ is defined, we have:

$$\varsigma^{(7)}(T) \equiv [\Theta(T_R([b])) \mapsto \Theta(T_R([a]))]\varsigma(T)$$

For the first conjunct, for all ranged $[r]$ and h, we remark that the antecedent possesses the equality $\Theta(T_R([r])) \asymp \varsigma(\Theta(h))$. Starting with that equality and with lemma 3.4.14, we then have:

$$\begin{gathered}\Theta(T_R([r])) \asymp \varsigma(\Theta(h)) \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b]))\\ \Vdash \Theta(T_R([r])) \asymp \big[\Theta(T_R([b])) \mapsto \Theta(T_R([a]))\big]\varsigma(\Theta(h))\\ \equiv\\ \Theta(T_R([r])) \asymp \varsigma^{(7)}(\Theta(h))\end{gathered}$$

The second conjunct is proved similarly.

For the third conjunct, for all ranged $[r]$ and h, we do not necessarily possess $\Theta(T_R([a])) \asymp \varsigma(\Theta(h))$ in the antecedent: we instead have $\Theta(T_R([b])) \asymp \varsigma(\Theta(h))$. Using the same reasoning as for the first two conjuncts, we obtain the following:

$$\Theta(T_R([b])) \asymp \varsigma(\Theta(h)) \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b])) \Vdash \Theta(T_R([b])) \asymp \varsigma^{(7)}(\Theta(h))$$

With lemma 3.3.6, we get the desired result:

$$\begin{gathered}\Theta(T_R([a])) \asymp \Theta(T_R([b])) \curlywedge \Theta(T_R([b])) \asymp \varsigma(\Theta(h)) \curlywedge \Theta(T_R([a])) \asymp \Theta(T_R([b]))\\ \Vdash \Theta(T_R([a])) \asymp \varsigma^{(7)}(\Theta(h))\end{gathered}$$

A.8.2.3 Validity of $\mathscr{K}^{(7)}$

We now know that $\mathscr{K}^{(7)}$ is well-formed. We are interested in showing K-INV14 and K-INV15. We start with the latter.

We pick an arbitrary T from $\mathrm{Im}(\Theta)$ and proceed to show $\varsigma^{(7)}(T)\downarrow$ by a structural induction on T. By validity of $\mathscr{K}$, we know that $\varsigma(T)$ is defined. Since the application of the IH is straightforward, we focus on the base cases.

***Case* $T = [x]$.**

***Subase* $\mathcal{Q}$*-Find*$(\mathcal{Q}^{(7)}, [x]) \in \{[a], [b]\}$.**

Then, $[x] \in \mathcal{Q}$-`MembersOf`$(\mathcal{Q}^{(7)}, [ab])$, so $\varsigma^{(7)}([x]) = \Theta(T_R^{(7)}([ab])) = \Theta(T_R([a])) = \varsigma([a])$ which is defined by validity of $\mathscr{K}$.

***Subase* $\mathcal{Q}$*-Find*$(\mathcal{Q}^{(7)}, [x]) = [r],\ [r] \notin \{[a], [b]\}$.**

Then, $[x] \in \mathcal{Q}$-`MembersOf`$(\mathcal{Q}^{(7)}, [r])$, and since $[r]$ is neither $[a]$ nor $[b]$, we have $\varsigma^{(7)}([x]) = \Theta(T_R^{(7)}([r])) = \Theta(T_R([r])) = \varsigma([r])$ which is defined.

***Case* $T = [x]\big[\vec{A}\big]$.**

***Subase* $\mathcal{Q}$*-Find*$(\mathcal{Q}^{(7)}, [x]) \in \{[a], [b]\}$.**

Similar to the $T = [x]$ case, we have $\varsigma^{(7)}([x]) = \Theta(T_R([a])) = \varsigma([a])$. As such, $\varsigma^{(7)}([x]\big[\vec{A}\big]) = \varsigma([x]\big[\vec{A}\big])$ which is defined.

***Subase* $\mathcal{Q}$*-Find*$(\mathcal{Q}^{(7)}, [x]) = [r], [r] \notin \{[a], [b]\}$.**

We have $\varsigma^{(7)}([r]) = \Theta(T_R([r])) = \varsigma([r])$. Unsurprisingly, we have $\varsigma^{(7)}([r]\big[\vec{A}\big]) = \varsigma([r]\big[\vec{A}\big])$ which is defined.

Showing K-INV14 is curiously quite straightforward. Given the fact that $\mathrm{Im}(\mathcal{D}^{(7)}) \subseteq \mathrm{Im}(\mathcal{D})$ and $\Theta = \Theta^{(7)}$, it is sufficient to show that, if a type $T \in \mathcal{T}_{EC}$ for which ς is defined and determined under $\mathscr{K}$, it is determined under $\mathscr{K}^{(7)}$ as well. To do so, we examine the definition of $\mathcal{T}_{EC}$-`IsDet`. We remark that the only non-trivial case is, of course, the DNF one. Then, we are interested in showing that, if two types U and V are provably not subtype of each other under $\mathscr{K}$, the absence of subtyping remains under $\mathscr{K}^{(7)}$. Inspecting $\mathcal{T}_{EC}$-`IsSubtype` reveals two base cases where `false` can be returned. The first one is the case where U and V are closed. The result does not depend on the knowledge structure, so the absence of subtyping remains. The second one is the case comparing two classes together where one class does not extend the other. This result is independent of the knowledge structure as well. As such, for any types U and V, if $\mathcal{T}_{EC}$-`IsSubtype`$(\mathscr{K}, U, V)$ returns `false`, the result of $\mathcal{T}_{EC}$-`IsSubtype`$(\mathscr{K}^{(7)}, U, V)$ is `false` as well.

A.8.2.4 Epilogue

It is straightforward to see that, at the end of the loop at (9), $\mathscr{K}^{(8)}$ is valid and $\mathscr{K}^{(7)} \Vdash \mathscr{K}^{(8)}$. We now focus on showing the postconditions, starting with the first set.

Postcondition Q-MGH1.

Straightforward, except for:

$$\forall [x] \in \mathscr{K}.\ \mathcal{T}_{EC}\text{-}\mathtt{kind}(\mathscr{K}, [x]) = \mathcal{T}_{EC}\text{-}\mathtt{kind}(\mathscr{K}^{(8)}, [x])$$
$$\equiv$$
$$\forall [x] \in \mathscr{K}.\ \mathrm{kind}(\varsigma([x])) = \mathrm{kind}(\varsigma^{(8)}([x]))$$

By `RemoveMember`, $\varsigma^{(8)} = \varsigma^{(7)}$, $\mathcal{Q}^{(8)} = \mathcal{Q}^{(7)}$ and $T_R^{(8)} = T_R^{(7)}$. We proceed on a case analysis on $\mathcal{Q}$-`Find`$(\mathcal{Q}, [x])$.

***Case* $\mathcal{Q}$*-Find*$(\mathcal{Q}, [x]) = [a]$.**

Then, $\mathrm{kind}(\varsigma([x])) = \mathrm{kind}(\Theta(T_R([a]))) = \mathrm{kind}(\Theta(T_R^{(8)}([a])))$

***Case* $\mathcal{Q}$*-Find*$(\mathcal{Q}, [x]) = [b]$.**

By the preconditions, $\mathrm{kind}(\varsigma([a])) = \mathrm{kind}(\varsigma([b]))$, so $\mathrm{kind}(\Theta(T_R([a]))) = \mathrm{kind}(\Theta(T_R([b])))$. As such, we have $\mathrm{kind}(\varsigma([x])) = \mathrm{kind}(\Theta(T_R([b]))) = \mathrm{kind}(\Theta(T_R([a]))) = \mathrm{kind}(\Theta(T_R^{(8)}([a])))$.

Case $\mathcal{Q}$-***Find***$(\mathcal{Q}, [x]) = [y],\ [y] \notin \{[a], [b]\}$**.**

Then, $\mathrm{kind}(\varsigma([x])) = \mathrm{kind}(\Theta(T_R([y]))) = \mathrm{kind}(\Theta(T_R^{(8)}([y])))$.

Postcondition Q-MGH2.

Straightforward. For $\mathsf{toMerge} \subseteq \mathscr{K}$, we recall that all $[x]$ in $\mathscr{K}$ are in $\mathscr{K}^{(8)}$ and vice-versa. By validity of $\mathscr{K}^{(8)}$ and construction of $\mathsf{occ}_{[a]}$ and $\mathsf{occ}_{[b]}$, $\mathcal{R}^{(8)}(h_1)$ and $\mathcal{R}^{(8)}(h_2)$ are defined and are in $\mathscr{K}^{(8)}$.

Postcondition Q-MGH3.

We essentially need to show:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \curlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}^{(8)}, \mathsf{toMerge}) \curlywedge \curlywedge\{\varsigma([x]) \asymp \varsigma^{(8)}([x]) : [x] \in \mathscr{K}\}$$

Note that we already have $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \mathscr{K}^{(8)}$.

We start by showing $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \curlywedge \mathsf{cstrts}$. Using a similar reasoning as the proof for `TryAddInequality`, we get $\mathscr{K}^{(8)} \Vdash \curlywedge \mathsf{cstrts}$. With the previous remark, we get to conclude this point.

To show $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash M(\mathscr{K}^{(8)}, \mathsf{toMerge})$, it is sufficient to analyze the branch at (10b). In particular, we are interested in showing:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \varsigma^{(8)}([ec_1]) \asymp \varsigma^{(8)}([ec_2])$$

Since $[ec_1]$ and $[ec_2]$ are the representatives of their EC, the goal entailment is equivalent to

$$\mathscr{K}^{(8)} \Vdash \Theta^{(8)}(T_R^{(8)}([ec_1])) \asymp \Theta^{(8)}(T_R^{(8)}([ec_2]))$$

By K-INV5, we obtain:

$$\begin{gathered} T_R^{(8)}([ec_1]) \in \mathcal{M}^{(8)}([ec_1]) \\ h_1 \in \mathcal{M}^{(8)}([ec_1]) \\ T_R^{(8)}([ec_2]) \in \mathcal{M}^{(8)}([ec_2]) \\ h_2 \in \mathcal{M}^{(8)}([ec_2]) \end{gathered}$$

Combining this with lemma A.2.1 gives:

$$\begin{gathered} \mathscr{K}^{(8)} \Vdash \varsigma^{(8)}(\Theta^{(8)}(T_R^{(8)}([ec_1]))) \asymp \varsigma^{(8)}(\Theta^{(8)}(h_1)) \curlywedge \\ \varsigma^{(8)}(\Theta^{(8)}(T_R^{(8)}([ec_2]))) \asymp \varsigma^{(8)}(\Theta^{(8)}(h_2)) \end{gathered}$$

The underlying types of T_R do not contain any EC_H (by K-INV7); as such these are idempotent under ς:

$$\begin{gathered} \mathscr{K}^{(8)} \Vdash \Theta^{(8)}(T_R^{(8)}([ec_1])) \asymp \varsigma^{(8)}(\Theta^{(8)}(h_1)) \curlywedge \\ \Theta^{(8)}(T_R^{(8)}([ec_2])) \asymp \varsigma^{(8)}(\Theta^{(8)}(h_2)) \end{gathered}$$

$\mathcal{T}_{EC}$-`Equiv` states:

$$\mathscr{K}^{(8)} \Vdash \varsigma^{(8)}(\Theta^{(8)}(h_1)) \asymp \varsigma^{(8)}(\Theta^{(8)}(h_2))$$

Combining these facts together with lemmas 3.3.6 and 3.3.8 concludes the proof for $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash M(\mathscr{K}^{(8)}, \mathsf{toMerge})$.

We are left with showing the entailment:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma([b]) \Vdash \curlywedge\{\varsigma([x]) \asymp \varsigma^{(8)}([x]) : [x] \in \mathscr{K}\}$$

Similarly to A.8.2.2 in the equality constraint set section, we claim $\varsigma^{(8)}$ is equivalent to $[T_R([b]) \mapsto T_R([a])] \circ \varsigma$. Then, for all $T : \mathcal{T}_{EC}$ such that $\varsigma^{(8)}(T)$ is defined, we have $\varsigma^{(8)}(T) \equiv [\Theta(T_R([b])) \mapsto \Theta(T_R([a]))]\varsigma(T)$.

Since $\varsigma([a]) = \Theta(T_R([a]))$ and $\varsigma([b]) = \Theta(T_R([b]))$, we apply lemma 3.3.8 with the definition of $\varsigma^{(8)}$ to obtain the entailment.

□

A.9 Propagation of determinacy

The propagation of determinacy is divided into multiple small functions. We prove these in the following order:

1. `PropagateHeadSubst`
2. `PropagateDNFRefresh`
3. `PropagateTrySubst`
4. `PropagateDeterminacy`
5. `GatherAffected`
6. `GatherPotentiallyAffected`

In particular, we prove the "main function" `PropagateDeterminacy` in 4th because we deem it is useful to know the underlying of the other functions before examining `PropagateDeterminacy`.

A.9.1 `PropagateHeadSubst`

Proof. The proof for `PropagateHeadSubst` is essentially based around the loop at (1). As usual, we first prove that the invariants hold before the first iteration and that they are maintained at the end of each iteration. Since the base case is trivial, we go ahead with the iterative step.

Branch (2).

We start by analyzing the branch at (2).

Statement (2a).

We argue that the call to $\mathcal{T}_{EC}$-`ApplyHeadSubstitution` is well-defined. From the LIH, we have $\varsigma = \varsigma^{(1)}$. As such, since $\varsigma(T)$ is defined (by the precondition), $\varsigma^{(1)}(T)$ is defined as well. $\varsigma^{(1)}([a])$ is defined because $[a] \in \mathcal{Q} = \mathcal{Q}^{(1)}$ and $\varsigma^{(1)}(\Theta^{(1)}(h))$ is defined as well by validity of $\mathscr{K}^{(1)}$ and due to having $h \in \mathscr{K}^{(1)}$. By assumptions, $[a]$ and T have the same kind under $\mathscr{K}$. Since $\varsigma = \varsigma^{(1)}$, we have $\text{kind}(\varsigma([a])) = \text{kind}(\varsigma^{(1)}([a]))$ (and similarly for T). The kinds are thus preserved under $\mathscr{K}^{(1)}$. Because h is not in T_R we have by K-INV6 that $\Theta^{(1)}(h)$ satisfies the $\mathcal{T}_{EC}$-`in`-Θ-`Inv` predicate. T satisfies it by assumptions.

We can now use $\mathcal{T}_{EC}$-`ApplyHeadSubstitution` postconditions:

- $S^{(1)}$ is defined and we have $\varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(T) \Vdash \varsigma^{(1)}(\Theta^{(1)}(h)) \asymp \varsigma^{(1)}(S^{(1)})$.
- $S^{(1)}$ and $\Theta^{(1)}(h)$ have the same kind.

Branch (2b).

We short-circuit. It is straightforward to check that the loop invariants are maintained.

Statement (2c).

