Higher-ranked Polymorphism in Dependency Analyses

Fabian Thorand

Master's Thesis
ICA-5607000

July 5, 2017

Department of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Supervisor:
dr. J. Hage

Second Examiner:
dr. W.S. Swierstra

Abstract

Automated program analyses are useful tools for verification and optimization, naming only two important use cases. In this thesis, we focus on type-based analyses for functional programming languages tracking certain forms of program dependencies. Some uncover dependencies of outputs on inputs, such as binding-time or information flow security analysis. Others find which parts of a program contribute to its final result, for example call-tracking or slicing analysis.

It has been found by Abadi et al. that all these analyses (and more) share a common structure in their respective type and effect systems. Their *dependency core calculus* (DCC) formalizes these commonalities and serves as a framework for dependency analyses. An appealing consequence of this unification is the fact that certain results about the analyses in question only need to be proven once.

One such result is the *noninterference* theorem. It states that, for example in the context of binding-time analysis, expressions deemed evaluable at compile-time cannot access dynamic values. It is therefore an important safety guarantee for subsequently using the analysis results.

However, the dependency core calculus is a monovariant system which limits its precision. A relatively recent technique for improving the quality of results of type based analyses is higher-ranked polyvariance. It has been successfully applied to flow analysis by Holdermans and Hage and later to exception analysis by Koot.

In this thesis, we devise a type and effect system based on DCC, but equipped with higher-ranked polyvariance. On top of that, we prove a noninterference theorem with respect to a call-by-name operational semantics. Moreover, we adapt the type reconstruction algorithm by Koot to our dependency analysis and prove that it always yields correct results.

Acknowledgments

First of all, I am very grateful to my supervisor Jurriaan Hage for his guidance and valuable advice during the writing of this thesis. Moreover, I would like to thank my friends who were helpful both in academic matters and in providing the needed distractions. Lastly, I would like to express my deepest gratitude to my parents for their continuous support throughout my life.

Contents

1. Introduction **7**
1.1. Type-Based Analyses 7
1.1.1. Type Systems 7
1.1.2. Type and Effect Systems 8
1.2. Dependency Core Calculus 9
1.3. Contributions 10
1.4. Structure 10

2. Declarative Type System **12**
2.1. Source Language 13
2.1.1. Syntax 13
2.1.2. Underlying Type System 14
2.2. $\lambda^{\sqcup}$-calculus 16
2.2.1. Syntax 16
2.2.2. Sorting 17
2.2.3. Denotational Semantics 19
2.3. Target Language 23
2.3.1. Terms and Types 23
2.3.2. Subtyping 27
2.3.3. Type and Effect System 31
2.4. Embedding Various Analyses 36
2.4.1. Binding-Time Analysis 36
2.4.2. Security Analysis 36
2.4.3. Exception Analysis 38
2.5. Operational Semantics 38
2.6. Noninterference 42

3. Type Reconstruction **46**
3.1. Equality of Dependency Terms 46
3.1.1. Reduction 47
3.1.2. Canonical Forms 49
3.1.3. Canonical Ordering 50
3.2. Ensuring Modularity 52
3.2.1. Pattern Types 52
3.2.2. Conservative Types 55
3.2.3. Type Completion 57
3.2.4. Least Types 59

3.3. Reconstruction Algorithm . . . 61
3.4. Problems with Earlier Approaches . . . 66
3.5. Correctness of the Algorithm . . . 67
3.5.1. Soundness . . . 67
3.5.2. Completeness . . . 68
3.5.3. Best Analyses . . . 75

4. Implementation **82**
4.1. Additional Features . . . 82
4.2. Project Structure . . . 83
4.3. REPL . . . 83

5. Evaluation **86**
5.1. Construction and Elimination . . . 86
5.2. Polymorphic Recursion . . . 87
5.3. Higher-Ranked Polyvariance . . . 88

6. Related Work **92**
6.1. Region Analysis . . . 92
6.2. Resource Analysis . . . 93

7. Conclusion and Future Work **94**

Bibliography **97**

A. Proofs **99**

List of Figures

2.1. Source language: syntax . . . 13
2.2. Underlying type system ($\Gamma \vdash_{\mathrm{t}} t : \tau$) . . . 15
2.3. $\lambda^{\sqcup}$-calculus: syntax . . . 16
2.4. $\lambda^{\sqcup}$-calculus: sorting rules ($\Sigma \vdash_{\mathrm{s}} \xi : \kappa$) . . . 17
2.5. $\lambda^{\sqcup}$-calculus: denotational semantics . . . 20
2.6. Target language: syntax . . . 24
2.7. Well-formedness of annotated types ($\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}$) . . . 26
2.8. Subtyping relation ($\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$) . . . 28
2.9. Declarative type and effect system ($\Sigma \mid \Gamma \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \;\&\; \xi$) . . . 32
2.10. Values in the target language . . . 39
2.11. Evaluation contexts . . . 39
2.12. Small-step semantics ($t_1 \to t_2$) . . . 40

3.1. $\lambda^{\sqcup}$-calculus: reduction rules . . . 48
3.2. Canonical order on dependency terms ($\Sigma_1 \mid \Sigma_2 \vdash \xi_1 <_C \xi_2$) . . . 50
3.3. Pattern types ($\Sigma \vdash_p \widehat{\tau} \;\&\; \xi \triangleright \Sigma'$) . . . 53
3.4. Type completion ($\Sigma \vdash_c \tau : \widehat{\tau} \;\&\; \xi \triangleright \Sigma'$) . . . 57
3.5. Type reconstruction algorithm ($\mathcal{R}$) . . . 63
3.6. Least upper bound of types ($\sqcup$) . . . 63
3.7. Completion algorithm ($\mathcal{C}$) . . . 64
3.8. Instantiation algorithm ($\mathcal{I}$) . . . 64
3.9. Matching algorithm ($\mathcal{M}$) . . . 65

4.1. Concrete syntax of the source language . . . 85

1. Introduction

Many type-based analyses for functional programming languages are tracking some form of program dependencies. Some examples are binding-time analysis, secure information flow analysis and slicing analysis. All these analyses share some commonalities that have been formalized in the *dependency core calculus* by Abadi et al. [1].

First of all, we explain the kind of type based analyses being in the focus of this thesis. The subsequent section provides a brief description of the dependency core calculus. This chapter is concluded by outlining the contributions made by this thesis.

1.1. Type-Based Analyses

We start by giving an account of common features of type systems for functional languages. Type-based analyses are extensions of such type systems that keep track of additional information about the program in question.

1.1.1. Type Systems

Type systems are used to statically check that a program only executes well-defined actions at runtime. The basis for most type systems of functional programming languages is the Hindley-Milner type system used in ML. Milner proved for the polymorphic ML type system that "Well-typed expressions do not go wrong" [18]. This means that the execution of well-typed programs according to the semantics does not get stuck due to undefined behavior (e.g. trying to use an integer where a function is needed).

This section provides a broad overview over the topics related to the Hindley-Milner type discipline that are relevant for the subject of the thesis.

Algorithm W A popular type inference algorithm for the Hindley-Milner type system is the so called *Algorithm W*, described by Milner [18] extending prior work by Hindley [7]. It has the important property that it always infers the so called *principal type* of an expression, i.e. the most general type that can be instantiated to any more specific type that could be assigned to that expression.

Higher-ranked Polymorphism The original Hindley-Milner type system only allows let-polymorphism. Quantification of type variables is only possible for let-bound values, while arguments to lambda expressions are restricted to monomorphic types.

Kfoury and Tiuryn [13] gave an account of type reconstruction in higher-ranked lambda calculi. They show that reconstruction for rank 2 types is polynomial-time equivalent to the ML type system and that reconstruction for rank $k > 2$ is undecidable.

Polymorphic recursion The original ML type system only allows monomorphic recursion, where recursive functions could only be used polymorphically in subsequent definitions. Mycroft found real-world examples where monomorphic recursion prevents typing of an otherwise valid function definition and therefore proposed an extension to the ML type system consisting of a polymorphic rule for the fixpoint combinator [19]. This allows to derive types where multiple recursive call sites can use different instantiations of the polymorphically typed recursive function.

For computing the type of polymorphically recursive functions, Mycroft devised an algorithm based on fixpoint iteration on the complete partial order of type schemes with an artificial top element denoting failure.

It has later been shown by Henglein in "Type inference with polymorphic recursion" [6] that type inference in the presence of polymorphic recursion is undecidable by reducing the problem to semi-unification. The latter has been proved undecidable by Kfoury, Tiuryn, and Urzyczyn [14].

1.1.2. Type and Effect Systems

A *type and effect system* usually consists of an underlying type system, enriched by adding annotations and effects to types. Those capture additional properties of the expressions, resulting in a better approximation of the dynamic semantics than conventional types alone.

Sometimes, an analysis can lose precision due to restrictions in the type system. This happens because the annotations and effects of the formal parameters of functions must match the actual arguments. Wansbrough and Jones identified this problem as the *poisoning problem*, because one call to a function poisons the other use sites [25].

The remainder of this section consists of techniques that are used for alleviating the poisoning problem. A general overview over type and effect systems is given by Nielson and Nielson in their 1999 paper "Type and Effect Systems" [20].

Subeffecting, subtyping and polyvariance *Subeffecting* describes the "enlarging" of effects on the outermost position, thereby making the analysis more imprecise. However, subeffecting is usually necessary to match the types of different branches of the program (e.g. in conditionals). The enlarging of effects happens early, so that all demands by the type and effect system are met when they occur later on in nested positions [20].

Subtyping introduces a subsumption relation between types, allowing to "adapt" types to a certain degree at use-sites [20]. While *subeffecting* only considers the topmost effect on a type, *subtyping* extends to nested positions (i.e. function arguments/return types) as well. This makes it possible to delay the widening, therefore losing less precision than *subeffecting*.

Polyvariance denotes polymorphism over the annotations in a type and effect system. It allows for a greater precision by having different instantiations of annotations at each use-site of bindings [20].

Higher-ranked Polyvariance Analogously to rank-1 polymorphism, rank-1 polyvariance only allows universal quantification of annotation and effect variables at the topmost level of a function type. On the contrary, higher-ranked polyvariance allows quantification to occur anywhere in a type and is another measure for increasing the precision of a type based analysis.

While rank-1 polyvariance is prevalent in a lot of type and effect systems throughout the available literature, there is only a small number that is using higher-ranked polyvariance. Two examples with slightly different approaches are “Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators” by Holdermans and Hage [10] and “Higher-ranked Exception Types” by Koot [15].

Although, as outlined earlier, type inference in general is undecidable for higher-ranked polymorphism of rank three or higher, Holdermans and Hage note that the inference of polyvariant annotations remains decidable even in a higher-ranked setting. Essentially, this is because the underlying types are known and the places where annotations and quantifications are added are uniquely determined by the underlying type.

1.2. Dependency Core Calculus

The term *dependency analysis* is used by Abadi et al. to describe a certain type of program analysis that computes dependencies between program values of some sort. They devised a formal framework capturing the commonalities of these analyses, called the *Dependency Core Calculus* (DCC) [1]. Notably, DCC is a monovariant type and effect system with subtyping.

Annotations form a lattice and the type system distinguishes annotated values from unannotated values through monadic type constructors, one for each value of the lattice. The monadic structure is also reflected at the term level through the usual bind and return operations. It is noteworthy that the typing rule of the bind operation allows to return an annotation greater than or equal to the one that has been passed in.

Besides the annotation monads, there is also a special monadic type for creating pointed types (see [11]), designating expressions that might not terminate.

By choosing an appropriate lattice, DCC can be adapted to a variety of analyses, such as *binding-time analysis*, *security analysis* and *slicing analysis*.

Binding-time analysis is used in partial evaluation to find expressions that can be evaluated at compile time, as opposed to those that can only be computed at runtime because they depend on dynamic values such as user input. Zhang compared the precision and resource usage of several variants of a type-based binding-time analysis with respect to polyvariance, subeffecting and subtyping in his 2008 Master’s thesis “Binding-Time Analysis: Subtyping versus Subeffecting” [26].

Security analysis classifies values in “high-security” and “low-security” (or more intermediate classes), making sure that computations that lead to low-security values cannot depend on high-security values. An example for a type and effect system performing this kind of analysis is the SLam calculus by Heintze and Riecke [5].

Slicing analysis computes the set of program parts that are actually used in producing

the result [23]. As an example, consider the expression $(\lambda x.\, \mathit{fst}\ x + 1)\ (e_1, e_2)$. Only the expression e_1 and the lambda function contribute to the result of the program, whereas e_2 is not used.

All theses analyses have in common that they compute some form of dependency between outputs and inputs of a computation. Furthermore, they all share the property that values with lower classification cannot depend on values with higher classification. For example, a value with static binding-time cannot depend on a dynamic one, a low-security value cannot depend on a high-security one, and a computation deemed to only use a certain set of program parts cannot use values whose computation depends on other parts. This is often called the *noninterference property* in literature [16].

1.3. Contributions

First of all, we devise a type and effect system akin to DCC, but utilizing higher-ranked polyvariance with the goal of improving the precision of the analysis. Similar to DCC, it works with an arbitrary lattice abstracting over concrete analyses. This is a generalization of the exception analysis by Koot.

Secondly, we prove a noninterference theorem for our system with respect to a call-by-name operational semantics.

Additionally, we present a type reconstruction algorithm based on Koot's exception analysis. In the process, a prototype implementation of the type reconstruction has also been developed. In doing so, we uncovered and fixed a flaw in the handling of polymorphic recursion in the previous work that caused the type reconstruction to diverge on certain inputs.

Besides the algorithm itself, we provide soundness and completeness proofs ensuring its correctness with respect to the type and effect system. In particular, the fixpoint iteration used for inferring types and effects in the presence of polymorphic recursion always terminates. Additionally, we give a partial proof that the type reconstruction always yields the best analysis, under some restrictions.

1.4. Structure

Chapter 2 starts by defining the source language, i.e. the language of the programs that can be fed into the analysis. We proceed by presenting the $\lambda^{\sqcup}$-calculus, a simply typed lambda calculus enriched with a lattice structure that is used for representing effects in our analysis. It is based on the $\lambda^{\cup}$-calculus by Koot which in turn is modeled after the effect operators used by Holdermans and Hage.

Then, the target language is presented. It makes effects and annotations explicitly visible on the term level. This language is used for defining the type and effect system for higher-ranked dependency analyses. By means of several examples we show how the system can be instantiated to concrete analyses. The chapter concludes with a call-by-name operational semantics for the target language and the noninterference theorem.

The goal of chapter 3 is to introduce the type reconstruction algorithm and to provide its correctness proofs. But first, various definitions are introduced. It starts by outlining the canonicalization procedure that has been used by Koot for deciding term equality. We note that even though that approach turned out to be incomplete, equality of $\lambda^{\sqcup}$-terms is decidable for finite lattices. After that, the notions of *conservative* and *pattern* types are introduced, as they are essential for the modularity of the analysis. Only then, the type reconstruction algorithm and various auxiliary procedures are introduced. At this point, a more in-depth explanation follows as to why the previous approaches were not fully correct. We also present some ideas for improving the decision procedure for $\lambda^{\sqcup}$-term equality. The last section of this chapter is devoted to the correctness proofs.

Next, chapter 4 briefly introduces the prototype implementation of the analysis that has been developed alongside this thesis.

In chapter 5, the empirical results of the analysis are scrutinized. In particular, we identify some programs that benefit from a higher-ranked analysis, i.e. where non-higher-ranked analyses cannot provide the same precision.

Lastly, we present related work in chapter 6, in particular a different system for resource analysis that achieves similar precision to what could be expected from higher-ranked polyvariance. Then, we finish with a conclusion and several starting points for future work in chapter 7.

Appendix A holds the particularly lengthy proofs of some lemmas and theorems.

2. Declarative Type System

In this chapter we will lay the foundation for our dependency analysis with higher-ranked polyvariance. Similar to the dependency core calculus by Abadi et al., our analysis will be parameterized over the actual dependencies being traced. These dependencies are represented by values from some bounded, finite join-semilattice $\mathcal{L}$, defined as follows.

Definition 2.1. A *join-semilattice* $\mathcal{L}$ is an algebraic structure $\langle L, \sqcup \rangle$ consisting of an underlying set L and an associative, commutative and idempotent binary operation $\sqcup$, called *join*. Often, we also let $\mathcal{L}$ denote the underlying set.

The binary operation of a join-semilattice induces a binary relation on L given by

$$x \sqsubseteq y \iff x \sqcup y = y, \text{ for any } x, y \in L.$$

The join $x \sqcup y$ of two elements is the least upper bound of x and y w.r.t. $\sqsubseteq$.

A join-semilattice $\mathcal{L}$ is *bounded* if there exists a least element $\bot$ (called *bottom*) such that for all $x \in \mathcal{L}$ we have $x \sqcup \bot = x$.

A join-semilattice $\mathcal{L}$ is *finite* if the underlying set is finite.

Although the constructions in this chapter do not depend on the lattice to be finite, this condition is needed for the completeness of the algorithm presented in the next chapter.

The definitions in this chapter therefore assume that there is such a lattice, but they are entirely independent of the concrete lattice being chosen. Usually, we will use the letter ℓ to denote an element of the lattice $\mathcal{L}$. Because the lattice $\mathcal{L}$ determines the kind of dependencies traced by the analysis, we will also call it the *dependency lattice*.

We start by defining a *source language* of programs to be analyzed. This includes a description of the *underlying type system* relating the terms and types of the source language.

The next step is the definition of a simply-typed lambda calculus extended with a lattice structure that will be used for representing *dependency annotations* and *effects* in the type and effect system.

Then, we build a *target language* that makes the dependency annotations explicit on both the type and the term level. Subsequently, we can define the *declarative type and effect system* for our dependency analysis. It will relate annotated terms to annotated types and form the basis for the syntax-driven analysis defined in the next chapter.

We will also present a few examples of how our system can perform various analyses by defining a translation into the source language and choosing the lattice $\mathcal{L}$ accordingly.

The chapter concludes with a small-step semantics for the target language and the noninterference theorem built thereupon.

Types:

$$
\begin{array}{llll}
\tau \in \mathbf{Ty} & ::= & \text{unit} & \text{(unit type)} \\
& \mid & \tau_1 + \tau_2 & \text{(sum type)} \\
& \mid & \tau_1 \times \tau_2 & \text{(product type)} \\
& \mid & \tau_1 \to \tau_2 & \text{(function type)}
\end{array}
$$

Terms:

$$
\begin{array}{llll}
x \in \mathbf{Var} & & & \text{(term variables)} \\
\ell \in \mathcal{L} & & & \text{(underlying lattice)} \\
t \in \mathbf{Tm} & ::= & x & \text{(variable)} \\
& \mid & () & \text{(unit constructor)} \\
& \mid & \lambda x : \tau . t & \text{(abstraction)} \\
& \mid & t_1 \ t_2 & \text{(application)} \\
& \mid & (t_1, t_2) & \text{(pair constructor)} \\
& \mid & \text{proj}_i(t) & \text{(pair projections)} \\
& \mid & \text{inl}_{\tau_2}(t) \mid \text{inr}_{\tau_1}(t) & \text{(sum constructors)} \\
& \mid & \textbf{case } t \textbf{ of } \{\text{inl}(x) \to t_1; \text{inr}(y) \to t_2\} & \text{(sum eliminator)} \\
& \mid & \mu x : \tau . t & \text{(fixpoint)} \\
& \mid & \text{seq } t_1 \ t_2 & \text{(forcing)} \\
& \mid & \text{ann}_\ell(t) & \text{(raise annotation level)}
\end{array}
$$

Figure 2.1.: Source language: syntax

2.1. Source Language

The source language is largely based on the language of the dependency core calculus by Abadi et al. [1]. It is an explicitly typed call-by-name lambda calculus extended with basic data types and general recursion.

2.1.1. Syntax

Figure 2.1 shows the syntax of the types and terms of our source language. In addition to function types, it features a unit type as well as sum and product types. We use the letter τ to refer to underlying types and the letter t to refer to terms of the source language.

On the term level, products are introduced by the pair constructor (t_1, t_2) and eliminated through projection $\text{proj}_i(t)$ on one of the components ($i \in \{1, 2\}$). Similarly, sums are introduced via injection in one of the alternatives $\text{inl}_{\tau_2}(t)$ or $\text{inr}_{\tau_1}(t)$ and eliminated through case expressions. Due to the explicit typing, the sum constructors are annotated with the type of the other alternative, as the argument passed to the constructor only determines the type of one alternative of the sum type.

The language supports general recursion by means of a fixpoint combinator $\mu x : \tau.t$ that binds x in t to the value that t will eventually evaluate to. Being a call-by-name calculus, there is also a primitive seq $t_1\ t_2$ that explicitly forces the evaluation of t_1 before returning t_2.

The most important addition is the $\mathrm{ann}_\ell(t)$ expression. Operationally, it behaves just like the identity function. However, it is used in the annotated type system described further below for ensuring that the dependency annotation of the term t is at least as large as $\ell \in \mathcal{L}$.

Note that the term language does not provide let-bindings. For now, they are not required because the underlying type system does not support polymorphism, and hence bindings of the form **let** $x : \tau = t_1$ **in** t_2 can be written as $(\lambda x : \tau.t_2)\ t_1$ when they are non-recursive, or as $(\lambda x : \tau.t_2)\ (\mu x : \tau.t_1)$ when they are recursive.

Definition 2.2. We denote the set of free (term) variables in a source term t by $\mathrm{ftv}(t)$, defined as follows:

$$
\begin{array}{rcl}
\mathrm{ftv}(x) & = & \{x\} \\
\mathrm{ftv}(()) & = & \emptyset \\
\mathrm{ftv}(\lambda x : \tau.t) & = & \mathrm{ftv}(t) \setminus \{x\} \\
\mathrm{ftv}(t_1\ t_2) & = & \mathrm{ftv}(t_1) \cup \mathrm{ftv}(t_2) \\
\mathrm{ftv}((t_1, t_2)) & = & \mathrm{ftv}(t_1) \cup \mathrm{ftv}(t_2) \\
\mathrm{ftv}(\mathrm{proj}_i(t)) & = & \mathrm{ftv}(t) \\
\mathrm{ftv}(\mathrm{inl}_\tau(t)) & = & \mathrm{ftv}(t) \\
\mathrm{ftv}(\mathrm{inr}_\tau(t)) & = & \mathrm{ftv}(t) \\
\mathrm{ftv}(\mathbf{case}\ t\ \mathbf{of}\ \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(y) \to t_2\}) & = & \mathrm{ftv}(t_1) \cup (\mathrm{ftv}(t_2) \setminus \{x\}) \cup (\mathrm{ftv}(t_3) \setminus \{y\}) \\
\mathrm{ftv}(\mu x : \tau.t) & = & \mathrm{ftv}(t) \setminus \{x\} \\
\mathrm{ftv}(\mathrm{seq}\ t_1\ t_2) & = & \mathrm{ftv}(t_1) \cup \mathrm{ftv}(t_2) \\
\mathrm{ftv}(\mathrm{ann}_\ell(t)) & = & \mathrm{ftv}(t)
\end{array}
$$

2.1.2. Underlying Type System

The type system of the source language is shown in figure 2.2. It consists of judgments of the form $\Gamma \vdash_{\mathrm{t}} t : \tau$, expressing that term t has type τ under the environment Γ.

The following definition gives a formal description of environments and introduces the notation we use when working with them.

Definition 2.3 (type environment)**.** A *type environment* or *type context* Γ is a finite list of bindings from term variables to types.

1. The empty context is written as [] or $\emptyset$, and the context Γ extended with the binding of the variable x to the type τ is written $\Gamma, x : \tau$.

2. Let $\mathrm{dom}(\Gamma)$ denote the set of variables in the context, defined by

$$
\begin{array}{lcl}
\mathrm{dom}([]) & = & \emptyset \\
\mathrm{dom}(\Gamma, x : \tau) & = & \{x\} \cup \mathrm{dom}(\Gamma)
\end{array}
$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,[\text{U-Var}] \qquad \frac{}{\Gamma \vdash () : \text{unit}}\,[\text{U-Unit}] \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \text{ann}_\ell(t) : \tau}\,[\text{U-Ann}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}\,[\text{U-Pair}] \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{proj}_i(t) : \tau_i}\,[\text{U-Proj}]$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \text{inl}_{\tau_2}(t) : \tau_1 + \tau_2}\,[\text{U-Inl}] \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \text{inr}_{\tau_1}(t) : \tau_1 + \tau_2}\,[\text{U-Inr}]$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash t_1 : \tau \qquad \Gamma, y : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ \{\text{inl}(x) \to t_1; \text{inr}(y) \to t_2\} : \tau}\,[\text{U-Case}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\,[\text{U-App}] \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2}\,[\text{U-Abs}]$$

$$\frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \mu x : \tau.t : \tau}\,[\text{U-Fix}] \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{seq}\ t_1\ t_2 : \tau_2}\,[\text{U-Seq}]$$

Figure 2.2.: Underlying type system ($\Gamma \vdash_{\text{t}} t : \tau$)

3. A statement $\Gamma(x) = \tau$ shall mean that $x \in \text{dom}(\Gamma)$ and the rightmost occurrence of x binds it to τ. If $x \in \text{dom}(\Gamma)$, let $\Gamma(x)$ denote the type τ such that $\Gamma(x) = \tau$.

4. $\Gamma \setminus X$ where $X \subseteq \textbf{Var}$ denotes the context Γ where all bindings of variables in X have been removed.

5. We define the index $\text{idx}(x, \Gamma)$ of a variable in Γ as the number of bindings we need to skip in order to reach it.

$$\begin{aligned} \text{idx}(x, \Gamma, y : \tau) &= \begin{cases} 0 & \text{if } x = y \\ 1 + \text{idx}(x, \Gamma) & \text{if } x \neq y \end{cases} \\ \text{idx}(x, []) &= \infty \end{aligned}$$

   Note that whenever $x \in \text{dom}(\Gamma)$, $\text{idx}(x, \Gamma) \neq \infty$.

6. The image of the environment, denoted $\text{im}(\Gamma)$, is the set of types that are bound by the environment, i.e.

$$\begin{aligned} \text{im}([]) &= \emptyset \\ \text{im}(\Gamma, x : \tau) &= \{\tau\} \cup \text{im}(\Gamma) \end{aligned}$$

The underlying type system consists of the standard rules, with the notable exception of [U-Ann] for the annotation construct $\text{ann}_\ell(t)$. Such an explicitly annotated term has simply the same underlying type as the term itself. The annotation ℓ imposed on t only becomes relevant in the annotated type system.

Sorts:

$$
\begin{array}{llll}
\kappa \in \textbf{AnnSort} & ::= & \star & \text{(base sort)} \\
 & | & \kappa_1 \Rightarrow \kappa_2 & \text{(function sort)}
\end{array}
$$

Annotations:

$$
\begin{array}{llll}
\ell \in \mathcal{L} & & & \text{(underlying lattice)} \\
\beta \in \textbf{AnnVar} & & & \text{(annotation variables)} \\
\xi \in \textbf{AnnTm} & ::= & \beta & \text{(variable)} \\
 & | & \lambda\beta :: \kappa.\xi & \text{(abstraction)} \\
 & | & \xi_1\ \xi_2 & \text{(application)} \\
 & | & \ell & \text{(lattice value)} \\
 & | & \xi_1 \sqcup \xi_2 & \text{(lattice join operation)}
\end{array}
$$

Figure 2.3.: $\lambda^{\sqcup}$-calculus: syntax

2.2. $\lambda^{\sqcup}$-calculus

In this section we define a simply-typed lambda-calculus that will later be used for representing annotations in the type and effect system. It is inspired by Koot's set based $\lambda^{\cup}$-calculus [15]. However, we will generalize it to arbitrary bounded join-semilattices[1] as this is what our analysis requires annotations to be. Hence, we dub our variant the $\lambda^{\sqcup}$-calculus.

2.2.1. Syntax

The syntax of the types and terms of the $\lambda^{\sqcup}$-calculus is shown in figure 2.3. In order to avoid confusion, from now on we will refer to the types of the $\lambda^{\sqcup}$-calculus exclusively by *sorts*. Furthermore, the letter κ denotes sorts, β denotes variables and ξ denotes terms in the $\lambda^{\sqcup}$-calculus. We will also refer to terms of the $\lambda^{\sqcup}$-calculus as *dependency terms*.

There are only two sorts, the base sort $\star$ representing values in the underlying lattice $\mathcal{L}$ and the function sort $\kappa_1 \Rightarrow \kappa_2$.

On the term level, we allow arbitrary elements of the underlying lattice and a binary join operator in addition to the usual variables, function applications and lambda abstractions.

Definition 2.4. Let ξ be a $\lambda^{\sqcup}$-term. We denote the set of free annotation variables

[1] Actually, the $\lambda^{\sqcup}$-calculus does not require the lattice to be bounded. The dependency analysis in its current formulation does, however.

$$\frac{\Sigma(\beta) = \kappa}{\Sigma \vdash_s \beta : \kappa}\text{[S-VAR]} \quad \frac{\ell \in \mathcal{L}}{\Sigma \vdash_s \ell : \star}\text{[S-LAT]} \quad \frac{\Sigma \vdash_s \xi_1 : \kappa \qquad \Sigma \vdash_s \xi_2 : \kappa}{\Sigma \vdash_s \xi_1 \sqcup \xi_2 : \kappa}\text{[S-JOIN]}$$

$$\frac{\Sigma, \beta : \kappa_1 \vdash_s \xi : \kappa_2}{\Sigma \vdash_s \lambda\beta :: \kappa_1.\xi : \kappa_1 \Rightarrow \kappa_2}\text{[S-ABS]} \quad \frac{\Sigma \vdash_s \xi_1 : \kappa_1 \Rightarrow \kappa_2 \qquad \Sigma \vdash_s \xi_2 : \kappa_1}{\Sigma \vdash_s \xi_1\ \xi_2 : \kappa_2}\text{[S-APP]}$$

Figure 2.4.: $\lambda^{\sqcup}$-calculus: sorting rules ($\Sigma \vdash_s \xi : \kappa$)

occurring in ξ by $\text{fav}(\xi)$, recursively defined by

$$\begin{array}{lcl}
\text{fav}(\beta) & = & \{\beta\} \\
\text{fav}(\lambda\beta :: \kappa.\xi) & = & \text{fav}(\xi) \setminus \{\beta\} \\
\text{fav}(\xi_1\ \xi_2) & = & \text{fav}(\xi_1) \cup \text{fav}(\xi_2) \\
\text{fav}(\ell) & = & \emptyset \\
\text{fav}(\xi_1 \sqcup \xi_2) = & = & \text{fav}(\xi_1) \cup \text{fav}(\xi_2).
\end{array}$$

2.2.2. Sorting

Figure 2.4 shows the sorting rules of the $\lambda^{\sqcup}$-calculus. For variables, functions and applications, these are just the standard rules known from the simply-typed lambda calculus. Values of the underlying lattice are always of sort $\star$. The join operator, however, is defined on arbitrary terms of the same sorts. This will be justified semantically in the following section.

Analogous to type environments in the underlying type system (see definition 2.3), the sorting rules use *sort environments* denoted by the letter Σ. Instead of mapping term variables to types, they map annotation variables β to sorts κ. We denote the set of sort environments by **SortEnv**.

Additionally, the well-sorted judgment provides some insight into the free variables of dependency terms.

Lemma 2.5. *If we have* $\Sigma \vdash_s \xi : \kappa$*, then* $\text{fav}(\xi) \subseteq \text{dom}(\Sigma)$.

Proof. By induction on $\Sigma \vdash_s \xi : \kappa$.

[S-VAR] We have $\xi = \beta$ for some variable β and $\Sigma(\beta) = \kappa$. Hence, $\beta \in \text{dom}(\Sigma)$ and $\text{fav}(\xi) = \{\beta\} \subseteq \text{dom}(\Sigma)$.

[S-ABS] We have $\xi = \lambda\beta_1 :: \kappa_1.\xi'$ and $\kappa = \kappa_1 \Rightarrow \kappa_2$ for some β_1, κ_1, κ_2 and ξ' such that $\Sigma, \beta_1 :: \kappa_1 \vdash_s \xi' : \kappa_2$ holds. By induction, $\text{fav}(\xi') \subseteq \text{dom}(\Sigma, \beta_1 :: \kappa_1)$. But then, $\text{fav}(\xi) = \text{fav}(\xi') \setminus \{\beta_1\} \subseteq \text{dom}(\Sigma)$.