The requirement for the call are satisfied, thanks to $S^{(1)}$ being defined and the escape hatch at (2b). We remark that ($\blacklozenge$):

- $\varsigma^{(1)}(S^{(2)})$ is defined and we have $\mathscr{K}^{(1)} \Vdash \varsigma^{(1)}(S^{(1)}) \asymp \varsigma^{(1)}(S^{(2)})$.
- With the previous observation and lemma 3.3.8, we have:

$$\mathscr{K}^{(1)} \curlywedge \varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(T) \Vdash \varsigma^{(1)}(\Theta^{(1)}(h)) \asymp \varsigma^{(1)}(S^{(2)})$$

- With the LIH, we furthermore have that the previous expression is entailed by $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T)$:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(1)} \curlywedge \varsigma^{(1)}([a]) \asymp \varsigma^{(1)}(T) \Vdash \varsigma^{(1)}(\Theta^{(1)}(h)) \asymp \varsigma^{(1)}(S^{(2)})$$

- $\mathcal{T}_{EC}$-`SimplifyDNF` states that, if $S^{(2)}$ is not an EC_H, then it must satisfy the $\mathcal{T}_{EC}$-`in`-Θ-`Inv` predicate.
- $S^{(1)}$, $S^{(2)}$ and $\Theta^{(1)}(h)$ have the same kind under $\mathscr{K}$ and $\mathscr{K}^{(1)}$. Due to the validity of $\mathscr{K}^{(1)}$, $\mathcal{R}^{(1)}(h)$ has these three types.

Branch (2d).

We notice that the implicit `else` branch of (2d.i) maintains the invariants. For (2d.i), we essentially need to show that $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \varsigma^{(1)}(S^{(2)}) \asymp \varsigma^{(1)}(\mathcal{R}^{(1)}(h))$. By K-INV5, $h \in \mathrm{Im}(\mathcal{R}^{(1)}(h))$ and by lemma A.2.1, we get that $\mathscr{K}^{(1)} \Vdash \varsigma^{(1)}(\Theta^{(1)}(h)) \asymp \varsigma^{(1)}(\mathcal{R}^{(1)}(h))$. Combining this observation with (⧫) concludes this subcase.

Branch (2e).

We should first ensure that the call to `UpdateMemberDetermined` is well-formed. $\mathscr{K}^{(1)}$ is valid by the LIH. $S^{(2)}$ is determined under $\mathscr{K}^{(1)}$ since we have matched the branch. h is in $\mathrm{dom}(\Theta^{(1)})$ by the LIH since $h \in \mathsf{headSubst}^{(1)}$. From the precondition of `PropagateHeadSubst` and the LIH, we have $h \notin \mathrm{Im}(T_R^{(1)})$ as well. $\varsigma^{(1)}(S^{(2)})$ is defined as stated in (2c). The predicate $\mathcal{T}_{EC}$`-in-`Θ`-Inv`$(S^{(2)})$ is satisfied due to $S^{(2)}$ not being an EC_H (this case is filtered out by the `if` at (2d)).

We now show that the four loop invariant are held at the end of the iteration (of course assuming that we have matched branch (2e)).

First loop invariant.

From the first stated postcondition of `UpdateMemberDetermined`, we have:

$$\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(n)}) \land \varsigma^{(1)} = \varsigma^{(n)} \land \mathcal{Q}^{(1)} = \mathcal{Q}^{(n)} \land T_R^{(1)} = T_R^{(n)} \land G_{\preceq}^{(1)} = G_{\preceq}^{(n)}$$

By the LIH, we can conclude that the first invariant holds.

Second loop invariant.

We are interested in showing:

$$\begin{gathered}\mathrm{dom}(\Theta^{(n)}) \cup \mathsf{headSubst}^{(n),\mathrm{c}} = \mathrm{dom}(\Theta) \land \\ \forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathrm{c}}.\ \Theta(\tilde{h}) = \Theta^{(n)}(\tilde{h}) \land \\ \mathsf{headSubst}^{(n)} \subseteq \mathrm{dom}(\Theta^{(n)})\end{gathered}$$

given the LIH and the second postcondition of `UpdateMemberDetermined` stating:

$$\begin{gathered}\mathrm{dom}(\Theta^{(n)}) \cup \{h\} = \mathrm{dom}(\Theta^{(1)}) \land \\ \forall \tilde{h} \in \mathrm{dom}(\Theta^{(1)}) \setminus \{h\}.\ \Theta^{(1)}(\tilde{h}) = \Theta^{(n)}(\tilde{h})\end{gathered}$$

We start by showing the third conjunct:

$$\begin{aligned}\mathsf{headSubst}^{(n)} &\subseteq \mathsf{headSubst}^{(1)} \setminus \{h\} \\ &\subseteq \mathrm{dom}(\Theta^{(1)}) \setminus \{h\} \\ &\subseteq \mathrm{dom}(\Theta^{(n)}) \setminus \{h\} \\ &\subseteq \mathrm{dom}(\Theta^{(n)})\end{aligned}$$

The first inequality comes from the definition of $\mathsf{headSubst}^{(n)}$. The second one comes from the LIH $\mathsf{headSubst}^{(1)} \subseteq \mathrm{dom}(\Theta^{(1)})$ where we have subtracted $\{h\}$ from both sides. The third one uses the second set of postconditions of `UpdateMemberDetermined` where we have subtracted $\{h\}$ from both sides as well.

Next, we show the first conjunct, starting with the equality given by `UpdateMemberDetermined`:

$$\begin{gathered}\mathrm{dom}(\Theta^{(n)}) \cup \{h\} = \mathrm{dom}(\Theta^{(1)}) \\ \Rightarrow \mathrm{dom}(\Theta^{(n)}) \cup \{h\} \cup \mathsf{headSubst}^{(1),\mathrm{c}} = \mathrm{dom}(\Theta^{(1)}) \cup \mathsf{headSubst}^{(1),\mathrm{c}} \\ \Rightarrow \mathrm{dom}(\Theta^{(n)}) \cup \mathsf{headSubst}^{(n),\mathrm{c}} = \mathrm{dom}(\Theta)\end{gathered}$$

In the second equality, we add $\mathsf{headSubst}^{(n),\mathrm{c}}$ to both members. In the third equality, we apply the LIH to obtain the right member; for the left member, we use the definition of $\mathsf{headSubst}^{(n),\mathrm{c}}$.

We now show that the second conjunct tying Θ and $\Theta^{(n)}$ holds.

From the LIH, we have:

$$\forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(1),\mathsf{c}}.\ \Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h})$$

From `UpdateMemberDetermined`, we get:

$$\forall \tilde{h} \in \mathrm{dom}(\Theta^{(1)}) \setminus \{h\}.\ \Theta^{(1)}(\tilde{h}) = \Theta^{(n)}(\tilde{h})$$

We remark that showing:

$$\mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathsf{c}} \subseteq \mathrm{dom}(\Theta^{(1)}) \setminus \{h\}$$

and

$$\mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathsf{c}} \subseteq \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(1),\mathsf{c}}$$

is sufficient to prove the desired result as we can connect Θ, $\Theta^{(1)}$ and $\Theta^{(n)}$ with an equality for all $\tilde{h} \in \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathsf{c}}$.

We remark that the second inclusion is straightforward as $\mathsf{headSubst}^{(1),\mathsf{c}} \subseteq \mathsf{headSubst}^{(n),\mathsf{c}}$.

For the first inclusion, we start with the LIH:

$$\begin{aligned} &\mathrm{dom}(\Theta) \subseteq \mathrm{dom}(\Theta^{(1)}) \cup \mathsf{headSubst}^{(1),\mathsf{c}} \\ \Rightarrow\ &\mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathsf{c}} \subseteq (\mathrm{dom}(\Theta^{(1)}) \setminus \mathsf{headSubst}^{(n),\mathsf{c}}) \cup (\mathsf{headSubst}^{(1),\mathsf{c}} \setminus \mathsf{headSubst}^{(n),\mathsf{c}}) \\ \Rightarrow\ &\mathrm{dom}(\Theta) \setminus \mathsf{headSubst}^{(n),\mathsf{c}} \subseteq \mathrm{dom}(\Theta^{(1)}) \setminus \mathsf{headSubst}^{(n),\mathsf{c}} \subseteq \mathrm{dom}(\Theta^{(1)}) \setminus \{h\} \end{aligned}$$

The second inequality comes from subtracting $\mathsf{headSubst}^{(n),\mathsf{c}}$ from both sides. For the third inequality, we have used the fact that $\mathsf{headSubst}^{(1),\mathsf{c}} \subseteq \mathsf{headSubst}^{(n),\mathsf{c}}$ to simplify $\mathsf{headSubst}^{(1),\mathsf{c}} \setminus \mathsf{headSubst}^{(n),\mathsf{c}}$ and that $h \in \mathsf{headSubst}^{(n),\mathsf{c}}$.

Third loop invariant.

From `UpdateMemberDetermined`, we have $\mathrm{dom}(\mathcal{D}^{(1)}) \uplus \mathsf{dets}' \subseteq \mathrm{dom}(\mathcal{D}^{(n)})$. From the LIH, we have $\mathrm{dom}(\mathcal{D}) \uplus \mathsf{dets} \subseteq \mathrm{dom}(\mathcal{D}^{(1)})$. We thus have $\mathsf{dets}' \# \mathrm{dom}(\mathcal{D})$ and $\mathrm{dom}(\mathcal{D}) \uplus (\mathsf{dets} \cup \mathsf{dets}') \subseteq \mathrm{dom}(\mathcal{D}^{(n)})$ as expected.

Fourth loop invariant.

`UpdateMemberDetermined` states that we have:

$$\mathscr{K}^{(1)} \curlywedge \varsigma^{(1)}(\Theta^{(1)}(h)) \asymp \varsigma^{(1)}(S^{(2)}) \Vdash \mathscr{K}^{(n)} \curlywedge \bigcurlywedge \mathsf{cstrts}' \curlywedge M(\mathscr{K}^{(n)}, \mathsf{toMerge}')$$

Combining this with (♦) gives:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(n)} \curlywedge \bigcurlywedge \mathsf{cstrts}' \curlywedge M(\mathscr{K}^{(n)}, \mathsf{toMerge}')$$

To conclude this case, it remains to show:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \bigcurlywedge \mathsf{cstrts}^{(n)} \curlywedge M(\mathscr{K}^{(n)}, \mathsf{toMerge})$$

By the LIH and the previous observation, we indeed have:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \bigcurlywedge \mathsf{cstrts}^{(n)}$$

By the LIH, we also have:

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash M(\mathscr{K}^{(1)}, \mathsf{toMerge})$$

Since $\varsigma^{(1)} = \varsigma^{(n)}$, we get

$$\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash M(\mathscr{K}^{(n)}, \mathsf{toMerge})$$

Branch (2f).

Similar to (2e), except for an additional precondition for `UpdateMember` requiring $\Theta^{(1)}(h)$ to not be determined (due to the non-determinacy of $S^{(2)}$). `ApplyHeadSubstitution` and $\mathcal{T}_{EC}$-`SimplifyDNF` guarantee that, if the result is not determined, then the argument must be non-determined; therefore, $\Theta^{(1)}(h)$ is not determined under $\mathscr{K}^{(1)}$.

Branch (3).

This case is similar to (2e) and (2f).

Returned result (4).

At the end of the loop, we have $\mathsf{headSubst}^{(1),c} = \mathsf{headSubst}$. Therefore, the postconditions of `PropagateHeadSubst` hold by the LIs.

□

A.9.2 `PropagateDNFRefresh`

Proof. The proof is similar to `PropagateHeadSubst`. A notable difference is the call to $\mathcal{T}_{EC}$-`SimplifyDNF` at (2) requiring $\Theta^{(1)}(h)$ to be a DNF. By assumptions, $\Theta(h)$ is a DNF. By the LIH, $\Theta(h) = \Theta^{(1)}(h)$ (since $h \in \mathsf{refreshDNF}^{(1)}$); as such, $\Theta^{(1)}(h)$ is a DNF.

□

A.9.3 `PropagateTrySubst`

Proof. The proof is similar to `PropagateHeadSubst` as well. $\mathcal{T}_{EC}$-`TryApplyHeadSubstitution` extra preconditions are satisfied by the preconditions of `PropagateTrySubst`.

□

A.9.4 `PropagateDeterminacy`

Proof. We proceed by examining each statement.

Statements (1) and (2).

It is straightforward to see that the calls to `GatherAffected` and `GatherPotentiallyAffected` are well-formed.

Statement (3).

The first set of preconditions of `PropagateHeadSubst` is guaranteed by the preconditions of `PropagateDeterminacy`. The second set is ensured thanks to the postconditions of `GatherAffected`.

We obtain (⧫):

1. $\mathscr{K}$-`Valid`$(\mathscr{K}^{(1)}) \wedge \varsigma = \varsigma^{(1)} \wedge \mathcal{Q} = \mathcal{Q}^{(1)} \wedge T_R = T_R^{(1)} \wedge G_{\preceq} = G_{\preceq}^{(1)}$
2. $\mathrm{dom}(\Theta^{(1)}) \cup \mathsf{headSubst} = \mathrm{dom}(\Theta) \ \wedge \big(\forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}.\ \Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h})\big)$
3. $\mathrm{dom}(\mathcal{D}) \subseteq \mathrm{dom}(\mathcal{D}^{(1)}) \wedge \big(\forall [x] \in \mathsf{dets}^{(1)}.\ [x] \in \mathrm{dom}(\mathcal{D}^{(1)}) \wedge [x] \notin \mathrm{dom}(\mathcal{D})\big)$
4. $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(1)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(1)} \curlywedge M(\mathscr{K}^{(1)}, \mathsf{toMerge}^{(1)})$

Statement (4).

We verify that the call to `PropagateDNFRefresh` is well-formed. The validity of $\mathscr{K}^{(1)}$ comes from the first point of (⧫). `GatherAffected` states that $\mathsf{refreshDNF} \# \mathrm{Im}(T_R)$, so we have $(\mathsf{refreshDNF} \setminus \mathsf{headSubst}) \# \mathrm{Im}(T_R)$ as well. From point 1 of (⧫), we have $T_R = T_R^{(1)}$, we therefore have $(\mathsf{refreshDNF} \setminus \mathsf{headSubst}) \# \mathrm{Im}(T_R^{(1)})$. To verify $(\mathsf{refreshDNF} \setminus \mathsf{headSubst}) \subseteq \mathrm{dom}(\Theta^{(1)})$, we start with the fact that $\mathsf{refreshDNF} \subseteq \mathrm{dom}(\Theta)$ and subtract $\mathsf{headSubst}$ from both side of the inequality, yielding:

$$\mathsf{refreshDNF} \setminus \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}$$

Next, we employ fact 2 of (⧫) and subtract $\mathsf{headSubst}$ from both side as well, giving:

$$\mathrm{dom}(\Theta^{(1)}) \setminus \mathsf{headSubst} = \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}$$

Combining these two observations together, we get the desired result:

$$\mathsf{refreshDNF} \setminus \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta) \setminus \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta^{(1)}) \setminus \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta^{(1)})$$

It remains to show $\forall \tilde{h} \in \mathsf{refreshDNF} \setminus \mathsf{headSubst}.\ \mathcal{T}_{EC}\text{-}\mathtt{IsDNF}(\Theta^{(1)}(\tilde{h}))$ in order to conclude the well-formedness of the call. Noticing that $\mathsf{refreshDNF} \setminus \mathsf{headSubst} \subseteq \mathrm{dom}(\Theta) \setminus \mathsf{headSubst}$, we can apply fact 2 of (⧫) to get:

$$\forall \tilde{h} \in \mathsf{refreshDNF} \setminus \mathsf{headSubst}.\ \Theta(\tilde{h}) = \Theta^{(1)}(\tilde{h})$$

From `GatherAffected`, we know that all $\tilde{h}$ in $\mathsf{refreshDNF}$ have their underlying type $\Theta(\tilde{h})$ being DNFs which concludes the point.