[S-APP] We have $\xi = \xi_1\ \xi_2$ such that $\Sigma \vdash_s \xi_1 : \kappa_2 \Rightarrow \kappa$ and $\Sigma \vdash_s \xi_2 : \kappa_2$ hold for some κ_2. By induction, $\text{fav}(\xi_1) \subseteq \text{dom}(\Sigma)$ and $\text{fav}(\xi_2) \subseteq \text{dom}(\Sigma)$. Thus, $\text{fav}(\xi) = \text{fav}(\xi_1) \cup \text{fav}(\xi_2) \subseteq \text{dom}(\Sigma)$.

[S-JOIN] Analogous to the previous case.

[S-Lat] We have $\xi = \ell$ for some $\ell \in \mathcal{L}$. Hence, $\text{fav}(\xi) = \emptyset \subseteq \text{dom}(\Sigma)$.

□

The following lemma allows us to modify unrelated bindings in the sort environment without harming well-sortedness.

Lemma 2.6. *Let ξ be a $\lambda^{\sqcup}$-term of sort κ and let Σ denote a sort environment such that $\Sigma \vdash_{\text{s}} \xi : \kappa$. Then, for all Σ' such that for all $\beta \in \text{fav}(\xi)$ we have $\Sigma(\beta) = \Sigma'(\beta)$, $\Sigma' \vdash_{\text{s}} \xi : \kappa$ also holds.*

Proof. By induction on the derivation tree of $\Sigma \vdash_{\text{s}} \xi : \kappa$.

[S-Var] By definition of the rule, $\xi = \beta'$ for some variable β'. The premise is $\Sigma(\beta') = \kappa$. By assumption $\Sigma'(\beta') = \Sigma(\beta')$ since $\beta' \in \text{fav}(\beta')$. Therefore, $\Sigma' \vdash_{\text{s}} \beta' : \kappa$ holds by [S-Var].

[S-Abs] By definition of the rule, $\xi = \lambda\beta' :: \kappa_1.\xi'$ for some β', κ_1 and ξ' and further $\kappa = \kappa_1 \Rightarrow \kappa_2$ for some κ_2. The premise is $\Sigma, \beta' :: \kappa_1 \vdash_{\text{s}} \xi' : \kappa_2$.

By definition, we have $\text{fav}(\xi) = \text{fav}(\xi') \setminus \{\beta'\}$. Therefore, $(\Sigma', \beta' :: \kappa_1)(\beta) = (\Sigma, \beta' :: \kappa_1)(\beta)$ for all $\beta \in \text{fav}(\xi')$. This means we can apply the induction hypothesis in order to get $\Sigma', \beta' :: \kappa_1 \vdash_{\text{s}} \xi' : \kappa_2$. By [S-Abs], we have $\Sigma \vdash_{\text{s}} \xi : \kappa$.

[S-App] By definition of the rule, $\xi = \xi_1\ \xi_2$. The premises are $\Sigma \vdash_{\text{s}} \xi_1 : \kappa' \Rightarrow \kappa$ and $\Sigma \vdash_{\text{s}} \xi_2 : \kappa'$ for some κ'.

Since $\text{fav}(\xi_1\ \xi_2) = \text{fav}(\xi_1) \cup \text{fav}(\xi_2)$, we have in particular that $\Sigma(\beta) = \Sigma'(\beta)$ for all $\beta \in \text{fav}(\xi_1)$. Hence we can apply the induction hypothesis and get $\Sigma' \vdash_{\text{s}} \xi_1 : \kappa' \Rightarrow \kappa$. Analogously, we can conclude $\Sigma' \vdash_{\text{s}} \xi_2 : \kappa'$. Lastly, $\Sigma' \vdash_{\text{s}} \xi_1\ \xi_2 : \kappa$ holds by [S-App].

[S-Join] Can be proven analogously to the previous case by propagating the environments.

[S-Lat] Trivial, because [S-Lat] can be used with any environment.

□

Lastly, we show that well-sorted substitutions preserve the well-sortedness of dependency terms. A substitution is a map from variables to terms usually denoted by the letter θ. The application of a substitution θ to a term ξ is written $\theta\xi$ and replaces all free variables in ξ that are also in the domain of ξ with the corresponding terms they are mapped to. A concrete substitution replacing the variables $\beta_1, \ldots, \beta_n$ with terms $\xi_1, \ldots, \xi_n$ is written $[\xi_1/\beta_1, \ldots, \xi_n/\beta_n]$.

Lemma 2.7. *Let ξ be a dependency term, let κ be a sort, let Σ, Σ' be sort environments such that $\Sigma, \Sigma' \vdash_{\text{s}} \xi : \kappa$ and let θ be a substitution such that $\Sigma \vdash_{\text{s}} \theta(\beta) : \Sigma'(\beta)$ for all $\beta \in \text{dom}(\theta) = \text{dom}(\Sigma')$. Then $\Sigma \vdash_{\text{s}} \theta\xi : \kappa$.*

Proof. By induction on the derivation of $\Sigma \vdash_{\text{s}} \xi : \kappa$.

[S-VAR] We have $\xi = \beta$ and $(\Sigma, \Sigma')(\beta) = \kappa$. If $\beta \in \mathrm{dom}(\theta) = \mathrm{dom}(\Sigma')$, we have $\Sigma \vdash_{\mathrm{s}} \theta\xi : \Sigma'(\beta)$ by assumption. If $\beta \notin \mathrm{dom}(\theta) = \mathrm{dom}(\Sigma')$, $\theta\xi = \xi$, and therefore $\Sigma \vdash_{\mathrm{s}} \theta\xi : \kappa$ by [S-VAR].

[S-ABS] We have $\xi = \lambda\beta :: \kappa_1.\xi'$, $\kappa = \kappa_1 \Rightarrow \kappa_2$ and the premise is $\Sigma, \Sigma', \beta : \kappa_1 \vdash_{\mathrm{s}} \xi' : \kappa_2$. We can assume without loss of generality that β is distinct from any variables in the contexts Σ and Σ', simply by renaming them. Then, the above statement is equivalent to $\Sigma, \beta : \kappa_1, \Sigma' \vdash_{\mathrm{s}} \xi' : \kappa_2$ by lemma 2.6 and $\theta(\lambda\beta :: \kappa_1.\xi') = \lambda\beta :: \kappa_1.\theta\xi'$.

Also by lemma 2.6, $\Sigma, \beta : \kappa_1 \vdash_{\mathrm{s}} \theta(\beta) : \Sigma'(\beta)$ for all $\beta \in \mathrm{dom}(\theta)$. By induction, $\Sigma, \beta : \kappa_1 \vdash_{\mathrm{s}} \theta\xi' : \kappa_2$. Hence, $\Sigma' \vdash_{\mathrm{s}} \theta\xi : \kappa$ by [S-ABS].

The remaining cases simply propagate the substitution. □

2.2.3. Denotational Semantics

In order to better characterize the functionality of the $\lambda^{\sqcup}$-calculus and to lay the foundations for the meta-theory of our analysis, we define the denotational semantics for the $\lambda^{\sqcup}$-calculus.

First of all, we need an auxiliary definition providing us with the pointwise extension of a lattice into a function space.

Definition 2.8 (pointwise extension)**.** Given a join-semilattice L and some set X we define the join-semilattice $X \to L$ to be the pointwise extension of L, i.e.

$$\begin{array}{rcll} \bot & = & \lambda x \in X.\bot & \text{(pointwise bottom)} \\ f \sqcup g & = & \lambda x \in X.f(x) \sqcup g(x) & \text{(pointwise join)} \end{array}$$

It follows that the induced binary relation $\sqsubseteq$ on $X \to L$ is exactly the pointwise extension of the underlying comparison, i.e.

$$f \sqsubseteq g \iff \forall x \in X.f(x) \sqsubseteq g(x).$$

Furthermore, we need the following property when defining the denotational semantics.

Definition 2.9. Let L_1, L_2 be bounded join-semilattices and $f : L_1 \to L_2$ a function between lattices. We say that f is *monotone* or *order-preserving* if for all $x, y \in L_1$, $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$.

Figure 2.5 shows the denotational semantics of the $\lambda^{\sqcup}$-calculus. The universe V_κ denotes the lattice that is represented by the sort κ. The base sort $\star$ represents the underlying lattice $\mathcal{L}$ and the function sort $\kappa_1 \Rightarrow \kappa_2$ represents the lattice constructed by pointwise extension of the lattice V_{κ_2} restricted to monotone functions.

The denotation function $[\![\cdot]\!]_\rho$ is parameterized with an environment ρ that provides the values of variables. The denotation of a lambda term is simply an element of the corresponding function space. Applications are therefore mapped directly to the underlying function application of the meta-theory.

Universes:

$$\begin{array}{rcl} V_\star & = & \mathcal{L} \\ V_{\kappa_1 \Rightarrow \kappa_2} & = & \{f : V_{\kappa_1} \to V_{\kappa_2} \mid f \text{ monotone}\} \end{array}$$

Environment:

$$\rho \quad : \quad \textbf{AnnVar} \to_{\text{fin}} \bigcup \{V_\kappa \mid \kappa \in \textbf{AnnSort}\}$$

Semantics:

$$\begin{array}{rcl} [\![\beta]\!]_\rho & = & \rho(\beta) \\ [\![\lambda\beta :: \kappa_1.\xi]\!]_\rho & = & \lambda v \in V_{\kappa_1}.\, [\![\xi]\!]_{\rho[\beta \mapsto v]} \\ [\![\xi_1\ \xi_2]\!]_\rho & = & [\![\xi_1]\!]_\rho\, ([\![\xi_2]\!]_\rho) \\ [\![\ell]\!]_\rho & = & \ell \\ [\![\xi_1 \sqcup \xi_2]\!]_\rho & = & [\![\xi_1]\!]_\rho \sqcup [\![\xi_2]\!]_\rho \end{array}$$

Figure 2.5.: $\lambda^{\sqcup}$-calculus: denotational semantics

This is unlike the $\lambda^{\cup}$-calculus by Koot where lambda terms are mapped to singleton sets of functions and function application is defined in terms of the union of the results of individually applying each function. The crucial difference is that we have offloaded this complexity into the definition of the pointwise extension of lattices. It is therefore important to note that the join operator used in the denotation of a term $\xi_1 \sqcup \xi_2$ depends on the sort κ of this term and belongs to the lattice V_κ.

We will now prove that this definition of the denotational semantics is well-defined when applied to well-typed terms, but first we need the following definition.

Definition 2.10 (compatible environments)**.** We say a sort environment Σ and an environment $\rho : \textbf{AnnVar} \to_{\text{fin}} \bigcup \{V_\kappa \mid \kappa \in \textbf{AnnSort}\}$ used in the denotational semantics are *compatible* if

1. they are defined on the same variables, i.e. $\text{dom}(\Sigma) = \text{dom}(\rho)$ and
2. for all $\beta \in \text{dom}(\Sigma)$ we have $\rho(\beta) \in V_{\Sigma(\beta)}$.

Given this notion of compatibility between the sort environment and the environment used in the denotational semantics, we can now establish the well-definedness of the semantics in the following lemma.

Lemma 2.11. *Given a term ξ of sort κ, a sort environment Σ and a* compatible *environment $\rho : \textbf{AnnVar} \to_{\text{fin}} \bigcup \{V_\kappa \mid \kappa \in \textbf{AnnSort}\}$ such that $\Sigma \vdash_{\text{s}} \xi : \kappa$ holds, then*

1. *$[\![\xi]\!]_{_}$ is monotone in the environment ρ, i.e. for all environments ρ_1 such that $\rho \sqsubseteq \rho_1$ (seen as a pointwise extension lattice), $[\![\xi]\!]_\rho \sqsubseteq [\![\xi]\!]_{\rho_1}$.*
2. *$[\![\xi]\!]_\rho$ is well-defined,*

3. $[\![\xi]\!]_\rho \in V_\kappa$.

Proof. See appendix, page 99. $\square$

Based on the denotational semantics, we can now define what it means for one dependency term to subsume another. This definition will become relevant for subtyping in order to widen effects where necessary.

Definition 2.12 (subsumption)**.** Given two dependency terms ξ_1 and ξ_2 and a sort κ such that $\Sigma \vdash_\mathrm{s} \xi_1 : \kappa$ and $\Sigma \vdash_\mathrm{s} \xi_2 : \kappa$, we say that ξ_2 *subsumes* ξ_1 under the environment Σ, written $\Sigma \vdash_\mathrm{sub} \xi_1 \sqsubseteq \xi_2$, if for all environments ρ compatible with Σ, we have $[\![\xi_1]\!]_\rho \sqsubseteq [\![\xi_2]\!]_\rho$.

Remark 2.13*.* It follows directly from the reflexivity, transitivity and antisymmetry of the partial order $\sqsubseteq$ that the subsumption relation $\Sigma \vdash_\mathrm{sub} \xi_1 \sqsubseteq \xi_2$ is also reflexive, transitive and antisymmetric for a fixed environment Σ.

In a similar way we can define what it means for two terms to be semantically equal.

Definition 2.14. Given two dependency terms ξ_1 and ξ_2 and a sort κ such that $\Sigma \vdash_\mathrm{s} \xi_1 : \kappa$ and $\Sigma \vdash_\mathrm{s} \xi_2 : \kappa$, we say that ξ_1 and ξ_2 are *semantically equal* under the environment Σ, written $\Sigma \vdash \xi_1 \equiv \xi_2$, if for all environments ρ compatible with Σ, we have $[\![\xi_1]\!]_\rho = [\![\xi_2]\!]_\rho$.

Remark 2.15*.* It is obvious that two terms are semantically equal exactly when each subsumes the other due to the antisymmetry of the order $\sqsubseteq$ in a lattice.

The following lemma shows that computing the denotation of a dependency term after a substitution can always be replaced by simply choosing an appropriate environment when evaluating the original term.

Lemma 2.16. *For all sort environments Σ, Σ' with disjoint domains, environments ρ compatible with Σ, dependency terms ξ and substitutions $\theta : \mathrm{dom}(\Sigma') \to$ **AnnTm** such that $\Sigma, \Sigma' \vdash_\mathrm{s} \xi : \kappa$ and $\Sigma \vdash_\mathrm{s} \theta(\beta') : \Sigma'(\beta')$ for all $\beta' \in \mathrm{dom}(\theta)$, we have $[\![\theta\xi]\!]_\rho = [\![\xi]\!]_{\rho'}$ for the environment ρ' compatible with Σ, Σ' given by*

$$\rho'(\beta) = \begin{cases} \rho(\beta) & \text{if } \beta \in \mathrm{dom}(\Sigma) \\ [\![\theta(\beta)]\!]_\rho & \text{otherwise} \end{cases}$$

Proof. By induction on ξ.

$\xi = \beta$ By definition of ρ', $[\![\theta\beta]\!]_\rho = \rho'(\beta) = [\![\beta]\!]_{\rho'}$.

$\xi = \lambda\beta :: \kappa_1.\xi'$ We have

$$[\![\theta\lambda\beta :: \kappa_1.\xi']\!]_\rho = \lambda v \in V_{\kappa_1}.\, [\![\theta\xi']\!]_{\rho[\beta\mapsto v]} = \lambda v \in V_{\kappa_1}.\, [\![\xi']\!]_{\rho''} = [\![\lambda\beta :: \kappa_1.\xi']\!]_{\rho'}$$

The first and third step use the definition of the denotational semantics while the second step follows by induction, giving us a suitable ρ''. We denote by $\rho \restriction_X$ the restriction of the environment to the variables in a set X. Then, we have $\rho' = \rho'' \restriction_{\mathrm{dom}(\rho'')\setminus\{\beta\}}$. Without loss of generality we assume that the variable β bound in the abstraction is fresh.

$\xi = \xi_1\ \xi_2$ We apply the induction hypothesis to both subterms. Since we have the same sort environment and substitution in both cases, the environment ρ' is also the same. We get

$$[\![\theta(\xi_1\ \xi_2)]\!]_\rho = [\![\theta\xi_1]\!]_\rho\,([\![\theta\xi_2]\!]_\rho) = [\![\xi_1]\!]_{\rho'}\,([\![\xi_2]\!]_{\rho'}) = [\![\xi_1\ \xi_2]\!]_{\rho'}$$

$\xi = \xi_1 \sqcup \xi_2$ Similar to the previous case.

$\xi = \ell$ Trivial, as the denotation of a constant is independent of the environment.

It remains to show that the environment ρ' is compatible with Σ, Σ'. Let $\beta \in \mathrm{dom}(\rho')$ be arbitrary. If $\beta \in \mathrm{dom}(\Sigma)$, we have $\rho'(\beta) = \rho(\beta) \in V_{\Sigma(\beta)}$ because ρ is compatible with Σ. If $\beta \in \mathrm{dom}(\Sigma')$, we have $\rho'(\beta) = [\![\theta(\beta)]\!]_\rho \in V_{\Sigma'(\beta)}$ because $\Sigma \vdash_{\mathrm{s}} \theta(\beta) : \Sigma'(\beta)$ and by lemma 2.11. □

We will also need the following fact that subsumption is preserved by certain substitutions.

Lemma 2.17. *Let Σ and Σ' be sort environments with disjoint domains and let θ_1 : $\mathrm{dom}(\Sigma') \to$ **AnnTm** and $\theta_2 : \mathrm{dom}(\Sigma') \to$ **AnnTm** be substitutions such that $\Sigma \vdash_{\mathrm{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$, $\Sigma \vdash_{\mathrm{s}} \theta_1(\beta) : \Sigma'(\beta)$ and $\Sigma \vdash_{\mathrm{s}} \theta_2(\beta) : \Sigma'(\beta)$ for all $\beta \in \mathrm{dom}(\Sigma')$.*

Let ξ_1 and ξ_2 be dependency terms such that $\Sigma, \Sigma' \vdash_{\mathrm{s}} \xi_1 : \kappa$ and $\Sigma, \Sigma' \vdash_{\mathrm{s}} \xi_2 : \kappa$ for some sort κ.

If we have $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$ we also have $\Sigma \vdash_{\mathrm{sub}} \theta_1\xi_1 \sqsubseteq \theta_2\xi_2$.

Proof. We have to show that $[\![\theta_1\xi_1]\!]_\rho \sqsubseteq [\![\theta_2\xi_2]\!]_\rho$ for all environments ρ compatible with Σ.

Applying lemma 2.16 to ξ_1 and ξ_2 we get two environments ρ_1 and ρ_2 such that $[\![\theta_1\xi_1]\!]_\rho = [\![\xi_1]\!]_{\rho_1}$ and $[\![\theta_2\xi_2]\!]_\rho = [\![\xi_2]\!]_{\rho_2}$.

Moreover, we know $\rho_1(\beta) \sqsubseteq \rho_2(\beta)$ for all $\beta \in \mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$. In order to see why, let $\beta \in \mathrm{dom}(\rho_1)$ be arbitrary. If $\beta \in \mathrm{dom}(\Sigma)$, we have $\rho_1(\beta) = \rho(\beta) = \rho_2(\beta)$. If $\beta \in \mathrm{dom}(\Sigma')$, we have $\rho_1(\beta) = [\![\theta_1(\beta)]\!]_\rho \sqsubseteq [\![\theta_2(\beta)]\!]_\rho = \rho_2(\beta)$ due to the fact that $\Sigma \vdash_{\mathrm{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$.

Now we can derive $[\![\theta_1\xi_1]\!]_\rho = [\![\xi_1]\!]_{\rho_1} \sqsubseteq [\![\xi_1]\!]_{\rho_2} \sqsubseteq [\![\xi_2]\!]_{\rho_2} = [\![\theta_2\xi_2]\!]_\rho$. The first and last step simply use lemma 2.16. The second step uses the fact that the denotation function is monotone in the environment parameter (by lemma 2.11). The third step holds by $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$. □

We can omit variables that do not occur free in a term from the environment when computing its denotation.

Lemma 2.18. *Let ξ be a dependency term, and let Σ be a context such that $\Sigma \vdash_{\mathrm{s}} \xi : \kappa$ for some κ. Let ρ be an environment compatible with Σ. Let $X \supseteq \mathrm{fav}(\xi)$ be a set of variables.*

Then, $[\![\xi]\!]_\rho = [\![\xi]\!]_{\rho\restriction_X}$.

Proof. By induction on ξ.

$\xi = \beta$ We have $X \supseteq \text{fav}(\beta) = \{\beta\}$. Clearly, $[\![\beta]\!]_\rho = \rho(\beta) = \rho\restriction_X(\beta) = [\![\beta]\!]_{\rho\restriction_X}$.

$\xi = \lambda\beta :: \kappa.\xi'$ Using the induction hypothesis, we can show

$$\begin{aligned}[\![\lambda\beta :: \kappa.\xi']\!]_\rho &= \lambda v \in V_\kappa.\ [\![\xi']\!]_{\rho[\beta\mapsto v]} = \lambda v \in V_\kappa.\ [\![\xi']\!]_{\rho[\beta\mapsto v]\restriction_{X\cup\{\beta\}}} \\ &= \lambda v \in V_\kappa.\ [\![\xi']\!]_{\rho\restriction_X[\beta\mapsto v]} = [\![\lambda\beta :: \kappa.\xi']\!]_{\rho\restriction_X}.\end{aligned}$$

$\xi = \xi_1\ \xi_2$ Since $X \supseteq \text{fav}(\xi_1\ \xi_2) = \text{fav}(\xi_1) \cup \text{fav}(\xi_2)$, we have

$$[\![\xi_1\ \xi_2]\!]_\rho = [\![\xi_1]\!]_\rho\,([\![\xi_2]\!]_\rho) = [\![\xi_1]\!]_{\rho\restriction_X}\,([\![\xi_2]\!]_{\rho\restriction_X}) = [\![\xi_1\ \xi_2]\!]_{\rho\restriction_X}.$$

$\xi = \xi_1 \sqcup \xi_2$ Since $X \supseteq \text{fav}(\xi_1 \sqcup \xi_2) = \text{fav}(\xi_1) \cup \text{fav}(\xi_2)$, we have

$$[\![\xi_1 \sqcup \xi_2]\!]_\rho = [\![\xi_1]\!]_\rho \sqcup [\![\xi_2]\!]_\rho = [\![\xi_1]\!]_{\rho\restriction_X} \sqcup [\![\xi_2]\!]_{\rho\restriction_X} = [\![\xi_1 \sqcup \xi_2]\!]_{\rho\restriction_X}.$$

$\xi = \ell$ Trivial, as $[\![\ell]\!]_\rho = \ell$ regardless of the environment.

□

Additionally, it is possible to change the context of subsumption statements under certain conditions.

Lemma 2.19. *Let ξ_1 and ξ_2 be dependency terms and Σ a sort environment such that $\Sigma \vdash_s \xi_1 : \kappa$, $\Sigma \vdash_s \xi_2 : \kappa$ and $\Sigma \vdash_{sub} \xi_1 \sqsubseteq \xi_2$. Then, $\Sigma' \vdash_{sub} \xi_1 \sqsubseteq \xi_2$ holds for all Σ' such that for all $\beta \in \text{fav}(\xi_1) \cup \text{fav}(\xi_2)$ we have $\Sigma(\beta) = \Sigma'(\beta)$.*

Proof. See appendix, page 100. □

2.3. Target Language

We can now take the next step towards the declarative type and effect system by defining the target language of our analysis. It extends the source language introduced earlier (see figure 2.1) with dependency annotations.

2.3.1. Terms and Types

The syntax of the target language is shown in figure 2.6. *Annotated types* of the target language are denoted by $\widehat{\tau}$ and *annotated terms* are denoted by $\widehat{t}$. In order to avoid syntactic noise, we sometimes denote annotated terms just by t when there is no ambiguity.

On the type level, there is an additional construct $\forall\beta :: \kappa.\widehat{\tau}$ quantifying over an annotation variable β of sort κ. Furthermore, the recursive occurrences in the sum, product and arrow types now each carry an annotation.

On the term level, the explicit type annotations of lambda expressions and fixpoints are now annotated types and also include a dependency annotation. Moreover, dependency abstraction and application have been added to reflect the quantification of dependency variables on the type level.

Annotated Types:

$$
\begin{array}{llll}
\widehat{\tau} \in \widehat{\mathbf{Ty}} & ::= & \forall \beta :: \kappa . \widehat{\tau} & \text{(annotation quantification)} \\
 & | & \widehat{\text{unit}} & \text{(unit type)} \\
 & | & \widehat{\tau}_1 \langle \xi_1 \rangle + \widehat{\tau}_2 \langle \xi_2 \rangle & \text{(sum type)} \\
 & | & \widehat{\tau}_1 \langle \xi_1 \rangle \times \widehat{\tau}_2 \langle \xi_2 \rangle & \text{(product type)} \\
 & | & \widehat{\tau}_1 \langle \xi_1 \rangle \to \widehat{\tau}_2 \langle \xi_2 \rangle & \text{(function type)}
\end{array}
$$

Terms

$$
\begin{array}{llll}
\widehat{t} \in \widehat{\mathbf{Tm}} & ::= & \cdots & \\
 & | & \lambda x : \widehat{\tau} \,\&\, \xi . \widehat{t} & \text{(abstraction)} \\
 & | & \mu x : \widehat{\tau} \,\&\, \xi . \widehat{t} & \text{(fixpoint)} \\
 & | & \cdots & \\
 & | & \Lambda \beta :: \kappa . \widehat{t} & \text{(dependency abstraction)} \\
 & | & \widehat{t} \, \langle \xi \rangle & \text{(dependency application)}
\end{array}
$$

Figure 2.6.: Target language: syntax

Definition 2.20. We denote the set of free (term) variables in a target term $\widehat{t}$ by $\text{ftv}(\widehat{t})$.

The actual definition of $\text{ftv}(\widehat{t})$ is completely analogous to what we have done for source terms.

Figure 2.7 shows the rules for well-formed annotated types that are used by the following definition.

Definition 2.21. An annotated type $\widehat{\tau}$ is *well-formed* under a sort environment Σ if $\Sigma \vdash_{\text{wft}} \widehat{\tau}$ holds.

Informally, a type is well-formed only if all annotations are of sort $\star$ and all annotation variables that are used have previously been bound.

As the following lemma demonstrates, we can modify the environment in a well-formedness judgment in a similar way to what we have shown for sorting judgments in lemma 2.6.

Lemma 2.22. *Let $\widehat{\tau}$ be an annotated type and Σ a sort environment such that $\Sigma \vdash_{\text{wft}} \widehat{\tau}$. Then, for all Σ' such that for all $\beta \in \text{fav}(\widehat{\tau})$ we have $\Sigma(\beta) = \Sigma'(\beta)$, $\Sigma' \vdash_{\text{wft}} \widehat{\tau}$ holds.*

Proof. This follows directly by induction using lemma 2.6 on the well-sortedness judgments. □

Additionally, we can define the underlying terms and types that correspond to annotated terms and types.

Definition 2.23. Let $\widehat{\tau}$ be an annotated type. We denote by $\lfloor\widehat{\tau}\rfloor$ the erasure of annotations, mapping each annotated type to its corresponding underlying type. It is recursively defined as

$$\begin{array}{rcl} \lfloor\forall\beta :: \kappa.\widehat{\tau}\rfloor & = & \lfloor\widehat{\tau}\rfloor \\ \lfloor\widehat{\text{unit}}\rfloor & = & \text{unit} \\ \lfloor\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle\rfloor & = & \lfloor\widehat{\tau}_1\rfloor + \lfloor\widehat{\tau}_2\rfloor \\ \lfloor\widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle\rfloor & = & \lfloor\widehat{\tau}_1\rfloor \times \lfloor\widehat{\tau}_2\rfloor \\ \lfloor\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle\rfloor & = & \lfloor\widehat{\tau}_1\rfloor \to \lfloor\widehat{\tau}_2\rfloor \end{array}$$

Moreover, we define erasure from target terms to source terms. Let $\widehat{t}$ be an annotated term. We denote by $\lfloor\widehat{t}\rfloor$ the corresponding source term defined as follows.

$$\begin{array}{rcl} \lfloor\lambda x : \widehat{\tau}\ \&\ \xi.\widehat{t'}\rfloor & = & \lambda x : \lfloor\widehat{\tau}\rfloor.\lfloor\widehat{t'}\rfloor \\ \lfloor\mu x : \widehat{\tau}\ \&\ \xi.\widehat{t'}\rfloor & = & \mu x : \lfloor\widehat{\tau}\rfloor.\lfloor\widehat{t'}\rfloor \\ \lfloor\Lambda\beta :: \kappa.\widehat{t'}\rfloor & = & \lfloor\widehat{t'}\rfloor \\ \lfloor\widehat{t'}\ \langle\xi\rangle\rfloor & = & \lfloor\widehat{t'}\rfloor \\ & \vdots & \end{array}$$

The remaining cases are defined by recursively erasing the sub-terms.

We also extend the definition of free annotation variables to annotated types as follows.

Definition 2.24. Let $\widehat{\tau}$ be an annotated type. The set of free annotation variables of $\widehat{\tau}$ is recursively defined by:

$$\begin{array}{lcl} \text{fav}(\forall\beta :: \kappa.\widehat{\tau}) & = & \text{fav}(\widehat{\tau}) \setminus \{\beta\} \\ \text{fav}(\widehat{\text{unit}}) & = & \emptyset \\ \text{fav}(\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle) & = & \text{fav}(\widehat{\tau}_1) \cup \text{fav}(\xi_1) \cup \text{fav}(\widehat{\tau}_2) \cup \text{fav}(\xi_2) \\ \text{fav}(\widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle) & = & \text{fav}(\widehat{\tau}_1) \cup \text{fav}(\xi_1) \cup \text{fav}(\widehat{\tau}_2) \cup \text{fav}(\xi_2) \\ \text{fav}(\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle) & = & \text{fav}(\widehat{\tau}_1) \cup \text{fav}(\xi_1) \cup \text{fav}(\widehat{\tau}_2) \cup \text{fav}(\xi_2) \end{array}$$

Furthermore, we define what it means for two annotated types to have the same shape.

Definition 2.25. We say two annotated types $\widehat{\tau}_1$ and $\widehat{\tau}_2$ *have the same shape* if

1. $\widehat{\tau}_1 = \widehat{\text{unit}}$ and $\widehat{\tau}_2 = \widehat{\text{unit}}$, or
2. $\widehat{\tau}_1 = \forall\beta\ \kappa\ \widehat{\tau}'_1$, $\widehat{\tau}_2 = \forall\beta\ \kappa\ \widehat{\tau}'_2$ and $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape, or
3. $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}''_1\langle\xi''_1\rangle$, $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle + \widehat{\tau}''_2\langle\xi''_2\rangle$ and $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape and $\widehat{\tau}''_1$ and $\widehat{\tau}''_2$ have the same shape, or
4. $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle \times \widehat{\tau}''_1\langle\xi''_1\rangle$, $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle \times \widehat{\tau}''_2\langle\xi''_2\rangle$ and $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape and $\widehat{\tau}''_1$ and $\widehat{\tau}''_2$ have the same shape, or
5. $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}''_1\langle\xi''_1\rangle$, $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle \to \widehat{\tau}''_2\langle\xi''_2\rangle$ and $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape and $\widehat{\tau}''_1$ and $\widehat{\tau}''_2$ have the same shape.

$$\frac{\Sigma, \beta :: \kappa \vdash_{\text{wft}} \widehat{\tau}}{\Sigma \vdash_{\text{wft}} \forall \beta :: \kappa . \widehat{\tau}} \text{[W-FORALL]} \qquad \frac{}{\Sigma \vdash_{\text{wft}} \widehat{\text{unit}}} \text{[W-UNIT]}$$

$$\frac{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1 \qquad \Sigma \vdash_{\text{s}} \xi_1 : \star \qquad \Sigma \vdash_{\text{wft}} \widehat{\tau}_2 \qquad \Sigma \vdash_{\text{s}} \xi_2 : \star}{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle} \text{[W-SUM]}$$

$$\frac{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1 \qquad \Sigma \vdash_{\text{s}} \xi_1 : \star \qquad \Sigma \vdash_{\text{wft}} \widehat{\tau}_2 \qquad \Sigma \vdash_{\text{s}} \xi_2 : \star}{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle} \text{[W-PROD]}$$

$$\frac{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1 \qquad \Sigma \vdash_{\text{s}} \xi_1 : \star \qquad \Sigma \vdash_{\text{wft}} \widehat{\tau}_2 \qquad \Sigma \vdash_{\text{s}} \xi_2 : \star}{\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle} \text{[W-ARR]}$$

Figure 2.7.: Well-formedness of annotated types ($\Sigma \vdash_{\text{wft}} \widehat{\tau}$)

Remark 2.26. We can view *having the same shape* as an equivalence relation. The conditions for reflexivity, symmetry and transitivity can be easily checked by induction.