We can now extract the postconditions of `PropagateDNFRefresh` (★):

1. $\mathscr{K}\text{-}\mathtt{Valid}(\mathscr{K}^{(2)}) \land \varsigma^{(1)} = \varsigma^{(2)} \land \mathcal{Q}^{(1)} = \mathcal{Q}^{(2)} \land T_R^{(1)} = T_R^{(2)} \land G_{\preceq}^{(1)} = G_{\preceq}^{(2)}$
2. $\mathrm{dom}(\Theta^{(2)}) \cup (\mathsf{refreshDNF} \setminus \mathsf{headSubst}) = \mathrm{dom}(\Theta^{(1)}) \land$
   $\forall \tilde{h} \in \mathrm{dom}(\Theta^{(1)}) \setminus (\mathsf{refreshDNF} \setminus \mathsf{headSubst}).\ \Theta^{(1)}(\tilde{h}) = \Theta^{(2)}(\tilde{h})$
3. $\mathrm{dom}(\mathcal{D}^{(1)}) \subseteq \mathrm{dom}(\mathcal{D}^{(2)}) \land \big(\forall [x] \in \mathsf{dets}^{(2)}.\ [x] \in \mathrm{dom}(\mathcal{D}^{(2)}) \land [x] \notin \mathrm{dom}(\mathcal{D}^{(1)})\big)$
4. $\mathscr{K}^{(1)} \Vdash \mathscr{K}^{(2)} \curlywedge \bigcurlywedge \mathsf{cstrts}^{(2)} \curlywedge M(\mathscr{K}^{(2)}, \mathsf{toMerge}^{(2)})$

Statement (5).

We check that the first set of precondition of `PropagateTrySubst` are held. Validity of $\mathscr{K}^{(2)}$ is ensured thanks to the point 1 of (★). Since $\varsigma = \varsigma^{(2)}$ and that $\varsigma(T)\downarrow$, we have $\varsigma^{(2)}(T)\downarrow$ as well. $\mathcal{T}_{EC}\text{-}\mathtt{in}\text{-}\Theta\text{-}\mathtt{Inv}(T)$ stems from the preconditions.

Next, we need to show that the second set of preconditions holds as well. We have $\mathrm{dom}(\mathsf{trySusbt}) \# \mathrm{Im}(T_R^{(2)})$ by the postcondition of `GatherPotentiallyAffected` and the fact that $T_R = T_R^{(2)}$. To show $\mathrm{dom}(\mathsf{trySubst}) \subseteq \mathrm{dom}(\Theta^{(2)})$, we first observe that $\mathrm{dom}(\mathsf{trySubst})$ is disjoint from $\mathsf{headSubst}$ and $\mathsf{refreshDNF}$. Indeed, for all $h_1 \in \mathrm{dom}(\mathsf{trySubst})$, we have $\mathtt{IsAbsAppTycon}(\Theta(h_1))$. For the $h_2 \in \mathsf{headSubst}$, we have the inverse, that is, $\neg\mathtt{IsAbsAppTycon}(\Theta(h_2))$. Finally, since the $h_3 \in \mathsf{refreshDNF}$ are DNFs, they cannot be an applied abstract type constructor by definition of $\mathcal{T}_{EC}\text{-}\mathtt{IsDNF}$ and $\mathcal{T}_{EC}\text{-}\mathtt{IsAbsAppTycon}$ (we thus have $\neg\mathtt{IsAbsAppTycon}(\Theta(h_3))$). Combining this fact with the postcondition of `GatherPotentiallyAffected` yields:

$$\mathrm{dom}(\mathsf{trySubst}) \subseteq \mathrm{dom}(\Theta) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF})$$

Picking fact 2 from (⧫) and subtracting by $\mathsf{headSubst} \cup \mathsf{refreshDNF}$ gives:

$$\mathrm{dom}(\Theta^{(1)}) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}) = \mathrm{dom}(\Theta) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF})$$

We do the same with fact 2 from (★):

$$\mathrm{dom}(\Theta^{(2)}) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}) = \mathrm{dom}(\Theta^{(1)}) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF})$$

Gluing these facts together gives us the expected result:

$$
\begin{aligned}
\mathrm{dom}(\mathsf{trySubst}) &\subseteq \mathrm{dom}(\Theta) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}) \\
&\subseteq \mathrm{dom}(\Theta^{(1)}) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}) \\
&\subseteq \mathrm{dom}(\Theta^{(2)}) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}) \\
&\subseteq \mathrm{dom}(\Theta^{(2)})
\end{aligned}
$$

We now show that $\forall \tilde{h} \in \mathrm{dom}(\mathsf{trySubst}).\ \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(\tilde{h}))$. Adapting and combining fact 2 from (♦) and (★) gives us:

$$\forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF}).\ \Theta(\tilde{h}) = \Theta^{(2)}(\tilde{h})$$

All $\tilde{h}$ in $\mathrm{dom}(\mathsf{trySubst})$ satisfy $\mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta(\tilde{h}))$ and since:

$$\mathrm{dom}(\mathsf{trySubst}) \subseteq \mathrm{dom}(\Theta) \setminus (\mathsf{headSubst} \cup \mathsf{refreshDNF})$$

they also satisfy $\mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(\Theta^{(2)}(\tilde{h}))$.

To conclude the well-formedness of the call to `PropagateTrySubst`, it remains to show:

$$
\begin{gathered}
\forall U \in \bigcup \mathrm{Im}(\mathsf{trySubst}).\ \big[\varsigma(U)^{(2)}{\downarrow} \wedge \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(U) \wedge \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U) \wedge \\
\mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(2)}, U) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(2)}, T)\big]
\end{gathered}
$$

All of these conjuncts stem from `GatherPotentiallyAffected` and the fact that $\varsigma = \varsigma^{(2)}$ (we remind that $\mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}^{(2)}, U)$ expands to $\mathrm{kind}(\varsigma^{(2)}(U)) = \mathrm{kind}(\varsigma(U))$, the same applies for T).

This little journey allows us to extract the postcondition of `PropagateDNFRefresh`:

- $\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(3)}) \wedge \varsigma^{(2)} = \varsigma^{(3)} \wedge \mathcal{Q}^{(2)} = \mathcal{Q}^{(3)} \wedge T_R^{(2)} = T_R^{(3)} \wedge G_{\preceq}^{(2)} = G_{\preceq}^{(3)}$
- $\mathrm{dom}(\mathcal{D}^{(2)}) \subseteq \mathrm{dom}(\mathcal{D}^{(3)}) \wedge \big(\forall [x] \in \mathsf{dets}^{(3)}.\ [x] \in \mathrm{dom}(\mathcal{D}^{(3)}) \wedge [x] \notin \mathrm{dom}(\mathcal{D}^{(2)})\big)$
- $\mathscr{K}^{(2)} \Vdash \mathscr{K}^{(3)} \curlywedge \mathop{\Large\curlywedge} \mathsf{cstrts}^{(3)} \curlywedge M(\mathscr{K}^{(3)}, \mathsf{toMerge}^{(3)})$

We did not include the properties satisfied by $\mathrm{dom}(\Theta^{(2)})$ and $\mathrm{dom}(\Theta^{(3)})$ as we do no longer need them.

Statement (6).

We now connect all facts together (■):

1. $\mathscr{K}\texttt{-Valid}(\mathscr{K}^{(4)}) \wedge \varsigma = \varsigma^{(4)} \wedge \mathcal{Q} = \mathcal{Q}^{(4)} \wedge T_R = T_R^{(4)} \wedge G_{\preceq} = G_{\preceq}^{(4)}$
2. $\mathrm{dom}(\mathcal{D}) \subseteq \mathrm{dom}(\mathcal{D}^{(4)}) \wedge \big(\forall [x] \in \mathsf{dets}^{(4)}.\ [x] \in \mathrm{dom}(\mathcal{D}^{(4)}) \wedge [x] \notin \mathrm{dom}(\mathcal{D})\big)$
3. $\mathscr{K} \curlywedge \varsigma([a]) \asymp \varsigma(T) \Vdash \mathscr{K}^{(4)} \curlywedge \mathop{\Large\curlywedge} \mathsf{cstrts}^{(4)} \curlywedge M(\mathscr{K}^{(4)}, \mathsf{toMerge}^{(4)})$

For point 3, we have used the fact that $\varsigma = \varsigma^{(1)} = \varsigma^{(2)} = \varsigma^{(3)} = \varsigma^{(4)}$, which allows us to combine the $M(\mathscr{K}^{(i)}, \mathsf{toMerge}^{(i)})$ together.

Loop (7).

The loop invariants are held before entering the loop, as shown in (■).

To prove correctness, we have to show that the call to `PropagateDeterminacy` is well-formed, find a measure that is decreased for the recursive call, and finally, that the loop invariants are held after the end of the iteration.

Starting with well-formedness, the validity of $\mathscr{K}^{(4)}$ is ensured by the LIH. $[x]$ is necessarily contained in $\mathscr{K}^{(4)}$ by the LIH and the validity of $\mathscr{K}^{(4)}$. Because $[x] \in \mathrm{dom}(\mathcal{D}^{(4)})$, we also have $\mathcal{Q}\texttt{-Find}(\mathcal{Q}^{(4)}, [x])$ by validity of $\mathscr{K}^{(4)}$. $\Theta^{(4)}(\mathcal{D}^{(4)}([x]))$ is defined and determined under $\mathscr{K}^{(4)}$ by validity of $\mathscr{K}^{(4)}$. Again, by validity of $\mathscr{K}^{(4)}$, $\Theta^{(4)}(\mathcal{D}^{(4)}([x]))$ satisfies the $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ invariant and has the same kind as $[x]$.

Next, we find a measure and show that the measure of the arguments of the recursive call is decreased with respect to the current instance. We choose $m(\mathscr{K}) = |\mathcal{Q}\texttt{-AllMembers}(\mathscr{K})| - |\mathrm{dom}(\mathcal{D})|$ and show that

$m(\mathscr{K}^{(4)}) < m(\mathscr{K})$. It is only defined for valid $\mathscr{K}$'s where $\mathrm{dom}(\mathcal{D}) \subseteq \mathcal{Q}\texttt{-AllMembers}(\mathscr{K})$.
The first observation we make is that $\mathcal{Q}\texttt{-AllMembers}(\mathscr{K}) = \mathcal{Q}\texttt{-AllMembers}(\mathscr{K}^{(4)})$, as implied by the LIH $\mathcal{Q} = \mathcal{Q}^{(4)}$. It is thus sufficient to show that $|\mathrm{dom}(\mathcal{D})| < |\mathrm{dom}(\mathcal{D})^{(4)}|$. By the LIH, we have $\mathrm{dom}(\mathcal{D}) \uplus \mathsf{dets} \subseteq \mathrm{dom}(\mathcal{D})^{(4)}$. Because $\mathrm{dom}(\mathcal{D})$ and dets are disjoint and that dets has at least one element (otherwise we would not have entered the loop), we obtain $|\mathrm{dom}(\mathcal{D})| < |\mathrm{dom}(\mathcal{D})| + |\mathsf{dets}| \leq |\mathrm{dom}(\mathcal{D}^{(4)})|$.

Showing the loop invariants hold is a matter of a straightforward application of the LIH and the IH.

Returned result (8).
The postconditions of `PropagateDeterminacy` are respected by the LI.

□

A.9.5 `GatherAffected`

Proof. By induction on the measure $m(\mathsf{processedECs}) = |\mathrm{dom}(\mathcal{M})| - |\mathsf{processedECs}|$. The measure only applies for $\mathsf{processedECs} \subseteq \mathrm{dom}(\mathcal{M})$, which is ensured by the precondition for the current instance.

As usual, we proceed by examining statement. We remark that the early return at (1) trivially satisfies the postconditions. Furthermore, the loop invariants of (2) are trivially satisfied as well before the first iteration.

We are now interested in proving that the invariants hold at the end of each iteration.

Branch (2a)

We should first prove that the call to `GatherAffected` is well defined. Thanks to the validity of $\mathscr{K}$, we indeed have $\mathcal{Q}\texttt{-Find}(\mathcal{Q}, \mathcal{R}(h)) = \mathcal{R}(h)$ (by K-INV2). We also have $\mathsf{processedECs}^{(1)} \cup \{[b], \mathcal{R}(h)\} \subseteq \mathrm{dom}(\mathcal{M})$ by the precondition and the LIH. Next, we prove that the measure decreases: $m(\mathsf{processedECs}^{(1)} \cup \{[b]\}) < m(\mathsf{processedECs})$. By the LIH, $\mathsf{processedECs}^{(1)} \supseteq \mathsf{processedECs}$. Since $[b] \notin \mathsf{processedECs}$ (otherwise, we would have returned at (1)), we have $\mathsf{processedECs}^{(1)} \cup \{[b]\} \supset \mathsf{processedECs}$, therefore, the measures decreases. To prove that the loop invariants hold at the end of the iteration for the considered case, it is sufficient to apply the LIH and the IH.

Branch (2b.i)
Straightforward application of the LIH.

Branch (2b.ii)
Straightforward application of the LIH.

Returned result (3).
The postconditions are respected by the LI.

□

A.9.6 `GatherPotentiallyAffected`

Proof. Straightforward proof by establishing a loop invariant. We only show that we maintain the invariant at (2a.i) and (2a.ii). Since the loop invariant and the postconditions are similar, the LI guarantees the postconditions of `GatherPotentiallyAffected`.

For (2a.i), $\mathsf{trySubst}'(h')$ satisfies the invariant by the LIH. As such, it suffices to show that $\{\Theta(h)\}$ maintains them as well (which will also prove case (2a.ii)). We note that $h' \notin \mathrm{Im}(T_R)$ by construction of the iterated set. By validity of $\mathscr{K}$, h' must be in $\mathrm{dom}(\Theta)$ (K-INV2 and K-INV12).

Showing that $\Theta(h')$ is an abstract type constructor application is slightly intricate. First, by K-INV12, F must appear in a head position within $\Theta(h')$. Since F is an abstract type constructor and that $\Theta(h')$ satisfies the $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ predicate (by K-INV9 and K-INV6), F must be the unique head in $\Theta(h')$; therefore, $\Theta(h')$ is an abstract type constructor application.

It remains to show that $\Theta(h)$ satisfies the last conjunct of the LIH. By K-INV3, all EC_H appearing in $\Theta(h)$ appear in $\mathscr{K}$, therefore $\varsigma(\Theta(h))$ is defined. Since we have matched $\Theta(h)$ against an abstract type constructor application, it must satisfy the $\mathcal{T}_{EC}\texttt{-IsAbsAppTycon}$ predicate. Finally, by K-INV8 and K-INV6, it is of the same form as $[a]$ and it satisfies the $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}$ predicate (because h is not in T_R).

□

A.10 Termination of the simplification loop (sketch)

The termination condition of `$\mathcal{C}$-Simplify` essentially boils down to ensuring that the compaction phase does not give back constraints we have already accumulated.

Given the subtyping constraint $S \preceq T$ we would like to integrate into $\mathscr{K}$, we observe that `Compact` may only yield new constraints in the following two cases:

1. $([s], [t]) \notin E_{\preceq}$
2. $[s]$ and $[t]$ need to be merged.

These two cases are mutually exclusive: if the edge $([s], [t])$ is not in $E_{\preceq}$, then `TryAddInequality` returns an empty set of ECs to merge; we therefore do not enter the merge loop.

If the number of ECs we create is bounded (with respect to the original constraints C_G), the number of subtyping edges we can add (case (1)) and the number of ECs we can merge (case (2)) are bounded as well. To prove termination, we therefore need to show that the number of ECs we can possibly create is bounded.