Lemma 2.27. *If two target types $\widehat{\tau}_1$ and $\widehat{\tau}_2$ have the same shape, then $\lfloor\widehat{\tau}_1\rfloor = \lfloor\widehat{\tau}_2\rfloor$.*

Proof. By induction on $\widehat{\tau}_1$.

$\widehat{\tau}_1 = \widehat{\textbf{unit}}$ Then $\widehat{\tau}_2 = \widehat{\text{unit}}$ as well, and clearly $\lfloor\widehat{\tau}_1\rfloor = \lfloor\widehat{\tau}_2\rfloor$.

$\widehat{\tau}_1 = \forall\beta\ \kappa\ \widehat{\tau}_1'$ Then $\widehat{\tau}_2 = \forall\beta\ \kappa\ \widehat{\tau}_2'$ and $\widehat{\tau}_1'$ and $\widehat{\tau}_2'$ have the same shape. By induction, $\lfloor\widehat{\tau}_1'\rfloor = \lfloor\widehat{\tau}_2'\rfloor$. Therefore, $\lfloor\widehat{\tau}_1\rfloor = \lfloor\widehat{\tau}_2\rfloor$.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle$ Then $\widehat{\tau}_2 = \widehat{\tau}_2'\langle\xi_2'\rangle + \widehat{\tau}_2''\langle\xi_2''\rangle$ and $\widehat{\tau}_1'$ and $\widehat{\tau}_2'$ have the same shape, as well as $\widehat{\tau}_1''$ and $\widehat{\tau}_2''$. By induction, $\lfloor\widehat{\tau}_1'\rfloor = \lfloor\widehat{\tau}_2'\rfloor$ and $\lfloor\widehat{\tau}_1''\rfloor = \lfloor\widehat{\tau}_2''\rfloor$. Therefore, $\lfloor\widehat{\tau}_1\rfloor = \lfloor\widehat{\tau}_2\rfloor$.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle \times \widehat{\tau}_1''\langle\xi_1''\rangle$ Analogous to the previous case.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_1''\rangle$ Analogous to the previous case.

□

The following lemma shows that substitutions satisfying certain conditions preserve well-formedness of annotated types.

Lemma 2.28. *Let $\widehat{\tau}$ be an annotated type and let Σ, Σ' be sort environments such that $\Sigma, \Sigma' \vdash_{\text{wft}} \widehat{\tau}$ and let θ be a substitution such that $\Sigma \vdash_{\text{s}} \theta(\beta) : \Sigma'(\beta)$ for all $\beta \in \text{dom}(\theta) = \text{dom}(\Sigma')$. Then $\Sigma \vdash_{\text{wft}} \theta\widehat{\tau}$.*

Proof. By induction on the derivation tree of $\Sigma, \Sigma' \vdash_{\text{wft}} \theta$.

[W-FORALL] We have $\widehat{\tau} = \forall\beta :: \kappa . \widehat{\tau}'$ and the premise $\Sigma, \Sigma', \beta :: \kappa \vdash_{\text{wft}} \widehat{\tau}'$. Without loss of generality, we can assume that β is distinct from all variables in the environments Σ, Σ' by renaming. We can then reorder the context to $\Sigma, \beta :: \kappa, \Sigma' \vdash_{\text{wft}} \widehat{\tau}'$ by lemma 2.22 and $\theta\widehat{\tau} = \forall\beta :: \kappa . \theta\widehat{\tau}'$.

Also by lemma 2.22, $\Sigma, \beta :: \kappa \vdash_{\mathrm{s}} \theta(\beta') : \Sigma'(\beta')$ for all $\beta' \in \mathrm{dom}(\theta) = \mathrm{dom}(\Sigma')$. By induction, $\Sigma, \beta :: \kappa \vdash_{\mathrm{wft}} \theta\widehat{\tau}'$. Hence, $\Sigma \vdash_{\mathrm{wft}} \theta\widehat{\tau}$ by [W-FORALL].

[W-UNIT] Trivial, since $\theta\widehat{\mathrm{unit}} = \widehat{\mathrm{unit}}$.

[W-SUM] We have $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ such that $\Sigma, \Sigma' \vdash_{\mathrm{wft}} \widehat{\tau}_1$, $\Sigma, \Sigma' \vdash_{\mathrm{wft}} \widehat{\tau}_2$, $\Sigma, \Sigma' \vdash_{\mathrm{s}} \xi_1 : \star$ and $\Sigma, \Sigma' \vdash_{\mathrm{s}} \xi_2 : \star$ hold.

By lemma 2.7, $\Sigma \vdash_{\mathrm{s}} \theta\xi_1 : \star$ and $\Sigma \vdash_{\mathrm{s}} \theta\xi_2 : \star$. By induction, $\Sigma \vdash_{\mathrm{wft}} \theta\widehat{\tau}_1$ and $\Sigma \vdash_{\mathrm{wft}} \theta\widehat{\tau}_2$. Thus, we can derive $\Sigma \vdash_{\mathrm{wft}} \theta\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ by [W-SUM].

The cases for [W-PROD] and [W-ARR] can be proven analogously. □

2.3.2. Subtyping

An important part of a type and effect system is subtyping, or the weaker subeffecting. In order to achieve the full generality we aim for, we choose subtyping over subeffecting for our analysis.

Figure 2.8 shows the rules defining the subtyping relation on annotated types. A type $\widehat{\tau}_1$ is a subtype of $\widehat{\tau}_2$ under a sort environment Σ, written $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$, if a value of type $\widehat{\tau}_1$ can be used in places where a value of type $\widehat{\tau}_2$ is required.

Note that the subtyping relation still requires types to have the same shape, i.e. only sums can be subtypes of sums, etc. It only relates the annotations inside the types using the subsumption relation $\Sigma \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$ between dependency terms (see definition 2.12). Moreover, the subtyping relation implicitly demands that both types are well-formed under the environment.

The [SUB-FORALL] rule requires that the quantified variable has the same name in both types. This is not a restriction, as we can simply rename the variables in one or both of the types accordingly in order to make them match and prevent unintentional capturing of previously free variables.

Furthermore, in the [SUB-SUM] and [SUB-PROD] rules, both nested occurrences are in a covariant position, whereas the argument in the [SUB-ARR] rule is in a contravariant position.

The following lemma shows that the subtyping relation implies that the two involved types have the same shape.

Lemma 2.29. *Let Σ be a sort environment, and let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ be well-formed annotated types. If $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$, then $\widehat{\tau}_1$ and $\widehat{\tau}_2$ have the same shape.*

Proof. By induction on $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$.

[SUB-REFL] Since $\widehat{\tau}_1 = \widehat{\tau}_2$, they clearly have the same shape.

[SUB-TRANS] There is a $\widehat{\tau}_3$ such that $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$ and $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_2$ hold. By induction, $\widehat{\tau}_1$ and $\widehat{\tau}_3$ have the same shape, and $\widehat{\tau}_3$ and $\widehat{\tau}_2$ do. Therefore, $\widehat{\tau}_1$ and $\widehat{\tau}_2$ also have the same shape.

$$\frac{}{\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}} \text{ [Sub-Refl]} \qquad \frac{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_3}{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3} \text{ [Sub-Trans]}$$

$$\frac{\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2}{\Sigma \vdash_{\text{sub}} \forall \beta :: \kappa . \widehat{\tau}_1 \leqslant \forall \beta :: \kappa . \widehat{\tau}_2} \text{ [Sub-Forall]}$$

$$\frac{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1 \qquad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1 \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2 \qquad \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2}{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}'_2\langle \xi'_2 \rangle} \text{ [Sub-Sum]}$$

$$\frac{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1 \qquad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1 \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2 \qquad \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2}{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle \times \widehat{\tau}'_2\langle \xi'_2 \rangle} \text{ [Sub-Prod]}$$

$$\frac{\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}_1 \qquad \Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi_1 \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2 \qquad \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2}{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle \to \widehat{\tau}'_2\langle \xi'_2 \rangle} \text{ [Sub-Arr]}$$

Figure 2.8.: Subtyping relation ($\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$)

[Sub-Forall] We have $\widehat{\tau}_1 = \forall \beta\ \kappa . \widehat{\tau}'_1$ and $\widehat{\tau}_2 = \forall \beta\ \kappa . \widehat{\tau}'_2$ such that $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ holds. By induction, $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape, thus $\widehat{\tau}_1$ and $\widehat{\tau}_2$ also have the same shape.

[Sub-Sum] We have $\widehat{\tau}_1 = \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}''_1\langle \xi''_1 \rangle$ and $\widehat{\tau}_2 = \widehat{\tau}'_2\langle \xi'_2 \rangle + \widehat{\tau}''_2\langle \xi''_2 \rangle$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_2$ hold. By induction, $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ have the same shape, as well as $\widehat{\tau}''_1$ and $\widehat{\tau}''_2$. Therefore, $\widehat{\tau}_1$ and $\widehat{\tau}_2$ also have the same shape.

The cases for [Sub-Prod] and [Sub-Arr] can be proven analogously to [Sub-Sum]. □

The following lemma allows us to draw conclusions about the subtyping relation between parts of two types when there is a subtyping relation between the whole types.

Lemma 2.30. *1. If we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle \to \widehat{\tau}'_2\langle \xi'_2 \rangle$, then we also have $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}_1$, $\Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi_1$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2$.*

2. If we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}'_2\langle \xi'_2 \rangle$, then we also have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1$, $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2$.

3. If we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle \times \widehat{\tau}'_2\langle \xi'_2 \rangle$, then we also have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1$, $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2$.

4. If we have $\Sigma \vdash_{\text{sub}} \forall \beta :: \kappa . \widehat{\tau}_1 \leqslant \forall \beta :: \kappa . \widehat{\tau}_2$, then we also have $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$.

Proof. We prove the first statement by induction on the derivation of $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle \to \widehat{\tau}'_2\langle \xi'_2 \rangle$.

[Sub-Refl] We have $\widehat{\tau}_1 = \widehat{\tau}'_1$, $\widehat{\tau}_2 = \widehat{\tau}'_2$, $\xi_1 = \xi'_1$ and $\xi_2 = \xi'_2$. The conclusions hold by reflexivity.

[Sub-Trans] There are $\widehat{\tau}$, $\widehat{\tau}'$, ξ, ξ' such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau}\langle\xi\rangle \to \widehat{\tau}'\langle\xi'\rangle$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}\langle\xi\rangle \to \widehat{\tau}'\langle\xi'\rangle \leqslant \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_2'\langle\xi_2'\rangle$. The conclusions follow by induction and transitivity.

[Sub-Arr] The conclusions follow directly from the premises of the rule at hand.

All other cases can be ruled out because they do not apply to arrow types.

The remaining statements about sum types, product types and quantification follow analogously. □

Similar to the previous kinds of judgments, we can reorder the environment of subtyping judgments under certain conditions.

Lemma 2.31. *Let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ be annotated types and Σ a sort environment such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$. Then, for all Σ' such that for all $\beta \in \text{fav}(\widehat{\tau}_1 \cup \widehat{\tau}_2)$ we have $\Sigma(\beta) = \Sigma'(\beta)$, $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ holds.*

Proof. By induction on $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$.

[Sub-Refl] We have $\widehat{\tau}_1 = \widehat{\tau}_2$. Since we implicitly assume the well-formedness of this type, we also have $\Sigma' \vdash_{\text{wft}} \widehat{\tau}_1$ by lemma 2.22. Thus, we can derive $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ by [Sub-Refl].

[Sub-Trans] There is a $\widehat{\tau}_3$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_2$ hold. By induction, $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$ and $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_2$. Therefore, $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ holds by [Sub-Trans].

[Sub-Forall] We have $\widehat{\tau}_1 = \forall\beta :: \kappa.\widehat{\tau}_1'$ and $\widehat{\tau}_2 = \forall\beta :: \kappa.\widehat{\tau}_2'$ such that $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$ holds.

By definition, $\text{fav}(\widehat{\tau}_1') \subseteq \text{fav}(\widehat{\tau}_1) \cup \{\beta\}$ and $\text{fav}(\widehat{\tau}_2') \subseteq \text{fav}(\widehat{\tau}_2) \cup \{\beta\}$. Since additionally, $(\Sigma, \beta :: \kappa)(\beta) = \kappa = (\Sigma', \beta :: \kappa)(\beta)$, we can apply the induction hypothesis. This results in $\Sigma', \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$, from which we can derive $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ by [Sub-Forall].

[Sub-Sum] We have $\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle$ and $\widehat{\tau}_2 = \widehat{\tau}_2'\langle\xi_2'\rangle + \widehat{\tau}_2''\langle\xi_2''\rangle$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$, $\Sigma \vdash_{\text{sub}} \xi_1' \sqsubseteq \xi_2'$ and $\Sigma \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$ hold.

By induction, $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$ and $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$. By lemma 2.19, $\Sigma' \vdash_{\text{sub}} \xi_1' \sqsubseteq \xi_2'$ and $\Sigma' \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$. In conclusion, we have $\Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ by [Sub-Sum].

[Sub-Prod] Analogous to the previous case.

[Sub-Arr] Analogous to the previous cases, except taking into account the contravariance of the argument types and effects.

□

We also show that substitutions preserve the subtyping relation of annotated types.

Lemma 2.32. *Let $\widehat{\tau}_1$, $\widehat{\tau}_2$ be annotated types, let Σ and Σ' be sort environments and let $\theta : \mathrm{dom}(\Sigma') \to$ **AnnTm** be a substitution.*

If we have $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$, then we also have $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_2$.

Proof. By induction on the derivation of $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$.

[Sub-Refl] We have $\widehat{\tau}_1 = \widehat{\tau}_2$ and the type $\theta\widehat{\tau}_1$ is well-formed under Σ by lemma 2.28 (because $\widehat{\tau}_1$ is implicitly assumed to be well-formed), hence we have $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_2$ by [Sub-Refl].

[Sub-Trans] There is some $\widehat{\tau}_3$ such that $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$ and $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_2$ hold. By induction, we get $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_3$ and $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_3 \leqslant \theta\widehat{\tau}_2$. Thus, $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_2$ can be derived using transitivity.

[Sub-Forall] We have $\widehat{\tau}_1 = \forall\beta :: \kappa.\widehat{\tau}'_1$ and $\widehat{\tau}_2 = \forall\beta :: \kappa.\widehat{\tau}'_2$ such that $\Sigma, \Sigma', \beta :: \kappa \vdash_{\mathrm{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ holds.

We assume without loss of generality that β does not occur in the environment Σ' by appropriate renaming.

By lemma 2.31, we have $\Sigma, \beta{::}\kappa, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$. Applying the induction hypothesis results in $\Sigma, \beta :: \kappa \vdash_{\mathrm{sub}} \theta\widehat{\tau}'_1 \leqslant \theta\widehat{\tau}'_2$. We can derive $\Sigma \vdash_{\mathrm{sub}} \forall\beta :: \kappa.\theta\widehat{\tau}'_1 \leqslant \forall\beta :: \kappa.\theta\widehat{\tau}'_2$ by [Sub-Forall]. By assumption, β is not in the domain of θ, hence $\theta\forall\beta :: \kappa.\widehat{\tau}'_1 = \forall\beta :: \kappa.\theta\widehat{\tau}'_1$ and similarly $\theta\forall\beta :: \kappa.\widehat{\tau}'_2 = \forall\beta :: \kappa.\theta\widehat{\tau}'_2$. This completes the proof.

[Sub-Sum] We have $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}''_1\langle\xi''_1\rangle$ and $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle + \widehat{\tau}''_2\langle\xi''_2\rangle$ such that $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$, $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_2$, $\Sigma \vdash_{\mathrm{sub}} \xi'_1 \sqsubseteq \xi'_2$ and $\Sigma \vdash_{\mathrm{sub}} \xi''_1 \sqsubseteq \xi''_2$ hold.

Applying lemma 2.17 using θ as both the left and the right substitution results in $\Sigma \vdash_{\mathrm{sub}} \theta\xi'_1 \sqsubseteq \theta\xi'_2$ as well as $\Sigma \vdash_{\mathrm{sub}} \theta\xi''_1 \sqsubseteq \theta\xi''_2$. By induction, we get $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}'_1 \leqslant \theta\widehat{\tau}'_2$ and $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}''_1 \leqslant \theta\widehat{\tau}''_2$.

This lets us derive $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_2$ using [Sub-Sum].

[Sub-Prod] Analogous to the previous case.

[Sub-Arr] We have $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}''_1\langle\xi''_1\rangle$ and $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle \to \widehat{\tau}''_2\langle\xi''_2\rangle)$ such that $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}'_2 \leqslant \widehat{\tau}'_1$, $\Sigma, \Sigma' \vdash_{\mathrm{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_2$, $\Sigma \vdash_{\mathrm{sub}} \xi'_2 \sqsubseteq \xi'_1$ and $\Sigma \vdash_{\mathrm{sub}} \xi''_1 \sqsubseteq \xi''_2$ hold.

Applying lemma 2.17 using θ as both the left and the right substitution results in $\Sigma \vdash_{\mathrm{sub}} \theta\xi'_2 \sqsubseteq \theta\xi'_1$ as well as $\Sigma \vdash_{\mathrm{sub}} \theta\xi''_1 \sqsubseteq \theta\xi''_2$. By induction, we get $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}'_2 \leqslant \theta\widehat{\tau}'_1$ and $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}''_1 \leqslant \theta\widehat{\tau}''_2$.

This lets us derive $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau}_1 \leqslant \theta\widehat{\tau}_2$ using [Sub-Arr].

□

2.3.3. Type and Effect System

We have now all the definitions in place in order to explain the declarative type and effect system shown in figure 2.9. It consists of judgments of the form $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ expressing that under the sort environment Σ and the type and effect environment $\widehat{\Gamma}$, the annotated term $\widehat{t}$ has the annotated type $\widehat{\tau}$ and the dependency term ξ. The dependency term in this context is also called the *effect* of $\widehat{t}$. It is implicitly assumed that every type $\widehat{\tau}$ is also well-formed under Σ, i.e. $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}$, and that the resulting effect ξ is of sort $\star$, i.e. $\Sigma \vdash_{\mathrm{s}} \xi : \star$.

Remark 2.33. Since the type system relates an expression to both a type and an effect, we write top level type signatures for a term $\widehat{t}$ as $\widehat{t} : \widehat{\tau} \,\&\, \xi$ with the meaning that the term $\widehat{t}$ has the annotated type $\widehat{\tau}$ and the effect ξ.

A type and effect environment $\widehat{\Gamma}$ is defined analogously to type environments (see definition 2.3), but instead maps term variables x to pairs of an annotated type $\widehat{\tau}$ and an effect ξ. We extend the definition of the set of free annotation variables to type and effect environments by taking the union of all types and effects occurring in the environment, denoted by $\mathrm{fav}(\widehat{\Gamma})$. We denote the set of type and effect environments by **TyEffEnv**. When it is clear from the context whether we mean a type environment Γ or a type and effect environment $\widehat{\Gamma}$, we simply write Γ in both cases.

We can also define the erasure function pointwise on type and effect environments by erasing annotations from the type mappings and forgetting about the effects:

$$\begin{aligned} \lfloor [\,] \rfloor &= [\,] \\ \lfloor \widehat{\Gamma}, x : \widehat{\tau} \,\&\, \xi \rfloor &= \lfloor \widehat{\Gamma} \rfloor, x : \lfloor \widehat{\tau} \rfloor. \end{aligned}$$

Most of the typing rules correspond to a rule from the underlying type system. However, there are three new rules, namely [T-Sub] for subtyping and [T-AnnAbs] and [T-AnnApp] for dependency abstraction and application.

The following list gives a short description of every rule in the type and effect system.

[T-Var] Here, both the annotated type and the effect are looked up in the environment.

[T-Unit] The effect of the unit value defaults to the least annotation. While we could admit an arbitrary effect here, the same can be achieved by using the subtyping rule [T-Sub].

[T-App] The rule for function application seems overly restrictive by requiring that the types and effects of the arguments match, and that the effects of the return value and the function itself are the same. However, in combination with the subtyping rule [T-Sub], this effectively does not restrict the analysis in any way.

Moreover, the effect of the argument does not play a role in the resulting effect of the application. This is because we are dealing with a lazy language which means that the argument is not necessarily evaluated

$$\frac{\widehat{\Gamma}(x) = \widehat{\tau} \,\&\, \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} x : \widehat{\tau} \,\&\, \xi}\;[\text{T-Var}] \qquad \frac{}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} () : \widehat{\text{unit}} \,\&\, \bot}\;[\text{T-Unit}]$$

$$\frac{\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} t : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \lambda x : \widehat{\tau}_1 \,\&\, \xi_1 . t : \widehat{\tau}_1\langle \xi_1 \rangle \to \tau_2 \langle \xi_2 \rangle \,\&\, \bot}\;[\text{T-Abs}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_1 : \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \xi_2 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_2 : \widehat{\tau}_1 \,\&\, \xi_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_1 \; t_2 : \widehat{\tau}_2 \,\&\, \xi_2}\;[\text{T-App}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} (t_1, t_2) : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \bot}\;[\text{T-Pair}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \xi_i}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{proj}_i(t) : \widehat{\tau}_i \,\&\, \xi_i}\;[\text{T-Proj}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}_1 \,\&\, \xi_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{inl}_{\lfloor \widehat{\tau}_2 \rfloor}(t) : \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \bot}\;[\text{T-Inl}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{inr}_{\lfloor \widehat{\tau}_1 \rfloor}(t) : \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \bot}\;[\text{T-Inr}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \xi \qquad \begin{array}{c} \Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi \\ \Sigma \mid \widehat{\Gamma}, y : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\text{te}} t_2 : \widehat{\tau} \,\&\, \xi \end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \;\textbf{case}\; t \;\textbf{of}\; \{\text{inl}(x) \to t_1; \text{inr}(y) \to t_2\} : \widehat{\tau} \,\&\, \xi}\;[\text{T-Case}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi \qquad \Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(t) : \widehat{\tau} \,\&\, \xi}\;[\text{T-Ann}] \qquad \frac{\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau} \,\&\, \xi \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau} \,\&\, \xi . t : \widehat{\tau} \,\&\, \xi}\;[\text{T-Fix}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{seq}\; t_1 \; t_2 : \widehat{\tau}_2 \,\&\, \xi}\;[\text{T-Seq}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}' \,\&\, \xi' \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \qquad \Sigma \vdash_{\text{sub}} \xi' \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi}\;[\text{T-Sub}]$$

$$\frac{\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi \qquad \beta \notin \text{fav}(\widehat{\Gamma}) \cup \text{fav}(\xi)}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \Lambda \beta :: \kappa . t : \forall \beta :: \kappa . \widehat{\tau} \,\&\, \xi}\;[\text{T-AnnAbs}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \forall \beta :: \kappa . \widehat{\tau} \,\&\, \xi \qquad \Sigma \vdash_{\text{s}} \xi' : \kappa}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t \; \langle \xi' \rangle : [\xi' \,/\, \beta]\widehat{\tau} \,\&\, \xi}\;[\text{T-AnnApp}]$$

Figure 2.9.: Declarative type and effect system ($\Sigma \mid \Gamma \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$)

before the function call. It should be noted that this does not mean that the effects of arguments are ignored completely. If the body of a function makes use of an argument, the type system makes sure that its effect is also incorporated into the result.

[T-Pair] When constructing a pair, the effects of the components are stored in the type while the pair itself is assigned the least effect.

[T-Proj] When accessing a component of a pair, we require that the effect of the pair matches the effect of the projected component. Again, this is no restriction due to the subeffecting rule.

[T-Inl/Inr] The argument to the injection constructor only determines the type and annotation of one component of the sum type while the other component can be chosen arbitrarily as long as the underlying type matches the annotation on the constructor.

[T-Case] Again, to keep the rule simple, we demand that the types of both branches match, and that additionally the effects of both branches and the scrutinee are equal.

[T-Ann] The annotation rule requires that the effect of the term being annotated is at least as large as the lattice element ℓ. This can be achieved through prior application of the subtyping rule.

[T-Fix] In the fixpoint rule, not only the types but also the effects of the term itself and the bound variables must match. Note that this rule also admits polyvariant recursion, since quantification can occur anywhere in an annotated type.

[T-Seq] Since seq t_1 t_2 forces the evaluation of its first argument, it requires that its effect is part of the final result. This is justified because the result depends on the termination behavior of t_1.

Note that the current formulation assumes that t_1 is only evaluated to *weak head normal form* [22, p. 198]. If this were not the case, we would need to include all nested annotations belonging to forced subterms in the resulting effect as well.

[T-Sub] The subtyping rule allows to weaken the annotations nested inside a type through the subtyping relation (see figure 2.8), as well as the effects itself through the subsumption relation (see definition 2.12).

[T-AnnAbs] This rule introduces an annotation variable β of sort κ in the body t of the abstraction. The second premise ensures that the annotation variable does not escape its scope determined by the quantification on the type level.

[T-AnnApp] The annotation application rule allows the instantiation of an annotation variable with an arbitrary well-sorted dependency term.

The remainder of this section contains several lemmas about the declarative type system that will become useful in later proofs. We begin by showing that whenever a target term is typeable in the declarative type system, the corresponding source term is typeable in the underlying type system with the erased annotated type.

Lemma 2.34. *If we have* $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ *then we also have* $\lfloor \widehat{\Gamma} \rfloor \vdash_{\mathrm{t}} \lfloor \widehat{t} \rfloor : \lfloor \widehat{\tau} \rfloor$.

Proof. See appendix, page 101. □

The converse of the foregoing lemma, i.e. that every well-typed source term has a corresponding target term that is typeable in the declarative type system, is covered in the next chapter.

The following lemma is used in various proofs making assumptions about the structure of a typing derivation.

Lemma 2.35. *If we have a derivation* $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$*, then there is a derivation of the statement that ends with exactly one application of* [T-Sub].

Proof. By induction on $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$.

[T-Sub] We have

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau}' \,\&\, \xi' \qquad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \qquad \Sigma \vdash_{\mathrm{sub}} \xi' \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

and by induction, there is a derivation for $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau}' \,\&\, \xi'$ ending with [T-Sub]:

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau}'' \,\&\, \xi'' \qquad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau}'' \leqslant \widehat{\tau}' \qquad \Sigma \vdash_{\mathrm{sub}} \xi'' \sqsubseteq \xi'}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau}' \,\&\, \xi'}\ [\text{T-Sub}]$$

But then we can derive

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau}'' \,\&\, \xi'' \qquad \dfrac{\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}'' \leqslant \widehat{\tau}' \qquad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau}' \leqslant \widehat{\tau}}{\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}'' \leqslant \widehat{\tau}} \qquad \Sigma \vdash_{\mathrm{sub}} \xi'' \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi}$$

by [Sub-Trans] and the transitivity of subsumption.

In all other cases, we can derive

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi \qquad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau} \leqslant \widehat{\tau} \qquad \Sigma \vdash_{\mathrm{sub}} \xi \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

by reflexivity. □

Remark 2.36. The above lemma lets us deconstruct a typing judgment for a known term, because all rules except for [T-SUB] are syntax directed.

In particular, the following lemma allows us to deconstruct the typing judgment of an annotated term.

Lemma 2.37. *If we have* $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} ann_\ell(t) : \widehat{\tau} \,\&\, \xi$*, then we also have* $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$ *and* $\Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi$*.*

Proof. By lemma 2.35, there is a derivation for $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(t) : \widehat{\tau} \,\&\, \xi$ of the following form.

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}' \,\&\, \xi' \qquad \Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi'}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(t) : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-ANN}] \qquad \begin{array}{c}\Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \\ \Sigma \vdash_{\text{sub}} \xi' \sqsubseteq \xi\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(t) : \widehat{\tau} \,\&\, \xi}\,[\text{T-SUB}]$$

But then, we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$ using rule [T-SUB], and we have $\Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi$ by transitivity. □

The existence of a typing derivation also provides information about the free variables in a target term.

Lemma 2.38. *If we have* $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$*, then* $\text{ftv}(t) \subseteq \text{dom}(\widehat{\Gamma})$*.*

Proof. By induction on the derivation of $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$. In particular, whenever we encounter a variable in t, that variable is also part of the context $\widehat{\Gamma}$. Also, whenever bound variables are removed from the context (as in the [T-ABS] and [T-CASE] rules), this is reflected by the definition of ftv. □

Similar to the well-sortedness judgments for dependency terms we can show that all variables in the context that are not free in the term have no effect on its typing judgment.

Lemma 2.39. *Let* t *be a target term,* $\widehat{\tau} \,\&\, \xi$ *a type and effect pair,* Σ *a sort environment and* $\widehat{\Gamma}, \widehat{\Gamma}'$ *two type and effect environments.*

If we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \& \xi$ *and for all* $x \in \text{ftv}(t)$, $\widehat{\Gamma}(x) = \widehat{\Gamma}'(x)$*, then* $\Sigma \mid \widehat{\Gamma}' \vdash_{\text{te}} t : \widehat{\tau} \& \xi$*.*

Proof. By induction on $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$, similar to the previous reordering proofs. □

Unsurprisingly, the same holds for annotations variables.

Lemma 2.40. *Let* t *be a target term,* $\widehat{\tau} \,\&\, \xi$ *a type and effect pair,* Σ *and* Σ' *sort environments and* $\widehat{\Gamma}$ *two type and effect environments.*

If we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \& \xi$ *and for all* $\beta \in \text{fav}(t)$, $\Sigma(x) = \Sigma'(x)$*, then* $\Sigma' \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \& \xi$*.*

Proof. By induction on $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$, again similar to previous proofs for the other judgments. □

2.4. Embedding Various Analyses

This section serves to demonstrate how we can instantiate $\mathcal{L}$ to concrete lattices in order to perform various kinds of dependency analyses. Even though all of the following lattices are finite, they allow expressive analyses nonetheless. We also motivate the noninterference theorem that will be introduced later in this chapter by presenting cases where it can be applied to guarantee the safety of the analysis.

2.4.1. Binding-Time Analysis

Binding-time analysis is used in partial evaluation to determine which parts of a program can be evaluated at compile time, and which parts depend on dynamic values like user input and therefore can only be evaluated at runtime [21].

We can capture this distinction with a two-element lattice $\mathcal{BT} = \{\mathbf{S}, \mathbf{D}\}$ with $\mathbf{S} \sqsubset \mathbf{D}$. Static binding-time is denoted by $\mathbf{S}$ while $\mathbf{D}$ denotes dynamic binding-time. By adding more intermediate values, this analysis can be generalized to staged compilation

This choice of lattice can be motivated by looking at the subtyping and subsumption relations. These rules allow us to use values annotated with static binding-time wherever dynamic binding-time is required, but not the other way around.

Any terms t that somehow depend on dynamic input can then be wrapped in an annotation, e.g. $\mathrm{ann}_{\mathbf{D}}(t)$, and any terms depending on t are then also identified as dynamic.

Example 2.41. Consider the following term $(\lambda x : \mathrm{int} \,\&\, \mathbf{D}.0)\ \mathrm{ann}_{\mathbf{D}}(t)$. We assume here that we have another base type, int, besides the unit type in order to present a more meaningful example.

We find that even though the term involves parts with dynamic binding-time, it is actually static. As the lambda-term represents a constant function, there is indeed a valid typing derivation where the function takes a dynamic argument and returns a static value.

$$\dfrac{\dfrac{\emptyset \mid x : \mathrm{int} \,\&\, \mathbf{D} \vdash_{\mathrm{te}} 0 : \mathrm{int} \,\&\, \mathbf{S}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \lambda x : \mathrm{int} \,\&\, \mathbf{D}.0 : \mathrm{int}\langle \mathbf{D} \rangle \to \mathrm{int}\langle \mathbf{S} \rangle \,\&\, \mathbf{S}} \qquad \dfrac{\dfrac{\vdots}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} t : \mathrm{int} \,\&\, \mathbf{D}} \qquad \mathbf{D} \sqsubseteq \mathbf{D}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{ann}_{\mathbf{D}}(t) : \mathrm{int} \,\&\, \mathbf{D}}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} (\lambda x : \mathrm{int} \,\&\, \mathbf{D}.0)\ \mathrm{ann}_{\mathbf{D}}(t) : \mathrm{int} \,\&\, \mathbf{S}}$$

The typing judgment of the function body assigns it a static effect. As we will later see, noninterference guarantees that the variable x in the context has no influence on the outcome of the evaluation in that case.