Appendix B

Utility functions

B.1 Presumed functions

B.1.1 Operations on union-find data structure

$\mathcal{Q}$-**New** $() : \mathcal{Q}$
- **Output:** A new union-find data structure $\mathcal{Q}$.

$\mathcal{Q}$-**MakeSet** $(\mathcal{Q}) : (\mathcal{Q}', [a])$
- **Input:** The union-find structure $\mathcal{Q}$
- **Output:** The updated $\mathcal{Q}'$ and a fresh $[a]$ that is the representative of the newly created partition.

$\mathcal{Q}$-**Union** $(\mathcal{Q}, [a], [b]) : (\mathcal{Q}', [ab])$
- **Input:** The union-find $\mathcal{Q}$ and two distinct partitions to merge. $[a]$ and $[b]$ must be contained in $\mathcal{Q}$ and be the representatives of their respective partition.
- **Output:** The updated $\mathcal{Q}'$ and an $[ab]$ which is the representative of the merged partition and is either $[a]$ or $[b]$.

$\mathcal{Q}$-**Find** $(\mathcal{Q}, [a]) : [r]$
- **Input:** The union-find $\mathcal{Q}$ and the element for which we would like to find the representative. $[a]$ must be be contained in $\mathcal{Q}$.
- **Output:** The representative of $[a]$.

$\mathcal{Q}$-**AllMembers** $(\mathcal{Q}) : \mathscr{P}(EC_H)$
- **Input:** The union-find $\mathcal{Q}$.
- **Output:** The set of all members in $\mathcal{Q}$.

$\mathcal{Q}$-**MembersOf** $(\mathcal{Q}, [a]) : \mathscr{P}(EC_H)$
- **Input:** The union-find $\mathcal{Q}$. $[a]$ must be contained in $\mathcal{Q}$ and be the representative of its partition.
- **Output:** The set of all members of the partition $[a]$ in $\mathcal{Q}$.

B.1.2 Operations on types

```
$\mathcal{T}$-IsSubtype ($\mathcal{K}$, $T_1 : \mathcal{T}^{cl}$, $T_2 : \mathcal{T}^{cl}$) : $K_3$
    Remark: As indicated by the signature, $T_1$ and $T_2$ are closed, that is:
      $\text{ftv}(T_1) = \text{ftv}(T_2) = \text{ftmv}(T_1) = \text{ftmv}(T_2) = \emptyset$.
    Postcondition: Returns true if $\Gamma \vdash T_1 <: T_2$, false if $\Gamma \nvdash T_1 <: T_2$ and undet otherwise.
$\mathcal{T}$-Fields ($T : \mathcal{T}$) : $\mathcal{V}_X \rightharpoonup \mathcal{T}$
    Output: A partial mapping of the fields contained in $T$ to their type.
DNF ($\mathcal{K}$, $T : \mathcal{T} \cup \mathcal{T}_{EC}$)
    Postcondition: Returns the DNF expansion of $T$. Only the heads need the transformation.
$\mathcal{T}$-IsInhabitedOracle ($T : \mathcal{T}$) : $K_3$
    Postcondition: Returns true if, for all assignments $\phi, \gamma$, there is a $p$ such that $\phi, \gamma \models p : T$,
      false if no such $p$ exist and undet otherwise.
```

B.1.3 Operations on DAG

```
Chain ($G = (V, E), a, b$)
    Precondition: $G$ is a DAG
    Output: A chain $a, x_1, ..., x_n, b$ ($n \geq 0$) such that $(a, x_1), ..., (x_i, x_{i+1}), ..., (x_n, b) \in E$ if it exists,
      and $NIL$ otherwise.
ExistChain ($G = (V, E), a, b$)
    Precondition: $G$ is a DAG
    Default implementation:
    Chain($G, a, b$) $\neq NIL$
ExistUndirChain ($G = (V, E), a, b$)
    Precondition: $G$ is a DAG
    Default implementation:
    ExistChain($G, a, b$) $\vee$ ExistChain($G, b, a$)
ReachableFrom ($G = (V, E), a$)
    Precondition: $G$ is a DAG
    Output: The set of all vertices in $V$ that are reachable from $a$.
LeadingTo ($G = (V, E), a$)
    Precondition: $G$ is a DAG
    Output: The set of all vertices in $V$ that reach $a$.
```

B.2 Auxiliary functions

B.2.1 Shape of types predicates

```
$\mathcal{T}_{EC}$-IsDNF ($\mathscr{K}$, $T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathscr{K}$-WellFormed($\mathscr{K}$) $\wedge$ $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$)
    Output: Return true if $T$ is a non-trivial DNF and false otherwise
    match $T$ :
        case $|_i^n \&_j^{m_i} T_{i,j}$ :
            return true
        otherwise :
            return false
$\mathcal{T}_{EC}$-IsDet ($\mathscr{K}$, $T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathscr{K}$-WellFormed($\mathscr{K}$) $\wedge$ $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$)
    Postcondition: If res = true:
        $\forall \phi, \gamma.\ \phi, \gamma \models \mathscr{K} \implies \mathrm{det}(\phi, \gamma, \varsigma(T))$
    match $T$ :
        case $|_i^n \&_j^{m_i} T_{i,j}$ :
            return $\forall T_{i,j}$. $\mathcal{T}_{EC}$-IsDetSingleHead($T_{i,j}$) $\wedge$
              No (provable) subtyping relationship between all ordered pairs of types in each conjuncts
              $\forall (T_{i,j_1}, T_{i,j_2}) \in \{(T_{i,j_1}, T_{i,j_2}) : 1 \leq i \leq n,\ 1 \leq j_1, j_2 \leq m_i,\ j_1 \neq j_2\}$.
                  $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}$, $T_{i,j_1}$, $T_{i,j_2}$) = false $\wedge$
              No (provable) subtyping relationship between all ordered pairs conjuncts
              $\forall (T_{i_1}, T_{i_2}) \in \{(\&_j^{m_{i_1}} T_{i_1,j}, \&_j^{m_{i_2}} T_{i_2,j}) : 1 \leq i_1, i_2 \leq n,\ i_1 \neq i_2\}$.
                  $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}$, $T_{i_1}$, $T_{i_2}$) = false
        otherwise :
            return $\mathcal{T}_{EC}$-IsDetSingleHead($T$)
$\mathcal{T}_{EC}$-IsDetSingleHead ($T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$) $\wedge$ $\neg\mathcal{T}_{EC}$-IsDNF($T$)
    match $T$ :
        case $T$ where $\mathrm{ftv}(T) = \mathrm{ftmv}(T) = \emptyset$ :
            return true
        $\vec{S}$ can be empty.
        case $Cls[\vec{S}]$ :
            return true
        Note: $[a]$ and $[a][\vec{S}]$ with their HK variants are not considered determined, even if they have a
          determined type. The reason is because we perform an explicit substitution when they become
          determined
        otherwise :
            return false
$\mathcal{T}_{EC}$-IsAbsAppTycon ($T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$)
    match $T$ :
        Note: only matches abstract type constructors (which may be bound in an enclosing HK abstraction).
        case $F[\vec{S}]$ or $[\vec{v}\vec{X} \lhd B]$ =>> $F[\vec{S}]$ :
            return true
        otherwise :
            return false
```

```
$\mathcal{T}_{EC}$-InHead ($\mathcal{Q}, sym, T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$)
    Precondition: $([a] \in sym \implies [a] \in \mathcal{Q}) \land ([a] \in T \implies [a] \in \mathcal{Q})$
    Remark: We employ $\mathcal{Q}$ instead of the whole $\mathcal{K}$ because well-formedness of $\mathcal{K}$ uses this function
    match $T$ :
        case $|_i^n \&_j^{m_i} T_{i,j}$ :
            return $\exists T_{i,j}.$ $\mathcal{T}_{EC}$-InSingleHead($\mathcal{Q}, T_{i,j}$)
        otherwise :
            return $\mathcal{T}_{EC}$-InSingleHead($\mathcal{Q}, T$)

$\mathcal{T}_{EC}$-InSingleHead ($\mathcal{Q}, sym, T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$) $\land \neg$ $\mathcal{T}_{EC}$-IsDNF($T$)
    Precondition: $([a] \in sym \implies [a] \in \mathcal{Q}) \land ([a] \in T \implies [a] \in \mathcal{Q})$
    match $(sym, T)$ :
        case $([a], [b])$ where $\mathcal{Q}$-Find($\mathcal{Q}, [a]$) $=$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
            return true
        case $((p, Q), p.Q)$ :
            return true
        case $((p, F), p.F[\vec{S}])$ :
            return true
        case $(TyCon, TyCon[\vec{S}])$ :
            return true
        Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
        case $(sym, [\vec{v}\vec{X} \lhd B]$ =>> $S)$ :
            return $\mathcal{T}_{EC}$-InSingleHead($\mathcal{Q}, sym, S$)
        otherwise :
            return false
```

B.2.2 Deduction

```
**ApproxDisjunction** $(C_1,\ C_2) : C_3$
    **Precondition:** $C_1$ and $C_2$ are trivial or composed of subtyping constraints
        (i.e., no constraints of the form $p : T$)
    **Postcondition:** $\forall \phi, \gamma.\ \phi, \gamma \models \mathcal{K} \implies$
            $[\phi, \gamma \models C_1 \vee \phi, \gamma \models C_2 \implies \phi, \gamma \models C_3]$
    **if** $C_1 = \texttt{true}$ **then**
        **return** $C_2$
    **else if** $C_2 = \texttt{true}$ **then**
        **return** $C_1$
    *All types appearing in constraints $C_1$ and $C_2$ respectively.*
    $T_1 \leftarrow \bigcup \{\{S, T\} : S \preceq T \in C_1\}$
    $T_2 \leftarrow \bigcup \{\{S, T\} : S \preceq T \in C_2\}$
    *Lower and upper bounds for all types.*
    $L_1 \leftarrow \{(T, \{L : L \preceq T \in C_1\}) : T \in T_1\}$
    $L_2 \leftarrow \{(T, \{L : L \preceq T \in C_2\}) : T \in T_2\}$
    $U_1 \leftarrow \{(T, \{U : T \preceq U \in C_1\}) : T \in T_1\}$
    $U_2 \leftarrow \{(T, \{U : T \preceq U \in C_2\}) : T \in T_2\}$
    $C_3 \leftarrow \texttt{true}$
    **for** $T \in T_1 \cup T_2$ **do**
        **if** $T \in \mathrm{dom}(L_1) \cap \mathrm{dom}(L_2)$ **then**
            $L \leftarrow \bot$
        **else**
            *Note: $L_1(T)$ and $L_2(T)$ cannot be empty, by construction.*
            $L \leftarrow \&(L_1(T) \cup L_2(T))$
        **if** $U \in \mathrm{dom}(U_1) \cap \mathrm{dom}(U_2)$ **then**
            $U \leftarrow \top$
        **else**
            $U \leftarrow |(U_1(T) \cup U_2(T))$
        $C_3 \leftarrow C_3 \curlywedge L \preceq T \curlywedge T \preceq U$
    **return** $C_3$
**DeductionIneqVec** $(\mathcal{K},\ \vec{S} : \mathcal{T}^N,\ \vec{T} : \mathcal{T}^N,\ \vec{v}) : D$
    **Remark:** This function is meant to be unfolded within `DeductionIneq`.
    **Remark:** Default value for $\vec{v}$ is $(+)^N$.
    $D \leftarrow \texttt{true}$
    **for** $i \leftarrow 1$ **to** $|\vec{S}|$ **do**
        **match** $v_i$ :
            **case** $+$ :
                $D \leftarrow D \curlywedge \texttt{DeductionIneq}(\mathcal{K}, S_i, T_i)$
            **case** $-$ :
                $D \leftarrow D \curlywedge \texttt{DeductionIneq}(\mathcal{K}, T_i, S_i)$
            **case** $\pm$ :
                $D \leftarrow D \curlywedge \texttt{DeductionIneq}(\mathcal{K}, S_i, T_i) \curlywedge \texttt{DeductionIneq}(\mathcal{K}, T_i, S_i)$
    **return** $D$
```

```
DeductionIneqDNF ($\mathscr{K}$, $|_i^n \&_j^{m_i} S_{i,j}$, $|_i^o \&_j^{p_i} T_{i,j}$)
    Remark: This function is meant to be unfolded within DeductionIneq.

    $D \leftarrow$ true
    for $i \leftarrow 1$ to $p$ do
        $D' \leftarrow$ 人$_{i'}^{n}$ DeductionIneqConjunct($\mathscr{K}, \&_j^{m_{i'}} S_{i',j}, \&_j^{m_i} T_{i,j}$)
        $D \leftarrow$ ApproxDisjunction($D, D'$)

    return $D$
DeductionIneqConjunct ($\mathscr{K}$, $\&_j^m S_j$, $\&_j^p T_j$)
    Remark: This function is meant to be unfolded within DeductionIneqDNF.

    $D \leftarrow$ true
    for $j \leftarrow 1$ to $m$ do
        $D' \leftarrow$ 人$_{j'}^{p}$ DeductionIneq($\mathscr{K}, S_j, T_{j'}$)
        $D \leftarrow$ ApproxDisjunction($D, D'$)

    return $D$
```