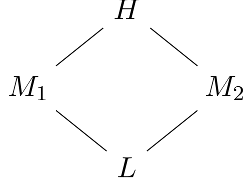
2.4.2. Security Analysis

The goal of security analyses, such as the SLam calculus [5], is to find out the flow of classified values in a program. If computations depend on high-security values, their results should also be marked as such.

A simple security analysis would only distinguish between low- and high-security values, which can be represented as the lattice $\mathcal{S}_1 = \{L, H\}$ with $L \sqsubset H$.

The desired security guarantee, i.e. that computations depending on high-security values are themselves classified as high-security, then follows from noninterference.

As the two element security lattice is identical to the one used in binding-time analysis, let us provide a more involved example with a four-element lattice $\mathcal{S}_2$ given by the following Hasse diagram[2].



In addition to a low- and a high-security annotation, we also have two intermediate classes M_1 and M_2 that are unrelated to each other. This means that any computation only working with values of class either M_1 or M_2 produces a result with the same classification. However, if a computation depends on both a value of class M_1 and a value of class M_2, it can only be assigned the highest security class H. That is because neither M_1 nor M_2 subsume the other.

In a real world case, M_1 and M_2 might denote the security levels of two different government departments that are only allowed to see their own information and unclassified (L) data. The highest security level (H) marks data that only the president may see.

Therefore, a function creating an overall activity report of all departments mixing this data returns a value classified as H. It is, in fact, guaranteed to do so by noninterference.

Example 2.42. For the sake of simplicity, we assume that the data from each department is a simple boolean (for example indicating some kind of success). The following aggregation function returns whether both departments reported a success.

$$\lambda r_1 : \widehat{\text{bool}} \,\&\, M_1 . \lambda r_2 : \widehat{\text{bool}} \,\&\, M_2 .\ \textbf{if}\ r_1\ \textbf{then}\ r_2\ \textbf{else}\ \textit{false}$$

The example uses a type $\widehat{\text{bool}}$ which can simply be seen as an abbreviation for the equivalent sum type $\widehat{\text{unit}}\langle\bot\rangle + \widehat{\text{unit}}\langle\bot\rangle$ with $\textit{true} = \text{inl}_{\text{unit}}(())$ and $\textit{false} = \text{inr}_{\text{unit}}(())$. Likewise, conditionals $\textbf{if}\ c\ \textbf{then}\ t_1\ \textbf{else}\ t_2$ can then be substituted by $\textbf{case}\ c\ \textbf{of}\ \{\text{inl}(_) \rightarrow t_1; \text{inr}(_) \rightarrow t_2\}$.

If we want to prevent each department from gaining knowledge of the result of the other department, the aggregation must have the highest security level. This is because knowing one input and the output can be enough to infer the second parameter under certain circumstances.

Indeed, this is ensured by the type system, in this case in particular by the rule [T-Case]. It requires the effects of the scrutinee as well as both branches to be the same. As

[2]There is an upward edge from x to y whenever $x \sqsubset y$ and there is no z such that $x \sqsubset z \sqsubset y$.

the subtyping rule only allows us to increase effects, the only annotation that is greater than both M_1 and M_2 is H.

Hence, the type of the above aggregation function is $\widehat{\text{bool}}\langle M_1\rangle \to \widehat{\text{bool}}\langle M_2\rangle \to \widehat{\text{bool}}\langle H\rangle$.

2.4.3. Exception Analysis

It turns out that this system is even general enough to capture the exception analysis that has been the starting point of this work.

To see why, suppose there is a finite set of exceptions E. We then define the base lattice of exception analysis $\mathcal{E} := \langle \mathcal{P}(E), \cup\rangle$, the lattice of sets of exception sets with the union operator.

The source language used in the paper by Koot [15] includes an exception term $\lightning^e_\tau$ of type τ raising the exception $e \in E$. It can be translated to $\text{ann}_{\{e\}}(\mathit{error}_\tau)$ where error_τ is some value of type τ that causes evaluation to get stuck.

Since our source language does not directly include such a term, we could define it as a non-terminating fixpoint $\mathit{error}_\tau = \mu x : \tau.x$.

Indeed, this translation also admits the desired typing in the declarative type and effect system, as witnessed by the following derivation tree.

$$\dfrac{\dfrac{\dfrac{(\widehat{\Gamma}, x : \widehat{\tau} \,\&\, \{e\})\ (x) = \widehat{\tau} \,\&\, \{e\}}{\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau} \,\&\, \{e\} \vdash_{\text{te}} x : \widehat{\tau} \,\&\, \{e\}}\ [\text{T-Var}]}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau} \,\&\, (\{e\}).x : \widehat{\tau} \,\&\, \{e\}}\ [\text{T-Fix}] \qquad \dfrac{\forall \rho.\ [\![\{e\}]\!]_\rho \subseteq [\![\{e\}]\!]_\rho}{\Sigma \vdash_{\text{sub}} \{e\} \sqsubseteq \{e\}}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_{\{e\}}(\mu x : \widehat{\tau} \,\&\, (\{e\}).x) : \widehat{\tau} \,\&\, \{e\}}\ [\text{T-Ann}]$$

Note that in this case we could have just omitted the explicit annotation because the fixpoint rule allows us to choose an arbitrary effect anyway. However, it serves as a guard that we indeed do make the right choice. Furthermore, the type reconstruction algorithm presented in the next chapter will produce the smallest possible effects and annotations, therefore requiring the explicit annotation to enforce the inclusion of the exception e in the effect.

2.5. Operational Semantics

The last part of this chapter is devoted to developing a noninterference proof for our declarative type system. We start by defining a small-step operational semantics for the target language based on the call-by-name semantics employed by Koot for exception terms [15].

Special care is taken when manipulating annotation terms $\text{ann}_\ell(t)$. They are moved just as far outwards as necessary in order to reach a normal form, thereby computing the least “permission” an evaluator must possess for computing a certain output.

Standard *progress* and *subject reduction* theorems ensure that well-typed terms can always be evaluated, and that the dynamic computation of the annotation value is indeed approximated by the static semantics.

$$
\begin{array}{lcl}
v' \in \mathbf{Nf'} & ::= & \lambda x : \widehat{\tau} \,\&\, \xi . t' \\
& | & \Lambda \beta :: \kappa . t' \\
& | & () \\
& | & \mathrm{inl}_\tau(t') \\
& | & \mathrm{inr}_\tau(t') \\
& | & (t_1, t_2) \\
\\
v \in \mathbf{Nf} & ::= & v' \mid \mathrm{ann}_\ell(v')
\end{array}
$$

Figure 2.10.: Values in the target language

$$
\begin{array}{lcl}
C & ::= & \mathrm{proj}_i(\square) \mid \square\ t_2 \mid \square\ \langle \xi \rangle \mid \mathrm{ann}_\ell(\square) \mid \mathrm{seq}\ \square\ t_2 \\
& | & \mathbf{case}\ \square\ \mathbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\}
\end{array}
$$

Figure 2.11.: Evaluation contexts

Figure 2.10 defines the values of the target language, i.e. those terms that cannot be further evaluated. Apart from a technicality related to annotations, they correspond exactly to the weak head normal forms of terms. The distinction for $\mathbf{Nf'} \subset \mathbf{Nf}$ is made in order to ensure that there is at most one annotation on the top level. Terms that are not values are called *redexes*.

Sometimes it is necessary to perform reduction on a sub-term, e.g. on the scrutinee of a case expression. Those circumstances are captured by contexts C (see figure 2.11) corresponding to terms with exactly one hole $\square$. We denote by $C[t]$ the term obtained by filling the single hole in C with t.

The reduction rules of the small-step semantics are shown in figure 2.12. It is an extension of the standard rules for a call-by-name evaluation of a functional language, i.e. in particular β-reduction is provided by [E-Abs] and deconstruction of data constructors by [E-Proj], [E-CaseInl] and [E-CaseInr]. The rule for fixpoints [E-Fix] unrolls the term one level at a time. Similar to β-reduction for term variables, rule [E-AnnAbs] implements β-reduction for annotation variables.

The [E-Context] rule allows to evaluate sub-terms where it is required for progressing the overall evaluation, and only there. This means that it is, for example, not possible to perform reduction on the argument of a constructor. The reason is that the semantics is designed to evaluate terms to WHNF, but not further.

Annotation terms $\mathrm{ann}_\ell(t)$ are always preserved by the semantics, in the sense that a term that is enclosed by an annotation always remains enclosed by an annotation at least as large. The rules moving annotations outwards are called *lifting* rules. There is also a rule [E-JoinAnn] which merges two adjacent annotation terms. Since the lattice values are joined, the resulting annotation is never smaller than any of the original ones.

$$\frac{}{(\lambda x : \widehat{\tau} \,\&\, \xi.t_1)\ t_2 \to [t_2 \,/\, x]t_1}\ [\textsc{E-Abs}] \qquad \frac{}{(\Lambda\beta :: \kappa.t')\ \langle\xi\rangle \to [\xi \,/\, \beta]t'}\ [\textsc{E-AnnAbs}]$$

$$\frac{}{\mu x : \widehat{\tau} \,\&\, \xi.t \to [\mu x : \widehat{\tau} \,\&\, \xi.t \,/\, x]t}\ [\textsc{E-Fix}] \qquad \frac{}{\mathrm{proj}_i(t_1, t_2) \to t_i}\ [\textsc{E-Proj}]$$

$$\frac{}{\textbf{case}\ \mathrm{inl}_\tau(t')\ \textbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\} \to [t' \,/\, x]t_2}\ [\textsc{E-CaseInl}]$$

$$\frac{}{\textbf{case}\ \mathrm{inr}_\tau(t')\ \textbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\} \to [t' \,/\, y]t_3}\ [\textsc{E-CaseInr}]$$

$$\frac{v' \in \mathbf{Nf}'}{\mathrm{seq}\ v'\ t_2 \to t_2}\ [\textsc{E-Seq}] \qquad \frac{t \to t'}{C[t] \to C[t']}\ [\textsc{E-Context}]$$

$$\frac{v' \in \mathbf{Nf}'}{(\mathrm{ann}_\ell(v'))\ t_2 \to \mathrm{ann}_\ell(v'\ t_2)}\ [\textsc{E-LiftApp}]$$

$$\frac{v' \in \mathbf{Nf}'}{(\mathrm{ann}_\ell(v'))\ \langle\xi\rangle \to \mathrm{ann}_\ell(v'\ \langle\xi\rangle)}\ [\textsc{E-LiftAnnApp}]$$

$$\frac{v' \in \mathbf{Nf}'}{\mathrm{proj}_i(\mathrm{ann}_\ell(v')) \to \mathrm{ann}_\ell(\mathrm{proj}_i(v'))}\ [\textsc{E-LiftProj}]$$

$$\frac{v' \in \mathbf{Nf}'}{\begin{array}{l}\textbf{case}\ (\mathrm{ann}_\ell(v'))\ \textbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\} \\ \to \mathrm{ann}_\ell(\ \textbf{case}\ v'\ \textbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\})\end{array}}\ [\textsc{E-LiftCase}]$$

$$\frac{v' \in \mathbf{Nf}'}{\mathrm{seq}\ (\mathrm{ann}_\ell(v'))\ t_2 \to \mathrm{ann}_\ell(\mathrm{seq}\ v'\ t_2)}\ [\textsc{E-LiftSeq}]$$

$$\frac{v' \in \mathbf{Nf}'}{\mathrm{ann}_{\ell_1}(\mathrm{ann}_{\ell_2}(v')) \to \mathrm{ann}_{\ell_1 \sqcup \ell_2}(v')}\ [\textsc{E-JoinAnn}]$$

Figure 2.12.: Small-step semantics ($t_1 \to t_2$)

All these rules require the annotated term to be a value already in order to make the reduction deterministic.

In the remainder of this section we present the theorems that ensure that our small-step semantics is compatible with the type system. The following progress theorem demonstrates that any well-typed term is either already a normal form, or an evaluation step can be performed.

Theorem 2.43 (Progress)**.** *If* $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$*, then either* $t \in \mathbf{Nf}$ *or there is a* t' *such that* $t \to t'$*.*

Proof. See appendix, page 101. □

The following lemma is needed in the proof of subject reduction. It states that substituting a variable in a well-typed term with a closed term of the right type again results in a well-typed term that no longer depends on the variable.

Lemma 2.44. *If* $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$ *and* $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t' : \widehat{\tau}' \,\&\, \xi'$*, then* $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$*.*

Proof. See appendix, page 103. □

Similarly, substituting an annotation variable with an annotation term of the right sort in a well-typed target term results again in a well-typed term.

Lemma 2.45. *If we have* $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ *and* $\Sigma \vdash_{\mathrm{s}} \xi' : \kappa$*, then* $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\mathrm{te}} [\xi' \,/\, \beta]\widehat{t} : [\xi' \,/\, \beta]\widehat{\tau} \,\&\, [\xi' \,/\, \beta]\xi$*.*

Proof. See appendix, page 105. □

Now we can show subject reduction, i.e. the reduction of a well-typed term results in a term of the same type.

Theorem 2.46 (Subject Reduction)**.** *If* $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$ *and there is a* t' *such that* $t \to t'$*, then* $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t' : \widehat{\tau} \,\&\, \xi$*.*

Proof. See appendix, page 106. □

Another useful property is that our small-step semantics is deterministic, i.e. for every term there is just one unique term a reduction can lead to.

Lemma 2.47 (Determinism)**.** *If we have* $t_1 \to t_2$ *and* $t_1 \to t_2'$*, then* $t_2 = t_2'$*.*

Proof. By induction on $t_1 \to t_2$.

[E-Abs] We have $t_1 = (\lambda x : \widehat{\tau} \,\&\, \xi . t_1')\ t$. If $t_1 \to t_2'$ also follows by [E-Abs], $t_2 = t_2'$ by definition. The only other rule with a matching head is [E-Context] with the context $\Box\ t$. But then, there would be a reduction $\lambda x : \widehat{\tau} \,\&\, \xi . t_1' \to t_1''$. This is a contradiction because a λ-abstraction is a normal form.

The cases for [E-AnnAbs], [E-Proj], [E-CaseInl], [E-CaseInr] and [E-Seq] can be proven analogously to the previous one. The same applies to [E-LiftApp], [E-LiftAnnApp], [E-LiftProj], [E-LiftCase], [E-LiftSeq] and [E-JoinAnn].

[E-FIX] Only this rule applies, therefore $t_2 = t'_2$ by definition.

[E-CONTEXT] We have $t_1 = C[t]$ and $t_2 = C[t']$ and a reduction $t \to t'$. Therefore, t cannot be a normal form. This rules out all potentially matching rules for $t_1 \to t'_2$, except for [E-CONTEXT] where we have $t'_2 = C[t'']$ and a reduction $t \to t''$. By induction, $t' = t''$ and therefore $t_2 = C[t'] = C[t''] = t'_2$.

□

In order to describe the evaluation of whole programs, we need to chain multiple individual reduction steps together.

Definition 2.48. We write $t \to^* v$ if there is a finite sequence of terms $(t_i)_{0 \leqslant i \leqslant n}$ with $t_0 = t$ and $t_n = v \in \mathbf{Nf}$ and reductions $(t_i \to t_{i+1})_{0 \leqslant i < n}$ between them. If there is no such a sequence, this is denoted by $t \Uparrow$ and t is said to *diverge*.

As expected, subject reduction extends naturally to a sequence of reductions.

Corollary 2.49. *If we have* $\emptyset \mid \emptyset \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$ *and* $t \to^* v$*, then* $\emptyset \mid \emptyset \vdash_{\text{te}} v : \widehat{\tau} \,\&\, \xi$*.*

Proof. By induction on the length of the reduction sequence, using theorem 2.46. □

And lastly, if a term evaluates to an annotated value, this annotation is compatible with the effect that has been assigned to the term.