B.2.3 Operations on $\mathscr{K}$

```
$\mathscr{K}$-New () : $\mathscr{K}$
    Postcondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}\texttt{)}$
    Postcondition: $\mathscr{K}\texttt{-to-}\mathcal{C}\texttt{(}\mathscr{K}\texttt{)} \equiv \texttt{true}$
    return $[\mathcal{M} \mapsto \emptyset,\ \Theta \mapsto \emptyset,\ \mathcal{R} \mapsto \emptyset,\ \mathcal{D} \mapsto \emptyset,\ \mathcal{Q} \mapsto \mathcal{Q}\texttt{-New()},\ \mathcal{I} \mapsto \emptyset,$
        $T_R \mapsto \emptyset,\ G_{\preceq} \mapsto (\emptyset,\emptyset),\ G_{EC} \mapsto (\emptyset,\emptyset,\emptyset,\emptyset),\ G_{\mathcal{S}} \mapsto (\emptyset,\emptyset,\emptyset),\ G_p \mapsto (\emptyset,\emptyset,\emptyset)]$
$\mathcal{T}_{EC}$-kind $(\mathscr{K}, T : \mathcal{T}_{EC}) : \kappa$
    Precondition: $\mathscr{K}$ satisfies K-INV1, K-INV2, K-INV5 and K-INV7
    Precondition: $[a] \in T \implies [a] \in \mathscr{K}$
    match $T$ :
        case $[a]$ :
            $[r] \leftarrow \mathcal{Q}\texttt{-Find(}\mathcal{Q}, [a]\texttt{)}$
            return $\mathrm{kind}(\Theta(T_R([r])))$
        case $[a]\,[\vec{A}]$ :
            return $\star$
        otherwise :
            We assume that the $EC_H$ composing $T$ are not an issue to obtain the kind.
            return $\mathrm{kind}(T)$
UpdateMember $(\mathscr{K}, h, S : \mathcal{T}_{EC}) : \mathscr{K}'$
    Description: Replace the underlying type of $h$ with $S$. $S$ must have the same determinacy
        as the current underlying type referenced by $h$
    Precondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}\texttt{)} \wedge h \notin \mathrm{Im}(T_R) \wedge \varsigma(S)\!\downarrow \wedge\ \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv(}S\texttt{)} \wedge$
        $\mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, S\texttt{)} = \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, \mathcal{R}(h)\texttt{)} \wedge h \in \mathrm{dom}(\Theta) \wedge$
        $\mathcal{T}_{EC}\texttt{-IsDet(}\mathscr{K}, \Theta(h)\texttt{)} \iff \mathcal{T}_{EC}\texttt{-IsDet(}\mathscr{K}, S\texttt{)}$
    Postcondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}'\texttt{)} \wedge \mathcal{Q} = \mathcal{Q}' \wedge T_R = T_R' \wedge G_{\preceq} = G_{\preceq}'$
    Postcondition: $\mathrm{dom}(\Theta') = \mathrm{dom}(\Theta)\ \wedge$
        $\forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus \{h\}.\ \Theta(\tilde{h}) = \Theta'(\tilde{h})$
    Postcondition: $\mathscr{K} \curlywedge \varsigma(\Theta(h)) \asymp \varsigma(S) \Vdash \mathscr{K}'$
    $\Theta' \leftarrow \Theta[h \mapsto S]$
    We need to update $G_{EC}$, $G_{\mathcal{S}}$ and $G_p$ as well (similarly to $\mathcal{T}_{EC}$-CreateEC)
    $(\mathsf{syms}, \mathsf{ecsH}, \mathsf{ecsNH}, \mathsf{pathDep}) \leftarrow \mathcal{T}_{EC}\texttt{-Composition(}S, \emptyset\texttt{)}$
    if $\mathsf{syms} \neq \emptyset$ then
        $U_{\mathcal{S}}' \leftarrow U_{\mathcal{S}} \cup \mathsf{syms}$
        We first remove all “old” appearances of $h$ before adding the new ones
        $E_{\mathcal{S}}' \leftarrow (E_{\mathcal{S}} \setminus \{(sym, h) : (sym, h) \in E_{\mathcal{S}}\}) \cup \{(sym, h) : sym \in \mathsf{syms}\}$
        $h$ is already in $V_{\mathcal{S}}$
        $G_{\mathcal{S}}' \leftarrow (U_{\mathcal{S}}' V_{\mathcal{S}}, E_{\mathcal{S}}')$
    else
        $G_{\mathcal{S}}' \leftarrow G_{\mathcal{S}}$
    if $\mathsf{ecsH} \cup \mathsf{ecsNH} \neq \emptyset$ then
        $U_{EC}' \leftarrow U_{EC} \cup \mathsf{ecsH} \cup \mathsf{ecsNH}$
        $E_{EC}' \leftarrow (E_{EC} \setminus \{([a], h) : ([a], h) \in E_{EC}\}) \cup \{([a], h) : [a] \in \mathsf{ecsH} \cup \mathsf{ecsNH}\}$
        $L_{EC}' \leftarrow \big(L_{EC} \restriction (\mathrm{dom}(L_{EC}) \setminus \{([a], h) : ([a], h) \in \mathrm{dom}(L_{EC})\})\big)$
                $\cup \{(([a], h), H) : [a] \in \mathsf{ecsH}\} \cup \{(([a], h), NH) : [a] \in \mathsf{ecsNH}\}$
        $G_{EC}' \leftarrow (U_{EC}', V_{EC}, E_{EC}', L_{EC}')$
    else
        $G_{EC}' \leftarrow G_{EC}$
    if $\mathsf{pathDep} \neq \emptyset$ then
        $U_p' \leftarrow U_p \cup \mathsf{pathDep}$
        $E_p' \leftarrow (E_p \setminus \{((p, sym), h) : ((p, sym), h) \in E_p\}) \cup \{((p, sym), h) : (p, sym) \in \mathsf{pathDep}\}$
        $G_p' \leftarrow (U_p', V_p, E_p')$
    else
        $G_p' \leftarrow G_p$
    return $\mathscr{K}[\Theta \mapsto \Theta', G_{\mathcal{S}} \mapsto G_{\mathcal{S}}', G_{EC} \mapsto G_{EC}', G_p \mapsto G_p']$
```

```
UpdateMemberDetermined $(\mathscr{K}, h, S : \mathcal{T}_{EC}) : (\mathscr{K}', \mathsf{cstrts}, \mathsf{toMerge}, \mathsf{dets})$
    Description: Replace the underlying type of $h$ with $S$. $S$ must be determined.
    Precondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}\texttt{)} \land \mathcal{T}_{EC}\texttt{-IsDet(}\mathscr{K}, S\texttt{)} \land h \in \mathrm{dom}(\Theta) \land h \notin \mathrm{Im}(T_R) \land \varsigma(S){\downarrow} \land$
        $\mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv(}S\texttt{)} \land \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, S\texttt{)} = \mathcal{T}_{EC}\texttt{-kind(}\mathscr{K}, \mathcal{R}(h)\texttt{)}$
    Postcondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}'\texttt{)} \land \varsigma = \varsigma' \land \mathcal{Q} = \mathcal{Q}' \land T_R = T_R' \land G_{\preceq} = G_{\preceq}'$
    Postcondition: $\mathrm{dom}(\Theta') \cup \{h\} = \mathrm{dom}(\Theta)$
        $\forall \tilde{h} \in \mathrm{dom}(\Theta) \setminus \{h\}.\ \Theta(\tilde{h}) = \Theta'(\tilde{h})$
    Postcondition: $\mathrm{dom}(\mathcal{D}) \uplus \mathsf{dets} \subseteq \mathrm{dom}(\mathcal{D}')$
    Postcondition: $\mathscr{K} \curlywedge \varsigma(\Theta(h)) \asymp \varsigma(S) \Vdash \mathscr{K}' \curlywedge \bigcurlywedge \mathsf{cstrts} \curlywedge M(\mathscr{K}', \mathsf{toMerge})$

    if $\mathcal{R}(h) \in \mathrm{dom}(\mathcal{D})$ then
        $\varsigma \leftarrow EC_H\texttt{-Subst(}\mathscr{K}\texttt{)}$
        $\mathsf{cstrts} \leftarrow \{\varsigma(S) \asymp \varsigma(\mathcal{D}(\mathcal{R}(h)))\}$
        $\mathscr{K} \leftarrow \texttt{RemoveMember(}\mathscr{K}, h\texttt{)}$
        return $(\mathscr{K}, \varsigma(S) \asymp \varsigma(\mathcal{D}(\mathcal{R}(h))), \emptyset, \emptyset)$
    else
        $(\_, [x]) \leftarrow \mathcal{T}_{EC}\texttt{-FindOrCreateEC(}\mathscr{K}, S, \emptyset, \emptyset, \texttt{true}, \texttt{false)}$
        if $[x] \neq NIL \land [x] \neq \mathcal{R}(h)$ then
            $\mathscr{K} \leftarrow \texttt{RemoveMember(}\mathscr{K}, h\texttt{)}$
            return $(\mathscr{K}, \emptyset, \big\{\{\mathcal{R}(h), [x]\}\big\}, \emptyset)$
        else
            $\mathcal{D} \leftarrow \mathcal{D}\big[\mathcal{R}(h) \mapsto h\big]$
            $\mathscr{K} \leftarrow \texttt{UpdateMember(}\mathscr{K}, h, S\texttt{)}$
            return $(\mathscr{K}, \emptyset, \emptyset, \{[x]\})$

RemoveMember $(\mathscr{K}, h) : \mathscr{K}'$
    Precondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}\texttt{)} \land h \notin \mathrm{Im}(T_R) \land h \in \mathrm{dom}(\Theta)$
    Postcondition: $\mathscr{K}\texttt{-Valid(}\mathscr{K}'\texttt{)} \land \varsigma = \varsigma' \land \mathcal{Q} = \mathcal{Q}' \land T_R = T_R' \land G_{\preceq} = G_{\preceq}' \land$
        $\big(\mathcal{D}(\mathcal{R}(h)) = h \implies \mathcal{R}(h) \notin \mathrm{dom}(\mathcal{D}')\big)$
    Postcondition: $\mathscr{K} \Vdash \mathscr{K}'$

    $\mathcal{M}' \leftarrow \mathcal{M}[\mathcal{R}(h) \mapsto \mathcal{M}(\mathcal{R}(h)) \setminus \{h\}]$
    $\mathcal{D}' \leftarrow \mathcal{D}[\mathcal{R}(h) \mapsto \uparrow]$
    $\Theta' \leftarrow \Theta[h \mapsto \uparrow]$
    $\mathcal{R}' \leftarrow \mathcal{R}[h \mapsto \uparrow]$

    $E_{\mathcal{S}}' \leftarrow E_{\mathcal{S}} \setminus \{(sym, h) : (sym, h) \in E_{\mathcal{S}}\}$
    $G_{\mathcal{S}}' \leftarrow (U_{\mathcal{S}} V_{\mathcal{S}} \setminus \{h\}, E_{\mathcal{S}}')$

    $E_{EC}' \leftarrow E_{EC} \setminus \{([a], h) : ([a], h) \in E_{EC}\}$
    $L_{EC}' \leftarrow \big(L_{EC} \upharpoonright (\mathrm{dom}(L_{EC}) \setminus \{([a], h) : ([a], h) \in \mathrm{dom}(L_{EC})\})\big)$
    $G_{EC}' \leftarrow (U_{EC}, V_{EC} \setminus \{h\}, E_{EC}', L_{EC}')$

    $E_p' \leftarrow E_p \setminus \{((p, sym), h) : ((p, sym), h) \in E_p\}$
    $G_p' \leftarrow (U_p, V_p \setminus \{h\}, E_p')$

    return $\mathscr{K}[\mathcal{M} \mapsto \mathcal{M}', \Theta \mapsto \Theta', \mathcal{R} \mapsto \mathcal{R}', \mathcal{D} \mapsto \mathcal{D}', G_{\mathcal{S}} \mapsto G_{\mathcal{S}}', G_{EC} \mapsto G_{EC}', G_p \mapsto G_p']$
```

B.2.4 Operations on types

```
$\mathcal{T}_{EC}$-ApplyHeadSubstitution ($\mathscr{K}, T : \mathcal{T}_{EC}, [a], S : \mathcal{T}_{EC}$) : $T' : \mathcal{T}_{EC}$
    Description: In $T$, replace all head occurrences of $[a]$ with $S$.
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$)
    Precondition: $\varsigma([a])\downarrow \wedge \varsigma(S)\downarrow \wedge \varsigma(T)\downarrow \wedge \mathcal{T}_{EC}$-kind($\mathscr{K}, S$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}, [a]$) $\wedge$
        $\mathcal{T}_{EC}$-in-$\Theta$-Inv($T$) $\wedge$ $\mathcal{T}_{EC}$-in-$\Theta$-Inv($S$)
    Postcondition: $\varsigma(T')\downarrow \wedge \varsigma([a]) \asymp \varsigma(S) \Vdash \varsigma(T) \asymp \varsigma(T')$
    Postcondition: $\mathcal{T}_{EC}$-kind($\mathscr{K}, T$) $=$ $\mathcal{T}_{EC}$-kind($\mathscr{K}, T'$)
    Postcondition: $\mathcal{T}_{EC}$-IsDet($\mathscr{K}, T$) $\implies T = T'$

    match $T$ :
        case $[b]$ where $\mathcal{Q}$-Find($\mathcal{Q}, [a]$) $=$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
            return $S$
        case $[b]\lceil\vec{A}\rceil$ where $\mathcal{Q}$-Find($\mathcal{Q}, [a]$) $=$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
            We can deconstruct $S$ because it has the same kind as $[a]$ and $[b]$ whose application is well-formed.
            ($[\vec{v}\vec{X} \lhd \_]$ =>> $U$) $\leftarrow S$
            return $[\vec{X} \mapsto \vec{A}]U$
        case $|_i^n \&_j^{m_i} T_{i,j}$
            $T'_{i,j} \leftarrow NIL$   for $1 \leq i \leq n, 1 \leq j \leq m_i$
            for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $m_i$ do
                match $T_{i,j}$ :
                    case $[b]$ where $\mathcal{Q}$-Find($\mathcal{Q}, [a]$) $=$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
                        $T'_{i,j} \leftarrow S$
                    case $[b]\lceil\vec{A}\rceil$ where $\mathcal{Q}$-Find($\mathcal{Q}, [a]$) $=$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
                        ($[\vec{v}\vec{X} \lhd \_]$ =>> $U$) $\leftarrow S$
                        return $[\vec{X} \mapsto \vec{A}]U$
                    otherwise :
                        $T'_{i,j} \leftarrow T_{i,j}$
            If $S$ is a DNF, $T'$ loses its DNF form, so we make sure to avoid that by applying a DNF
             transformation.
            return DNF($|_i^n \&_j^{m_i} T'_{i,j}$)
        Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
        case $[\vec{v}\vec{X} \lhd B]$ =>> $U$ :
            $U' \leftarrow \mathcal{T}_{EC}$-ApplyHeadSubstitution($\mathscr{K}, U, [a], S$)
            return $[\vec{v}\vec{X} \lhd B]$ =>> $U'$
        otherwise :
            return $T$
```

$\mathcal{T}_{EC}$-TryApplyHeadSubstitution $(\mathscr{K}, S : \mathcal{T}_{EC}, U : \mathcal{T}_{EC}, V : \mathcal{T}_{EC}) : S' : \mathcal{T}_{EC} \uplus \{NIL\}$

Description: In S, try to replace U to V. U and V must be of same kind. The head of S and T must be an applied abstract type constructor.

Remark: While U and V must have the same kind, S and U (or V) may have different kind.

Example: With $S = U = F[\texttt{String}]$ and $V = \texttt{Foo}$, we get $S' = V = \texttt{Foo}$.

Example: With $S = [X, Y] \texttt{=>>} F[Y, \texttt{Int}]$, $U = [Z] \texttt{=>>} F[Z, \texttt{Int}]$, and $V = [Z] \texttt{=>>} \texttt{List}[Z]$, we get $S' = [X, Y] \texttt{=>>} \texttt{List}[Y]$.

Example: With $S = [X] \texttt{=>>} F[X]$, $U = [Z] \texttt{=>>} F[\texttt{Inv}[Z]]$, and any V, we get $S' = NIL$.

Example: With $S = F[\texttt{Option}[\texttt{Int}]]$, $U = [Z <: \texttt{Int} \mid \texttt{String}] \texttt{=>>} F[\texttt{Option}[Z]]$, and $V = [Z <: \texttt{Int} \mid \texttt{String}] \texttt{=>>} \texttt{List}[Z]$, we get $S' = \texttt{List}[\texttt{Int}]$.

Precondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}) \land \varsigma(S)\downarrow, \varsigma(U)\downarrow, \varsigma(V)\downarrow$

Precondition: $\mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(S) \land \mathcal{T}_{EC}\texttt{-IsAbsAppTycon}(U) \land \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, U) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, V) \land \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S) \land \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(U) \land \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(V)$

Postcondition: $S' \neq NIL \implies \varsigma(S')\downarrow \land \mathscr{K} \curlywedge \varsigma(U) \asymp \varsigma(V) \Vdash \varsigma(S) \asymp \varsigma(S')$

Postcondition: $S' \neq NIL \implies \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, S) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, S') \land \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(S')$

Postcondition: $(S' \neq NIL \land \mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}, V)) \implies \mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}, S')$

```
case $(S, U)$ :
    case $(F[\vec{A}_1], F[\vec{A}_2])$ :
        if $\mathcal{T}_{EC}\texttt{-EquivVec}(\mathscr{K}, F[\vec{A}_1], F[\vec{A}_2])$ then
            return $V$
        else
            return $NIL$
    Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ and $\bar{Y}$ fresh.
    case $([\vec{v}_X \vec{X} \lhd B] \texttt{=>>} F[\vec{A}_1],\ [\vec{v}_Y \vec{Y} \lhd \_] \texttt{=>>} F[\vec{A}_2])$ :
        $\sigma \leftarrow \mathcal{T}_{EC}\texttt{-TryMatch}(\mathscr{K}, \bar{Y}, F[\vec{A}_2], F[\vec{A}_1])$
        if $\sigma \neq NIL$ then
            We can deconstruct $V$ because it has the same kind as $U$.
            $([\vec{v}_Y \vec{Y} \lhd \_] \texttt{=>>} \tilde{V}) \leftarrow V$
            return $[\vec{v}_X \vec{X} \lhd B] \texttt{=>>} \sigma(\tilde{V})$
        else
            return $NIL$
```

```
    Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
    case $(F[\vec{A}_1],\ [\vec{v}_Y \vec{Y} \triangleleft B_U]$ =>> $F[\vec{A}_2])$ where $\mathcal{T}_{EC}$-kind($\mathcal{K}, \vec{A}_1$) $=$ kind$(\vec{Y})$ :
        $\sigma \leftarrow \mathcal{T}_{EC}$-TryMatch($\mathcal{K}, \bar{Y}, F[\vec{A}_2], F[\vec{A}_1]$)
        Extending $\sigma$ with $\top$ for $Y$'s not appearing in $F[\vec{A}_2]$
        $\sigma' \leftarrow \sigma[Y \mapsto \top_{\mathrm{kind}(Y)}, Y \in \bar{Y} \setminus \mathrm{dom}(\sigma)]$
        Destructuring $\sigma'$.
        $[\vec{Y} \mapsto \vec{A}'] \leftarrow \sigma'$
        if $\sigma \neq NIL$ then
            We can deconstruct $V$ because it has the same kind as $U$.
            $([\vec{v}_Y \vec{Y} \triangleleft B_V]$ =>> $\tilde{V}) \leftarrow V$
            $\vec{A}'$ must satisfy the bounds $B_V$ if we want it to be applied to $V$.
            Since we claim $\mathcal{K} \curlywedge \varsigma(U) \asymp \varsigma(V) \Vdash \varsigma(S) \asymp \varsigma(S')$, we can assume that $U$ and $V$ are equivalent
              under $\mathcal{K}$. As such, we can also check if $\vec{A}'$ satisfy $B_U$ if we cannot prove it satisfies $B_V$.
            if $\mathcal{B}_{EC}$-Satisified($\mathcal{K}, B_V, [\vec{Y} \mapsto \vec{A}']$) $\vee$ $\mathcal{B}_{EC}$-Satisified($\mathcal{K}, B_U, [\vec{Y} \mapsto \vec{A}']$) then
                return $[\vec{Y} \mapsto \vec{A}']\tilde{V}$
            else
                return $NIL$
        else
            return $NIL$
    otherwise :
        return $NIL$
$\mathcal{T}$-InhabitedTypes $(p : \mathcal{P},\ T : \mathcal{T}) : \mathcal{P} \rightharpoonup \mathcal{T}$
    Description: Recursively retrieve all types that are inhabited by a field in $T$.
    Postcondition: $\forall (q, S) \in \mathsf{res}.\ p : T \Vdash q : S$
    toVisit $\leftarrow \{(p, T)\}$
    visited $\leftarrow \emptyset$
    while $\exists (q, S) \in$ toVisit do
        visited $\leftarrow$ visited $\cup \{(q, S)\}$
        fields $\leftarrow \mathcal{T}$-Fields($S$)
        Remove fields whose type has already been visited.
        fields $\leftarrow$ fields $\restriction (\mathrm{dom}(\mathsf{fields}) \setminus \mathrm{Im}(\mathsf{visited}))$
        toVisit $\leftarrow$ (toVisit $\setminus \{(q, S)\}) \cup \{(q.a, U) : (a, U) \in$ fields$\}$
    return visited
```

B.2.5 DNF related

```
$\mathcal{T}$-CommonTypes $(|_i^n \&_j^{m_i} T_{i,j} : \mathcal{T}) : \mathscr{P}(\mathcal{T})$
  return $\bigcap \{\{T_{i,j} : 1 \leq j \leq m_i\} : 1 \leq i \leq n\}$
$\mathcal{T}_{EC}$-SimplifyDNF $(\mathscr{K},\ T : \mathcal{T}_{EC}) : T' : \mathcal{T}_{EC}$
  Remark: Also accepts trivial DNFs and (possibly trivial) DNFs in HK abstraction as well.
  Precondition: $\mathscr{K}\texttt{-Valid}(\mathscr{K}) \wedge \varsigma(T)\!\downarrow \wedge \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T)$
  Postcondition: $\varsigma(T')\!\downarrow\ \wedge\ \mathscr{K} \Vdash \varsigma(T) \asymp \varsigma(T')$
  Postcondition: $\mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, T) = \mathcal{T}_{EC}\texttt{-kind}(\mathscr{K}, T')$
  Postcondition: $T' \notin EC_H \implies \mathcal{T}_{EC}\texttt{-in-}\Theta\texttt{-Inv}(T')$
  Postcondition: $\mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}, T) \implies \mathcal{T}_{EC}\texttt{-IsDet}(\mathscr{K}, T')$

  match $T$ :
    case $|_i^n \&_j^{m_i} T_{i,j}$ :
      We first try to simplify the inner conjuncts. We represent the DNF with a set, as it is easier to work with.
      $\bar{T}$ is used as an intermediate result to store the DNF with simplified conjuncts.
      $\bar{T} \leftarrow \emptyset$
      for $i \leftarrow 1$ to $n$ do
        $\bar{T}_i \leftarrow \emptyset$
        for $j_1 \leftarrow 1$ to $m_i$, $j_2 \leftarrow 1$ to $j_1$ do
          if $\mathcal{T}_{EC}\texttt{-Equiv}(\mathscr{K}, T_{i,j_1}, T_{i,j_2})$ then
            $\bar{T}_i \leftarrow \bar{T}_i \cup \{T_{i,j_1}\}$
          else
            $\bar{T}_i \leftarrow \bar{T}_i \cup \{T_{i,j_1}, T_{i,j_2}\}$
        $\bar{T} \leftarrow \bar{T} \cup \bar{T}_i$
      $\bar{T}' \leftarrow \emptyset$
      Now we attempt to simplify the disjunctions.
      for $\{\bar{T}_i, \bar{T}_j\} \in \binom{\bar{T}}{2}$ do
        Note: $\bar{T}_i$ and $\bar{T}_i$ have each at least one element. A singleton results in a trivial conjunction.
        if $\mathcal{T}_{EC}\texttt{-Equiv}(\mathscr{K}, \&\bar{T}_i, \&\bar{T}_j)$ then
          $\bar{T}' \leftarrow \bar{T}' \cup \{\bar{T}_i\}$
        else
          $\bar{T}' \leftarrow \bar{T}' \cup \{\bar{T}_i, \bar{T}_j\}$
      Note: may result in a trivial DNF (which is acceptable).
      return $|\&\bar{T}'$
    Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
    case $[\vec{v}\vec{X} \lhd B] \texttt{=>>} \tilde{T}$ :
      $\tilde{T}' \leftarrow \mathcal{T}_{EC}\texttt{-SimplifyDNF}(\mathscr{K}, \tilde{T})$
      return $[\vec{v}\vec{X} \lhd B] \texttt{=>>} \tilde{T}'$
    otherwise :
      return $T$
```

B.2.6 Equivalency of types and bounds

```
$\mathcal{T}_{EC}$-Equiv ($\mathcal{K}$, $S : \mathcal{T}_{EC}$, $T : \mathcal{T}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathcal{K}$-Valid($\mathcal{K}$) $\wedge\ \varsigma(S)\downarrow \wedge\ \varsigma(T)\downarrow$
    Postcondition: res = true $\implies \mathcal{K} \Vdash \varsigma(S) \asymp \varsigma(T)$
    match $(S, T)$ :
        case $(X, X)$ :
            return true
        case $([a], [b])$ where $\mathcal{Q}$-Find($\mathcal{Q}$, $[a]$) = $\mathcal{Q}$-Find($\mathcal{Q}$, $[b]$):
            return true
        case $([a]\,[\vec{U}], [b]\,[\vec{V}])$ where $\mathcal{Q}$-Find($\mathcal{Q}$, $[a]$) = $\mathcal{Q}$-Find($\mathcal{Q}$, $[b]$):
            return $\mathcal{T}_{EC}$-EquivVec($\mathcal{K}$, $\vec{U}$, $\vec{V}$)
        case $(TyCon[\vec{U}], TyCon[\vec{V}])$ where $\mathcal{Q}$-Find($\mathcal{Q}$, $[a]$) = $\mathcal{Q}$-Find($\mathcal{Q}$, $[b]$):
            return $\mathcal{T}_{EC}$-EquivVec($\mathcal{K}$, $\vec{U}$, $\vec{V}$)
        Note: assumes implicit $\alpha$-renaming to have $\bar{Y}$ fresh.
        case $([\vec{v}\vec{Y} \lhd B_1]$ =>> $\tilde{S}$, $[\vec{v}\vec{Y} \lhd B_2]$ =>> $\tilde{T})$ :
            return $\mathcal{B}_{EC}$-Equiv($\mathcal{K}$, $B_1$, $B_2$) $\wedge$ $\mathcal{T}_{EC}$-Equiv($\mathcal{K}$, $\tilde{S}$, $\tilde{T}$)
        Note: matches the $i$ and $m_j$ as well. Furthemore, we assume that the $m_j$ are sorted in an ascending
          order; that is, $m_1 \leq m_2, ..., \leq m_n$.
        case $(|_i^n \&_j^{m_i} T_{i,j},\ |_i^n \&_j^{m_i} S_{i,j})$ :
            The idea is to go over all set of conjunctions having the same number of terms and ensure that all
              conjuncts in $T$ have an equivalent conjunct in $S$ and vice-versa.
            for $\bar{i} \in \{\{i' : m_i = m_{i'},\ 1 \leq i' \leq n\} : 1 \leq i \leq n\}$ do
                Indices of conjuncts in $T$ and $S$ that have an equivalent conjunct in the other type.
                $\bar{T}_i, \bar{S}_i \leftarrow \emptyset$
                for $(i_1, i_2) \in \bar{i} \times \bar{i}$ do
                    We remind that $m_{i_1} = m_{i_2}$
                    if $\mathcal{T}_{EC}$-EquivConjunct($\mathcal{K}$, $\&_j^{m_{i_1}} T_{i_1,j}$, $\&_j^{m_{i_2}} S_{i_2,j}$) then
                        $\bar{T}_i \leftarrow \bar{T}_i \cup \{i_1\}$
                        $\bar{S}_i \leftarrow \bar{S}_i \cup \{i_2\}$
                If some conjuncts were not "matched", we cannot prove equivalency.
                if $\bar{T}_i \neq \bar{i} \vee \bar{S}_i \neq \bar{i}$ then
                    return false
            return true
        otherwise :
            return false

$\mathcal{T}_{EC}$-EquivVec ($\mathcal{K}$, $\vec{S} : \mathcal{T}_{EC}^N$, $\vec{T} : \mathcal{T}_{EC}^N$) : $\mathbb{B}$
    return $\forall i.\ 1 \leq i \leq |\vec{S}| \implies$ $\mathcal{T}_{EC}$-Equiv($\mathcal{K}$, $S_i$, $T_i$)

$\mathcal{T}_{EC}$-EquivConjunct ($\mathcal{K}$, $\&_j^m T_j : \mathcal{T}_{EC}$, $\&_j^m S_j : \mathcal{T}_{EC}$) : $\mathbb{B}$
    We do something similar as in $\mathcal{T}_{EC}$-Equiv.
    $\bar{T}_j, \bar{S}_j \leftarrow \emptyset$
    for $j_1, j_2 \leftarrow 1$ to $m$ do
        if $\mathcal{T}_{EC}$-Equiv($\mathcal{K}$, $T_{j_1}$, $S_{j_2}$) then
            $\bar{T}_j \leftarrow \bar{T}_j \cup \{j_1\}$
            $\bar{S}_j \leftarrow \bar{S}_j \cup \{j_2\}$
    return $\bar{T}_j = \bar{S}_j = \bar{i}$
```

```
$\mathcal{B}_{EC}$-Equiv ($\mathscr{K}$, $B_1 : \mathcal{B}_{EC}$, $B_2 : \mathcal{B}_{EC}$) : $\mathbb{B}$
    Precondition: $\mathscr{K}$-Valid($\mathscr{K}$) $\wedge \varsigma(B_1)\downarrow \wedge \varsigma(B_2)\downarrow \wedge \mathrm{dom}(B_1) = \mathrm{dom}(B_2) \wedge$
        $\mathrm{dom}(B_1) \# \mathrm{ftv}(\mathscr{K})$
    Postcondition: If res = true:
      $\forall \phi, \gamma.\ \phi, \gamma \models \mathscr{K} \implies \forall \vec{T} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{T}], \gamma \models \varsigma(B_1) \iff \phi[\vec{X} \mapsto \vec{T}], \gamma \models \varsigma(B_2)$
      where $\vec{X} = \mathrm{dom}(B_1)$. The quantified $\vec{T}$ has the same length and kind as $\vec{X}$.

    return $\forall X \in \mathrm{dom}(B_1)$.
        $\mathcal{T}_{EC}$-Equiv($\mathscr{K}, \pi_1(B_1(X)), \pi_1(B_2(X))$) $\wedge$
        $\mathcal{T}_{EC}$-Equiv($\mathscr{K}, \pi_2(B_1(X)), \pi_2(B_2(X))$)
```

B.2.7 Constraints satisfaction

```
$\mathcal{B}_{EC}$-Subsumes ($\mathcal{K}$, $B_1 : \mathcal{B}_{EC}$, $B_2 : \mathcal{B}_{EC}$) : $K_3$
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $\mathrm{dom}(B_1) = \mathrm{dom}(B_2) \wedge \mathrm{dom}(B_1) \# \mathrm{ftv}(\mathcal{K})$
    Postcondition: If res = true:
      $\forall \phi, \gamma.\ \phi, \gamma \models \mathcal{K} \implies \left[\forall \vec{T} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{T}], \gamma \models B_1 \implies \phi[\vec{X} \mapsto \vec{T}], \gamma \models B_2\right]$
        where $\vec{X} = \mathrm{dom}(B_1)$. The quantified $\vec{T}$ has the same length and kind as $\vec{X}$.
    Postcondition: If res = false:
      $\forall \phi, \gamma.\ \phi, \gamma \models \mathcal{K} \implies \left[\exists \vec{T} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{T}], \gamma \models B_1 \wedge \phi[\vec{X} \mapsto \vec{T}], \gamma \not\models B_2\right]$
    return $\forall X \in \mathrm{dom}(B_1)$.
        $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, \pi_1(B_2(X)), \pi_1(B_1(X))$) $\wedge$
        $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, \pi_2(B_1(X)), \pi_2(B_2(X))$)
$\mathcal{B}_{EC}$-BoundsEntailed ($\mathcal{K}$, $B : \mathcal{B}_{EC}$) : $K_3$
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $\mathrm{dom}(B) \# \mathrm{ftv}(\mathcal{K})$
    Postcondition: If res = true:
      $\forall \phi, \gamma.\ \phi, \gamma \models \mathcal{K} \implies \forall \vec{T} \in (\mathcal{T}^{\mathrm{cl}})^{|\vec{X}|}.\ \phi[\vec{X} \mapsto \vec{T}], \gamma \models B$
        where $\vec{X} = \mathrm{dom}(B)$. The quantified $\vec{T}$ has the same length and kind as $\vec{X}$.
    Postcondition: If res = false:
      $\forall \phi, \gamma.\ \phi, \gamma \models \mathcal{K} \implies \exists \vec{T} \in (\mathcal{T}^{\mathrm{cl}})^{N}.\ \phi[\vec{X} \mapsto \vec{T}], \gamma \not\models B$
    return $\forall X \in \mathrm{dom}(B_1)$.
        $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, \pi_1(B(X)), \pi_2(B(X))$)
$\mathcal{B}_{EC}$-Satisified ($\mathcal{K}$, $B : \mathcal{B}_{EC}, [\vec{X} \mapsto \vec{A}]$)
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$) $\wedge\ \varsigma(B)\!\downarrow \wedge\ \varsigma(\vec{A})\!\downarrow \wedge\ \bar{X} = \mathrm{dom}(B)$
    Postcondition: res = true $\implies \mathcal{K} \Vdash \varsigma(\vec{A}) \triangleleft \varsigma(B)$
    return $\forall (X_i, (L_i, U_i)) \in B$.
        $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, [\vec{X} \mapsto \vec{A}]L_i, A_i$) $\wedge$
        $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, A_i, [\vec{X} \mapsto \vec{A}]U_i$)
$\mathcal{T}_{EC}$-IsSubtype ($\mathcal{K}$, $T_1 : \mathcal{T}_{EC}$, $T_2 : \mathcal{T}_{EC}$) : $K_3$
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Postcondition: Returns true if $\mathcal{K} \Vdash T_1 \preceq T_2$
      false if, for all $\phi, \gamma$ satisfying $\mathcal{K}$, $\phi, \gamma \not\models T_1 \preceq T_2$ and undet otherwise.
    match $T_1 \preceq T_2$ :
        case $T_1 \preceq T_2$ where $T_1, T_2 \in \mathcal{T}^{\mathrm{cl}}$ :
            return $\mathcal{T}$-IsSubtype($T_1, T_2$)
        case $T \preceq T$ :
            return true
        case $T_1 \preceq \top$ :
            return true
        case $\bot \preceq T_2$ :
            return true
```

```
        case $Cls_1[\vec{S}_1] \preceq Cls_2[\vec{S}_2]$ :
            if $Cls_1$ does not extend $Cls_2$ then
                return false
            else if $Cls_1 = Cls_2$ then
                With $\vec{v}$ the variance signs of $Cls_1$
                return $\mathcal{T}_{EC}$-IsSubtypeVec($\mathscr{K}, \vec{S}_1, \vec{S}_2, \vec{v}$)
            else
                Then, $Cls_1$ extends $Cls_2$ $N \geq 1$ times through $\boldsymbol{\sigma}_1, ..., \boldsymbol{\sigma}_N$ such that:
                $Cls_1[\vec{S}] \preceq \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})] \preceq Cls_2[\vec{S}_2]$.
                return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, \&_i^N Cls_2[\boldsymbol{\sigma}_i(\vec{S})], Cls_2[\vec{S}_2]$)
        Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
        case $[\vec{v}\vec{X} \preceq B_1]$ =>> $S_1 \preceq [\vec{v}\vec{X} \preceq B_2]$ =>> $S_2$ :
            return $\mathcal{B}_{EC}$-Subsumes($\mathscr{K}, B_2, B_1$) $\wedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, S_1, S_2$)
        For simplicity, we assume it is possible to deconstruct a DNF as follows.
        case $T_1 \preceq U \mathbin{\&} V$ :
            return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, T_1, U$) $\wedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, T_1, V$)
        case $U \mid V \preceq T_2$ :
            return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, U, T_2$) $\wedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, V, T_2$)
        case $T_1 \preceq U \mid V$ :
            return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, T_1, U$) $\vee$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, T_1, V$)
        case $U \mathbin{\&} V \preceq T_2$ :
            return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, U, T_2$) $\vee$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, V, T_2$)
        case $[a] \preceq [b]$ :
            $[a] \leftarrow$ $\mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            $[b] \leftarrow$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
            if $[a] = [b]$ $\vee$ ExistChain($G_{\preceq}, [a], [b]$) then
                return true
            No recorded link between $[a]$ and $[b]$. If they both have a determined type, we can try that.
            else if $[a] \in \mathrm{dom}(\mathcal{D}) \wedge [b] \in \mathrm{dom}(\mathcal{D})$ then
                return $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, \mathcal{D}([a]), \mathcal{D}([b])$)
            else
                return undet
        case $[a][\vec{S}_1] \preceq [b][\vec{S}_2]$ :
            $[a] \leftarrow$ $\mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            $[b] \leftarrow$ $\mathcal{Q}$-Find($\mathcal{Q}, [b]$)
            We would like to get the variance sign of the equivalence classes. Since the constraint and the
              applications of $[a]$ and $[b]$ are well-formed, these have the same kind and variance. We can
              extract the variance sign by picking the type representative of $[a]$.
            ($[\vec{v}\vec{X} \triangleleft \_]$ =>> $\_$) $\leftarrow \varsigma([a])$
            if ExistChain($G_{\preceq}, [a], [b]$) $\wedge$ $\mathcal{T}_{EC}$-IsSubtypeVec($\mathscr{K}, \vec{S}_1, \vec{S}_2, \vec{v}$) then
                return true
            else if $[a] \in \mathrm{dom}(\mathcal{D}) \wedge [b] \in \mathrm{dom}(\mathcal{D})$ then
                Same comment applies as above.
                Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
                ($[\vec{v}\vec{X} \triangleleft B_1]$ =>> $\tilde{T}_1$) $\leftarrow \mathcal{D}([a])$
                ($[\vec{v}\vec{X} \triangleleft B_2]$ =>> $\tilde{T}_2$) $\leftarrow \mathcal{D}([b])$
                return $\mathcal{B}_{EC}$-Subsumes($\mathscr{K}, B_2, B_1$) $\wedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathscr{K}, [\vec{X} \mapsto \vec{S}_1]\tilde{T}_1, [\vec{X} \mapsto \vec{S}_2]\tilde{T}_2$)
            else
                return undet
```

```
        case $[a] \preceq T_2$ :
            $[a] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            if $[a] \in \mathrm{dom}(\mathcal{D})$ then
                return $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, \mathcal{D}([a]), T_2$)
            else
                return undet
        case $T_1 \preceq [a]$ :
            $[a] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            if $[a] \in \mathrm{dom}(\mathcal{D})$ then
                return $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, T_1, \mathcal{D}([a])$)
            else
                return undet
        case $[a][\vec{S}] \preceq T_2$ :
            $[a] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            if $[a] \in \mathrm{dom}(\mathcal{D})$ then
                Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
                $([\vec{v}\vec{X} \lhd \_] \texttt{=>>} \tilde{T}_1) \leftarrow \mathcal{D}([a])$
                return $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, [\vec{X} \mapsto \vec{S}]\tilde{T}_1, T_2$)
            else
                return undet
        case $T_1 \preceq [a][\vec{S}]$ :
            $[a] \leftarrow \mathcal{Q}$-Find($\mathcal{Q}, [a]$)
            if $[a] \in \mathrm{dom}(\mathcal{D})$ then
                Note: assumes implicit $\alpha$-renaming to have $\bar{X}$ fresh.
                $([\vec{v}\vec{X} \lhd \_] \texttt{=>>} \tilde{T}_2) \leftarrow \mathcal{D}([a])$
                return $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, T_1, [\vec{X} \mapsto \vec{S}]\tilde{T}_2$)
            else
                return undet
        otherwise :
            return undet
$\mathcal{T}_{EC}$-IsSubtypeVec ($\mathcal{K}$, $\vec{S} : \mathcal{T}_{EC}^N$, $\vec{T} : \mathcal{T}_{EC}^N$, $\vec{v}$) : $K_3$
    Remark: This function is meant to be unfolded within $\mathcal{T}_{EC}$-IsSubtype.
    Remark: Default value for $\vec{v}$ is $(+)^N$.

    res $\leftarrow$ true
    for $i \leftarrow 1$ to $|\vec{S}|$ do
        match $v_i$ :
            case $+$ :
                res $\leftarrow$ res $\curlywedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, S_i, T_i$)
            case $-$ :
                res $\leftarrow$ res $\curlywedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, T_i, S_i$)
            case $\pm$ :
                res $\leftarrow$ res $\curlywedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, S_i, T_i$) $\curlywedge$ $\mathcal{T}_{EC}$-IsSubtype($\mathcal{K}, T_i, S_i$)
    return res
```

B.2.8 Composition of a $\mathcal{T}_{EC}$

```
$\mathcal{T}_{EC}$-Composition ($T : \mathcal{T}_{EC}$, boundTyVars : $\mathscr{P}(\mathcal{V}_X)$) :
        (syms : $\mathscr{P}(\mathcal{S})$, ecsH : $\mathscr{P}(EC_H)$, ecsNH : $\mathscr{P}(EC_H)$, pathDep : $\mathscr{P}(\mathcal{P} \times \mathcal{S})$)
    Description: Decorticate a $\mathcal{T}_{EC}$ into symbols, EC handles, and path-dependent type projections.
    Postcondition: $sym \in$ syms $\implies$ $\mathcal{T}_{EC}$-InHead( _ , $sym, T$)
    Postcondition: $[a] \in$ ecsH $\cup$ ecsNH $\implies$ $[a] \in T$
    Postcondition: $(p, ty) \in$ pathDep $\implies$ $\mathcal{T}_{EC}$-InHead( _ , $p.ty, T$)
    syms, ecsH, ecsNH, pathDep $\leftarrow \emptyset$
    match $T$ :
        Note: For conveniance, we use $\bar{X}$ in the where clause even though $\bar{X}$ is not defined if the first
          pattern is matched. In that case, we default it to $\emptyset$.
        case $Y$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $Y$ where $Y \notin$ boundTyVars $\cup \bar{X}$ :
            syms $\leftarrow$ syms $\cup \{Y\}$
        Same comment applies here and for the body of the case as well.
        case $TyCon[\vec{S}]$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $TyCon[\vec{S}]$
          where $TyCon \notin$ boundTyVars $\cup \bar{X}$ :
            syms $\leftarrow$ syms $\cup \{TyCon\}$
            for $S \in \bar{S}$ do
                ( _ , ecsH$'$, ecsNH$'$, _ ) $\leftarrow$ $\mathcal{T}_{EC}$-Composition($S$, boundTyVars $\cup \bar{X}$)
                ecsNH $\leftarrow$ ecsH$'$ $\cup$ ecsNH$'$
        case $p$.type or $[\vec{v}\vec{X} \triangleleft B]$ =>> $p$.type :
            pathDep $\leftarrow$ pathDep $\cup \{(p, \texttt{type})\}$
        case $p.Q$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $p.Q$ :
            pathDep $\leftarrow$ pathDep $\cup \{(p, Q)\}$
        case $p.F[\vec{S}]$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $p.F[\vec{S}]$ :
            pathDep $\leftarrow$ pathDep $\cup \{(p, F)\}$
            for $S \in \bar{S}$ do
                ( _ , ecsH$'$, ecsNH$'$, _ ) $\leftarrow$ $\mathcal{T}_{EC}$-Composition($S$, boundTyVars $\cup \bar{X}$)
                ecsNH $\leftarrow$ ecsH$'$ $\cup$ ecsNH$'$
        case $[a]$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $[a]$ :
            ecsH $\leftarrow$ ecsH $\cup \{[a]\}$
        case $[a]\left[\vec{S}\right]$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $[a]\left[\vec{S}\right]$
            ecsH $\leftarrow$ ecsH $\cup \{[a]\}$
            for $S \in \bar{S}$ do
                ( _ , ecsH$'$, ecsNH$'$, _ ) $\leftarrow$ $\mathcal{T}_{EC}$-Composition($S$, boundTyVars $\cup \bar{X}$)
                ecsNH $\leftarrow$ ecsH$'$ $\cup$ ecsNH$'$
        case $|_i^n \&_j^{m_i} S_{i,j}$ or $[\vec{v}\vec{X} \triangleleft B]$ =>> $|_i^n \&_j^{m_i} S_{i,j}$ :
            for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $m_i$ do
                match $S_{i,j}$ :
                    (syms$'$, ecsH$'$, ecsNH$'$, pathDep$'$) $\leftarrow$ $\mathcal{T}_{EC}$-Composition($S$, boundTyVars $\cup \bar{X}$)
                    syms $\leftarrow$ syms $\cup$ syms$'$
                    ecsH $\leftarrow$ ecsH $\cup$ ecsH$'$
                    ecsNH $\leftarrow$ ecsNH $\cup$ ecsNH$'$
                    pathDep $\leftarrow$ pathDep $\cup$ pathDep$'$
        otherwise :
            pass
    return (syms, ecsH, ecsNH, pathDep)
```

```
$T_H$-Candidates $(\mathcal{K}, T : \mathcal{T}_{EC},\ \mathsf{boundTyVars} : \mathscr{P}(\mathcal{V}_X)) : \mathsf{res} : \mathscr{P}(T_H)$
    Description: Return the set of type handles that contain the same constituents as a given $\mathcal{T}_{EC}$.
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Postcondition: $\mathsf{res} \subseteq \mathrm{dom}(\Theta)$
    We collect things appearing in $T$ and then remove type handles whose underlying type contain do not
      contain one of those.
    $(\mathsf{tycons}, \mathsf{ecsH}, \_, \mathsf{pathDep}) \leftarrow \mathcal{T}_{EC}$-Composition($T, \mathsf{boundTyVars}$)
    return $(\bigcup \mathrm{Im}(\Theta)) \setminus$
            $\big(\{h : (sym, h) \notin E_{\mathcal{S}},\ sym \in \mathsf{tycons},\ h \in \bigcup \mathrm{Im}(\Theta)\}$
            $\cup \{h : ([a], h) \notin E_{EC},\ [a] \in \mathsf{ecsH},\ h \in \bigcup \mathrm{Im}(\Theta)\}$
            $\cup \{h : ((p, sym), h) \notin E_p,\ (p, sym) \in \mathsf{pathDep},\ h \in \bigcup \mathrm{Im}(\Theta)\}\big)$
```

B.2.9 EC processing

```
$\mathcal{T}$-FindOrCreateECVec $(\mathcal{K}, \vec{T} : \mathcal{T}^N, B_X : \mathcal{B}_{EC}, \vec{v}_X, \mathsf{inHead} : \mathbb{B}, \mathsf{create} : \mathbb{B})$
    Remark: This function is meant to be unfolded within $\mathcal{T}$-FindOrCreateEC.
    $\mathcal{K}' \leftarrow \mathcal{K}$
    $\vec{T}' \leftarrow NIL^{|\vec{T}|}$
    for $i \leftarrow 1$ to $|\vec{T}|$ do
        $(\mathcal{K}^{(n)}, \vec{T}'_i) \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathcal{K}', \vec{T}_i, B_X, \vec{v}_X, \mathsf{inHead}, \mathsf{create})$
        if $\vec{T}'_i = NIL$ then
            return $(\mathcal{K}, NIL)$
        $\mathcal{K}' \leftarrow \mathcal{K}^{(n)}$
    return $(\mathcal{K}', \vec{T}')$
$\mathcal{B}$-FindOrCreateEC $(\mathcal{K}, B_Y : \mathcal{B}, \vec{v}_Y\ B_X : \mathcal{B}, \vec{v}_X, \mathsf{create} : \mathbb{B})$
    Remark: This function is meant to be unfolded within $\mathcal{T}$-FindOrCreateEC.
    $\vec{Y} \leftarrow \mathrm{dom}(B_X)$
    We will recur on the bounds indicated in $B_Y : \mathcal{B}$ in order to build a $B'_Y : \mathcal{B}_{EC}$. For that, we prepare a
      $B_{tmp}$ with the enclosing $B_X$.
    To have a well-formed $B_{tmp}$, we need to somehow give a $B'_Y$ to $B_{tmp}$, but it is the thing we are trying to
      build. It is sufficient to give trivial bounds.
    Concatenate $B_X$ and the trivial bounds.
    $B_{tmp} \leftarrow B_X \cdot (Y \mapsto (\bot_{\mathrm{kind}(Y)}, \top_{\mathrm{kind}(Y)}), Y \in \vec{Y})$
    $\mathcal{K}' \leftarrow \mathcal{K}$
    The $B'_Y : \mathcal{B}_{EC}$ that we will build
    $B'_Y \leftarrow \emptyset$
    for $(Y_i, (L_i, U_i)) \in B_Y$ do
        $(\mathcal{K}^{(a)}, L'_i) \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathcal{K}', L_i, B_{tmp}, \vec{v}_X\vec{v}_Y, \mathsf{true}, \mathsf{create})$
        $(\mathcal{K}^{(n)}, U'_i) \leftarrow \mathcal{T}$-FindOrCreateEC$(\mathcal{K}^{(a)}, U_i, B_{tmp}, \vec{v}_X\vec{v}_Y, \mathsf{true}, \mathsf{create})$
        if $L'_i = NIL \vee U'_i = NIL$ then
            return $(\mathcal{K}, NIL)$
        $\mathcal{K}' \leftarrow \mathcal{K}^{(n)}$
        $B'_Y \leftarrow B^{(n)} \cdot (Y_i \mapsto (L'_i, U'_i))$
    return $(\mathcal{K}', B'_Y)$
$\mathcal{T}_{EC}$-TryFindECOfApplied $(\mathcal{K}, [a]\left[\vec{S}\right])$
    Precondition: $\mathcal{K}$-Valid$(\mathcal{K}) \wedge \mathrm{ftv}(\vec{S}) = \emptyset \wedge \varsigma([a]\left[\vec{S}\right])\downarrow$
    Postcondition: Similar to Q-FEC2, Q-FEC3, Q-FEC7
    for $h \in T_H$-Candidates$(\mathcal{K}, [a]\left[\vec{S}\right], \emptyset)$ do
        match $\Theta(h)$ :
            case $[b]\left[\vec{U}\right]$ where $\mathcal{Q}$-Find$(\mathcal{Q}, [a])$ = $\mathcal{Q}$-Find$(\mathcal{Q}, [b])$:
                if $\mathcal{T}_{EC}$-EquivVec$(\mathcal{K}, \vec{S}, \vec{U})$ then
                    return $(\mathcal{K}, \mathcal{R}(h))$
            otherwise :
                continue
    return $NIL$
```

B.2.10 Matching

```
$\mathcal{T}_{EC}$-TryMatch ($\mathscr{K}$, $\bar{X} : \mathscr{P}(\mathcal{V}_X)$, $T : \mathcal{T}_{EC}$, $S : \mathcal{T}_{EC}$)
    Description: Attempt to match the type variables $\bar{X}$ appearing in $T$ with respect to $S$.
      $S$ may not contain any type variable in $\bar{X}$. If $T$ and $S$ "match", a non-nil $\sigma$ representing the
      substitution of $\bar{X}$ is returned. Note that $\sigma$ may be empty, in which case no $\bar{X}$ appear in $T$.
    Example: With $\bar{X} = \{X_1, X_2\}$, $T = F[X_1, \texttt{Int}]$ and $S = F[\texttt{Foo}, \texttt{Int}]$, we get $\sigma = [X_1 \mapsto \texttt{Foo}]$.
    Example: With $\bar{X} = \{X_1, X_2\}$, $T = F[X_1, \texttt{Int}]$ and $S = F[\texttt{Foo}, Y]$, we get $\sigma = NIL$, assuming that $Y$
      is distinct from $X_1$ and $X_2$ and that $\mathscr{K}$ does not have any information about $Y$.
    Precondition: $\mathscr{K}\texttt{-WellFormed}(\mathscr{K}) \wedge \varsigma(T)\downarrow \wedge \varsigma(S)\downarrow$
    Precondition: $\bar{X} \# (\mathrm{ftv}(S) \cup \mathrm{ftv}(\mathscr{K}))$
    Postcondition: $\sigma \neq NIL \implies \mathrm{dom}(\sigma) \subseteq \bar{X} \wedge \varsigma(\sigma(T))\downarrow \wedge \mathscr{K} \Vdash \varsigma(\sigma(T)) \asymp \varsigma(S)$
    if $\bar{X} \# \mathrm{ftv}(T)$ then
        if $\mathcal{T}_{EC}\texttt{-Equiv}(\mathscr{K}, T, S)$ then
            return $\emptyset$
        else
            return $NIL$
    match $(T, S)$ :
        case $(X, S)$ where $X \in \bar{X}$ :
            return $[X \mapsto S]$
        case $([a][\vec{U}], [b][\vec{V}])$ where $\mathcal{Q}\texttt{-Find}(\mathcal{Q}, [a]) = \mathcal{Q}\texttt{-Find}(\mathcal{Q}, [b])$:
            return $\mathcal{T}_{EC}\texttt{-TryMatchVec}(\mathscr{K}, \bar{X}, \vec{U}, \vec{V})$
        case $(TyCon[\vec{U}], TyCon[\vec{V}])$ :
            return $\mathcal{T}_{EC}\texttt{-TryMatchVec}(\mathscr{K}, \bar{X}, \vec{U}, \vec{V})$
        case $([\vec{v}\vec{Y} \triangleleft B_1] \texttt{=>>} U,\ [\vec{v}\vec{Y} \triangleleft B_2] \texttt{=>>} V)$ :
            $\vec{Z} \leftarrow$ fresh type variables of same length and kind as $\vec{Y}$
            $\sigma_{\mathsf{body}} \leftarrow \mathcal{T}_{EC}\texttt{-TryMatch}(\mathscr{K}, \bar{X} \cup \bar{Z}, [\vec{Y} \mapsto \vec{Z}]U, [\vec{Y} \mapsto \vec{Z}]V)$
            $\sigma_B \leftarrow \emptyset$
            Matching the upper and lower bounds of $\vec{Z}$
            for $Z \in \bar{Z}$ do
                $(L_1, U_1) \leftarrow ([\vec{Y} \mapsto \vec{Z}]B_1)(Z)$
                $(L_2, U_2) \leftarrow ([\vec{Y} \mapsto \vec{Z}]B_2)(Z)$
                $\sigma_L \leftarrow \mathcal{T}_{EC}\texttt{-TryMatch}(\mathscr{K}, \bar{X} \cup \bar{Z}, [\vec{Y} \mapsto \vec{Z}]L_1, [\vec{Y} \mapsto \vec{Z}]L_2)$
                $\sigma_U \leftarrow \mathcal{T}_{EC}\texttt{-TryMatch}(\mathscr{K}, \bar{X} \cup \bar{Z}, [\vec{Y} \mapsto \vec{Z}]U_1, [\vec{Y} \mapsto \vec{Z}]U_2)$
                $\sigma_B \leftarrow \mathcal{T}_{EC}\texttt{-TryCombineSubstMatch}(\sigma_B,\ \mathcal{T}_{EC}\texttt{-TryCombineSubstMatch}(\sigma_L, \sigma_U))$
            $\sigma \leftarrow \mathcal{T}_{EC}\texttt{-TryCombineSubstMatch}(\sigma_{\mathsf{body}}, \sigma_B)$
            if $\sigma = NIL$ then
                return $NIL$
            We ensure that the $\vec{Z}$ are matched against each other, and that they do not appear in the returned
              solution.
            $\sigma_{\bar{Z}} \leftarrow \sigma \restriction \bar{Z}$
            $\sigma_{\bar{Z}^c} \leftarrow \sigma \restriction (\mathrm{dom}(\sigma) \setminus \bar{Z})$
            if $\sigma_{\bar{Z}} = \{\vec{Z} \mapsto \vec{Z}\} \wedge \mathrm{ftv}(\sigma_{\bar{Z}^c}) \# \bar{Z}$ then
                return $\sigma_{\bar{Z}^c}$
            else
                return $NIL$
```

```
        Note: matches the $i$ and $m_j$ as well. Furthemore, we assume that the $m_j$ are sorted in an ascending order; that is, $m_1 \leq m_2, ..., \leq m_n$.
        case ($|_i^n \&_j^{m_i} T_{i,j},\ |_i^n \&_j^{m_i} S_{i,j}$) :
            $\sigma_{\mathsf{acc}} \leftarrow \emptyset$
            for $\bar{i} \in \left\{\{i' : m_i = m_{i'},\ 1 \leq i' \leq n\} : 1 \leq i \leq n\right\}$ do
                $\sigma_{\mathsf{conj}} \leftarrow NIL$
                Trying to find a match for $\&_j^{m_i} T_{i,j}$ and $\&_j^{m_i} S_{i,j}$ for $i$ ranging in $\bar{i}$
                for $i \in \bar{i}$ do
                    $\sigma \leftarrow \mathcal{T}_{EC}$-TryMatchConjunct($\mathcal{K}, \bar{X}, \&_j^{m_i} T_{i,j}, \&_j^{m_i} S_{i,j}$)
                    A $NIL$ $\sigma$ means that this particular matching failed but there are other matching to try, so we keep on
                    if $\sigma \neq NIL$ then
                        if $\sigma_{\mathsf{conj}} = NIL$ then
                            $\sigma_{\mathsf{conj}} \leftarrow \sigma$
                        We have already found a matching before. We want all found matching to be equivalent since we do not accept ambiguous substitutions
                        else if $\neg\mathcal{T}_{EC}$-EquivSubstMatch($\mathcal{K}, \sigma_{\mathsf{conj}}, \sigma$) then
                            return $NIL$
                if $\sigma_{\mathsf{conj}} = NIL$ then
                    return $NIL$
                else
                    $\sigma_{\mathsf{acc}} \leftarrow \mathcal{T}_{EC}$-TryCombineSubstMatch($\sigma_{\mathsf{acc}}, \sigma_{\mathsf{conj}}$)
            return $\sigma_{\mathsf{acc}}$
        otherwise :
            return $NIL$
$\mathcal{T}_{EC}$-TryMatchConjunct ($\mathcal{K}$, $\bar{X} : \mathcal{P}(\mathcal{V}_X)$, $\&_j^m T_j : \mathcal{T}_{EC}$, $\&_j^m S_j : \mathcal{T}_{EC}$)
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $\bar{X} \# \mathrm{ftv}(\&_j^m S_j)$
    Postcondition: $\sigma \neq NIL \implies \mathrm{dom}(\sigma) \subseteq \bar{X} \wedge \mathcal{T}_{EC}$-Equiv($\mathcal{K}, \sigma(\&_j^m T_j), \&_j^m S_j$)

    $\sigma_{\mathsf{acc}} \leftarrow \emptyset$
    for $j \leftarrow 1$ to $m$ do
        $\sigma_j \leftarrow NIL$
        for $j' \leftarrow 1$ to $m$ do
            This is similar to what we do with whole conjuncts in $\mathcal{T}_{EC}$-TryMatch
            $\sigma \leftarrow \mathcal{T}_{EC}$-TryMatch($\mathcal{K}, \bar{X}, T_j, S_{j'}$)
            if $\sigma \neq NIL$ then
                if $\sigma_j = NIL$ then
                    $\sigma_j \leftarrow \sigma$
                else if $\neg\mathcal{T}_{EC}$-EquivSubstMatch($\mathcal{K}, \sigma_j, \sigma$) then
                    return $NIL$
        if $\sigma_j = NIL$ then
            return $NIL$
        else
            $\sigma_{\mathsf{acc}} \leftarrow \mathcal{T}_{EC}$-TryCombineSubstMatch($\sigma_{\mathsf{acc}}, \sigma_j$)
    return $\sigma_{\mathsf{acc}}$
```

```
$\mathcal{T}_{EC}$-TryMatchVec ($\mathcal{K}$, $\bar{X} : \mathcal{P}(\mathcal{V}_X)$, $\vec{T} : \mathcal{T}_{EC}^N$, $\vec{S} : \mathcal{T}_{EC}^N$)
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $|\vec{T}| = |\vec{S}| \wedge \bar{X} \# \mathrm{ftv}(\vec{S})$
    Postcondition: $\sigma \neq NIL \implies \mathrm{dom}(\sigma) \subseteq \bar{X} \wedge$ $\mathcal{T}_{EC}$-EquivVec($\mathcal{K}, \sigma(\vec{T}), \vec{S}$)
    $\sigma \leftarrow \emptyset$
    for $i \leftarrow 1$ to $|\vec{T}|$ do
        $\sigma \leftarrow$ $\mathcal{T}_{EC}$-TryCombineSubstMatch($\sigma$, $\mathcal{T}_{EC}$-TryMatch($\mathcal{K}, \bar{X}, T_i, S_i$))
    return $\sigma$
$\mathcal{T}_{EC}$-TryCombineSubstMatch ($\mathcal{K}$, $\sigma_1 : \mathcal{V}_X \rightharpoonup \mathcal{T}_{EC} \cup NIL$, $\sigma_2 : \mathcal{V}_X \rightharpoonup \mathcal{T}_{EC} \cup NIL$)
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $\sigma_1 \neq NIL \wedge \sigma_2 \neq NIL \implies (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)) \# (\mathrm{ftv}(\sigma_1) \cup \mathrm{ftv}(\sigma_2))$
    if $\sigma_1 = NIL \vee \sigma_2 = NIL$ then
        return $NIL$
    $\sigma \leftarrow \big(\sigma_1 \upharpoonright (\mathrm{dom}(\sigma_1) \setminus \mathrm{dom}(\sigma_2))\big) \cup \big(\sigma_2 \upharpoonright (\mathrm{dom}(\sigma_2) \setminus \mathrm{dom}(\sigma_1))\big)$
    for $X \in \mathrm{dom}(subst_1) \cap \mathrm{dom}(subst_2)$ do
        if $\mathcal{T}_{EC}$-Equiv($\mathcal{K}, \sigma_1(X), \sigma_2(X)$) then
            $\sigma \leftarrow \sigma \cup \{X \mapsto \sigma_1(X)\}$
        else
            return $NIL$
    return $\sigma$
$\mathcal{T}_{EC}$-EquivSubstMatch ($\mathcal{K}$, $\sigma_1 : \mathcal{V}_X \rightharpoonup \mathcal{T}_{EC}$, $\sigma_2 : \mathcal{V}_X \rightharpoonup \mathcal{T}_{EC}$)
    Precondition: $\mathcal{K}$-WellFormed($\mathcal{K}$)
    Precondition: $(\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)) \# (\mathrm{ftv}(\sigma_1) \cup \mathrm{ftv}(\sigma_2))$
    return $\mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2) \wedge (\forall X \in \mathrm{dom}(\sigma_1).$ $\mathcal{T}_{EC}$-Equiv($\mathcal{K}, \sigma_1(X), \sigma_2(X)$))
```