Theorem 2.50 (Semantic Soundness)**.** *If we have* $\emptyset \mid \emptyset \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$ *and* $t \to^* ann_\ell(v')$*, then* $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$*.*

Proof. By corollary 2.49, we have $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \widehat{\tau} \,\&\, \xi$. By lemma 2.37, we then also have $\emptyset \mid \emptyset \vdash_{\text{te}} v' : \widehat{\tau} \,\&\, \xi$ and, in particular, $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$. □

For example, a term that has been assigned a low security annotation during security analysis cannot evaluate to a value annotated with high security protection. The other way around is possible, however. This is because the static semantics can only be an approximation to the dynamic semantics. Consider the term **case** t_1 **of** $\{\text{inl}(x) \to \text{ann}_L(t_2); \text{inr}(x) \to \text{ann}_H(t_3)\}$. Whether the evaluation ends with a low or high annotation depends on whatever t evaluates to. As predicting this is an undecidable problem, the static type system can only over-approximate the result and assign an effect that encompasses both branches.

2.6. Noninterference

An important theorem for the safety of program transformations/optimizations using the results of dependency analysis is *noninterference*. Informally, it guarantees that if there is a target term t depending on some variable x such that $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$ holds and the effect ξ' of the variable is not encompassed by the resulting effect ξ (i.e. $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$), then t will always evaluate to the same normal form, regardless of the value of x.

Since we are working in a non-strict setting, our noninterference property will only apply to the topmost constructors of the resulting values. This is because the effects derived in the annotated type system only provide information about the evaluation to weak head normal form. Nested terms might possess lower as well as higher classifications. In particular, the subterms with greater effects than their enclosing constructors prevent us from making a more general statement because those can still depend on the context whereas the top-level constructor cannot.

In the noninterference theorem presented for the SLam calculus by Heintze and Riecke, this problem is circumvented by restricting the statement to so called *transparent* types, where the annotations of nested components are decreasing when moving further inward [5]. For example, a low-security constructor may only contain other low-security values.

The following definition gives a formal description of what it means for two values to have the same top-level constructor.

Definition 2.51. We say two normal forms $v_1, v_2 \in \mathbf{Nf}$ are *similar*, if their top level constructors (and annotations, if present) match, i.e. if

- $v_1 = \lambda x : \widehat{\tau} \,\&\, \xi.t_1$ and $v_2 = \lambda x : \widehat{\tau} \,\&\, \xi.t_2$ for some t_1, t_2, or
- $v_1 = \Lambda\beta :: \kappa.t_1$ and $v_2 = \Lambda\beta :: \kappa.t_2$ for some t_1, t_2, or
- $v_1 = ()$ and $v_2 = ()$, or
- $v_1 = \mathrm{inl}_\tau(t_1)$ and $v_2 = \mathrm{inl}_\tau(t_2)$ for some t_1, t_2, or
- $v_1 = \mathrm{inr}_\tau(t_1)$ and $v_2 = \mathrm{inr}_\tau(t_2)$ for some t_1, t_2, or
- $v_1 = (t_1, t_1')$ and $v_2 = (t_2, t_2')$ for some t_1, t_1', t_2, t_2', or
- $v_1 = \mathrm{ann}_\ell(v_1')$ and $v_2 = \mathrm{ann}_\ell(v_2')$ and v_1' and v_2' (which cannot be annotations themselves) are similar.

We also use $v_1 \simeq v_2$ to denote the statement that v_1 and v_2 are similar.

When we have $v_1 \simeq v_2$ this means that these two values are indistinguishable without further evaluation. This is the property that is guaranteed by our noninterference theorem developed in this section.

We first show noninterference for normal forms.

Lemma 2.52. *Let t be a target term such that $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$ and $\emptyset \vdash_{\mathrm{sub}} \xi' \not\sqsubseteq \xi$.*
If there is a t_1 with $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_1 : \widehat{\tau}' \,\&\, \xi'$ such that $[t_1 \,/\, x]t \in \mathbf{Nf}$, then for all t_2 with $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$ we have $[t_2 \,/\, x]t \in \mathbf{Nf}$ and $[t_1 \,/\, x]t \simeq [t_2 \,/\, x]t$.

Proof. We first show that $t \neq x$ by contradiction. Suppose that $t = x$, then by lemma 2.35 there is a derivation

$$\dfrac{\dfrac{(x : \widehat{\tau}' \,\&\, \xi')(x) = \widehat{\tau}' \,\&\, \xi'}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} x : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-Var}] \qquad \emptyset \vdash_{\mathrm{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \qquad \emptyset \vdash_{\mathrm{sub}} \xi' \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} x : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

contradicting our assumption $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$. A similar reasoning can be used to rule out the case $t = \text{ann}_\ell(x)$.

This leaves us with either $t = v$ or $t = \text{ann}_\ell(v)$ where v falls under one of the following cases.

$v = \lambda y : \widehat{\tau}_1 \mathbin{\&} \xi_1.t'$ We assume without loss of generality that $x \neq y$. Then, $[t_1/x]v \in \mathbf{Nf}'$ and $[t_2 \,/\, x]v \in \mathbf{Nf}'$, and we have $[t_1 \,/\, x]v \simeq [t_2 \,/\, x]v$.

$v = \Lambda\beta :: \kappa.t'$ We have $[t_1 / x]v = \Lambda\beta :: \kappa.[t_1 / x]t' \in \mathbf{Nf}'$ and $[t_2 / x]v = \Lambda\beta :: \kappa.[t_2 / x]t' \in \mathbf{Nf}'$, therefore $[t_1 \,/\, x]v \simeq [t_2 \,/\, x]v$.

$v = ()$ It is easy to see that $[t_1 \,/\, x]v = () \in \mathbf{Nf}'$ and $[t_2 \,/\, x]v = () \in \mathbf{Nf}'$, and therefore $[t_1 \,/\, x]v \simeq [t_2 \,/\, x]v$.

$v = \mathsf{inl}_\tau(t')$ We have $[t_1 / x]v = \text{inl}_\tau([t_1 / x]t') \in \mathbf{Nf}'$ and $[t_2 / x]v = \text{inl}_\tau([t_2 / x]t') \in \mathbf{Nf}'$, therefore $[t_1 \,/\, x]v \simeq [t_2 \,/\, x]v$.

$v = \mathsf{inr}_\tau(t')$ Analogous to the previous case.

$v = (t_1, t_2)$ Similar to the previous cases.

In each of the cases, we have $[t_1 \,/\, x]v, [t_2 \,/\, x]v \in \mathbf{Nf}'$ and $[t_1 \,/\, x]v \simeq [t_2 \,/\, x]v$, and therefore also $[t_1 \,/\, x]\text{ann}_\ell(v) \simeq [t_2 \,/\, x]\text{ann}_\ell(v)$. This lets us conclude $[t_2 \,/\, x]t \in \mathbf{Nf}$ and $[t_1 \,/\, x]t \simeq [t_2 \,/\, x]t$. □

The following lemma is used to "recover" a more general typing statement from a type that is well-typed after a substitution.

Lemma 2.53. *If we have a term t such that for all t' with $\emptyset \mid \emptyset \vdash_{\text{te}} t' : \widehat{\tau}' \mathbin{\&} \xi'$ we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \mathbin{\&} \xi$, then $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} t : \widehat{\tau} \mathbin{\&} \xi$.*

Proof. See appendix, page 110. □

The following lemma provides the noninterference property for an individual reduction step.

Lemma 2.54. *Let t be a target term such that $\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} t : \widehat{\tau} \mathbin{\&} \xi$ and $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$. Let t' be a target term.*

If there is a t_1 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_1 : \widehat{\tau}' \mathbin{\&} \xi'$ such that $[t_1 \,/\, x]t \to t'$, then there is a t'' such that $\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} t'' : \widehat{\tau} \mathbin{\&} \xi$ and for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \mathbin{\&} \xi'$ we have $[t_2 \,/\, x]t \to [t_2 \,/\, x]t''$.

Proof. See appendix, page 114. □

And finally, we extend this statement to a sequence of reduction steps using the previous lemmas.

Theorem 2.55 (Noninterference)**.** *Let t be a target term such that $\emptyset \mid x{:}\widehat{\tau}' \& \xi' \vdash_{\text{te}} t{:}\widehat{\tau} \& \xi$ and $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$. Let v be a value.*

If there is a t_1 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_1 : \widehat{\tau}' \& \xi'$ such that $[t_1 / x]t \to^* v$, then there is a t' such that for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2{:}\widehat{\tau}' \& \xi'$ we have $[t_2 / x]t \to^* [t_2 / x]t'$ and $[t_1 / x]t' \simeq [t_2 / x]t'$.

Proof. By induction on the length n of the reduction sequence $[t_1 / x]t \to^* v$.

$n = 0$ In this case, $[t_1 / x]t = v \in \mathbf{Nf}$. By lemma 2.52, we have for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$, $[t_2 / x]t \in \mathbf{Nf}$ and $[t_1 / x]t \simeq [t_2 / x]t$. We choose $t' = t$ and get $[t_2 / x]t \to^* [t_2 / x]t'$ and $[t_1 / x]t' \simeq [t_2 / x]t'$.

$n = n' + 1$ We have $[t_1 / x]t \to t_3$ and $t_3 \to^* v$. By lemma 2.54, there is a t'' such that $\emptyset \mid x : \widehat{\tau}' \& \xi' \vdash_{\text{te}} t'' : \widehat{\tau} \& \xi$ holds and for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$, $[t_2 / x]t \to [t_2 / x]t''$ holds. In particular, $[t_1 / x]t \to [t_1 / x]t''$ and by lemma 2.47, $t_3 = [t_1 / x]t''$.

By induction, we get a t' such that for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$ we have $[t_2 / x]t'' \to^* [t_2 / x]t'$ and $[t_1 / x]t' \simeq [t_2 / x]t'$.

Combining this with the previous result, we can conclude for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$ that $[t_2 / x]t \to^* [t_2 / x]t'$ holds.

□

The noninterference proofs crucially rely on the fact that the source term is well-typed, and the additional assumption $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$ stating that the effect of the variable in the context is not encompassed by the effect of the term being evaluated.

By introducing a restriction to *transparent* types, we can recover the notion of noninterference used for the SLam calculus. For example, if we have a transparent type $\widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \& \xi$ (i.e. $\emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi$ and $\emptyset \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi$) and $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$ holds, then we also know $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi_1$ and $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi_2$. Otherwise, we would get $\emptyset \vdash_{\text{sub}} \xi' \sqsubseteq \xi$ by transitivity, contradicting the assumption. This means all prerequisites of the noninterference theorem are still fulfilled.

Hence, it is possible in these cases to apply the noninterference theorem to the nested (possibly unevaluated) subterms of a constructor in weak head normal form. It cannot be applied to the bodies of lambda abstractions though, because it only deals with terms depending on exactly one variable.

3. Type Reconstruction

In this chapter we will present an algorithmic interpretation of the type and effect system introduced in the previous chapter. We will first describe several important pieces that eventually lead to the type reconstruction algorithm, before finishing the chapter with its correctness proofs.

3.1. Equality of Dependency Terms

The type reconstruction algorithm is meant to handle polymorphic recursion through Kleene-Mycroft-iteration. Such an algorithm based on fixpoint iteration needs a way to efficiently decide whether two dependency terms are equal according to the denotational semantics of the $\lambda^{\sqcup}$ calculus.

A straightforward, yet inefficient way to decide semantic equivalence is to enumerate all possible environments and compare the denotations of the two terms in all of these. This leads to the following lemma.

Lemma 3.1. *Semantic equality of two dependency terms ξ_1 and ξ_2 under an environment Σ is decidable.*

Proof. First, we show by induction on sorts κ that the universes V_κ are finite.

$\kappa = \star$ In this case we have $V_\kappa = V_\star = \mathcal{L}$ which by assumption is finite.

$\kappa = \kappa_1 \Rightarrow \kappa_2$ By induction, V_{κ_1} and V_{κ_2} are finite. Therefore, the set $V_{\kappa_2}^{V_{\kappa_1}}$ of functions from V_{κ_1} to V_{κ_2} is also finite (with cardinality $|V_{\kappa_2}^{V_{\kappa_1}}| = |V_{\kappa_2}|^{|V_{\kappa_1}|}$). Since $V_{\kappa_1 \Rightarrow \kappa_2} \subseteq V_{\kappa_2}^{V_{\kappa_1}}$, it must also be finite.

For any environment ρ that is compatible with Σ we know $\mathrm{dom}(\rho) = \mathrm{dom}(\Sigma)$ and for all $\beta \in \mathrm{dom}(\rho)$ we have $\rho(\beta) \in V_{\Sigma(\beta)}$. $\mathrm{dom}(\Sigma)$ is finite because sort environments are finite. Moreover, the image $\mathrm{im}(\rho) \subseteq \bigcup\{V_\kappa \mid \kappa \in \mathrm{im}(\Sigma)\} \subseteq \bigcup\{V_\kappa \mid \kappa \in \mathbf{AnnSort}\}$ is finite because the image of Σ can only consist of finitely many sorts and a finite union of finite sets is again finite, even though there are infinitely many sorts.

Hence, there can only be a finite number of environments ρ that are compatible with Σ.

The denotation function $[\![-]\!]_\rho$ is structurally recursive on terms and can therefore be computed in finite time.

Thus, equality of two dependency terms ξ_1 and ξ_2 can be decided by enumerating all environments ρ, computing the denotations $[\![\xi_1]\!]_\rho$ and $[\![\xi_2]\!]_\rho$ and then comparing the results. □

As the universes can grow quite large, especially if the base lattice $V_\star$ is already big (think of the set of all program locations in slicing analysis), this method of deciding equality is impractical to use in an actual algorithm.

The exception analysis by Koot [15] that was the starting point for this thesis therefore computed term equality in a purely syntactic manner. In order to compare two exception terms for equality, they are converted into a normal form and then checked for alpha equivalence.

However, during the writing of this thesis it turned out that this approach is incomplete. Two terms can have different canonical forms even though they are actually semantically equivalent. This causes the reconstruction algorithm to diverge on certain input programs. We will discuss this issue further in section 3.4 of this chapter after presenting the parts involved in the issue.

In the remainder of this section, we will present a canonical form for $\lambda^{\sqcup}$-terms similar to the one for the $\lambda^{\cup}$-calculus by Koot. While we are reusing the reduction rules and normal forms of the $\lambda^{\cup}$-calculus, we tried to provide a more precise formulation for the canonical ordering that is imposed on normal forms in order to make them unique.

Note that it suffers from the same problems as the one it is modeled on. Nevertheless it is still useful in its current state by making the annotations computed by the analysis smaller and more readable for humans. Therefore, canonical forms are not required for the correct functioning of our version of the type reconstruction algorithm.

3.1.1. Reduction

The first step towards canonical forms is to provide syntactic transformations for manipulating terms while maintaining semantic equivalence. Figure 3.1 shows the reduction rules of the $\lambda^{\sqcup}$-calculus.

In addition to the β-reduction rule known from the simply-typed λ-calculus, there are two reduction rules [R-Gamma$_1$] and [R-Gamma$_2$] similar to the ones used in the $\lambda^{\cup}$-calculus [15]. Furthermore, the ACI1 axioms of the $\sqcup$-operator naturally give rise to the corresponding syntactic transformations.

Besides the aforementioned reduction rules, we may also perform η-reduction and η-expansion on $\lambda^{\sqcup}$-terms which is justified by the denotational semantics and by the assumption of functional extensionality in the meta-theory.

Lemma 3.2. *Suppose ξ is a $\lambda^{\sqcup}$-term such that $\Sigma \vdash_s \xi : \kappa_1 \Rightarrow \kappa_2$ holds. Then ξ and $\lambda\beta :: \kappa_1.\xi\ \beta$ are semantically equal if β does not occur free in ξ.*

Proof. Suppose ρ is an arbitrary environment that is compatible with Σ.

$$\begin{aligned}
[\![\lambda\beta :: \kappa_1.\xi\ \beta]\!]_\rho &= \lambda v \in V_{\kappa_1}.\, [\![\xi\ \beta]\!]_{\rho[\beta\mapsto v]} \\
&= \lambda v \in V_{\kappa_1}.\, [\![\xi]\!]_{\rho[\beta\mapsto v]}\,([\![\beta]\!]_{\rho[\beta\mapsto v]}) \\
&= \lambda v \in V_{\kappa_1}.\, [\![\xi]\!]_{\rho}\,(v) = [\![\xi]\!]_\rho
\end{aligned}$$

The last step is justified by functional extensionality while the preceding steps follow from the definition of the denotational semantics and lemma 2.18. ☐

$$\frac{\xi_1 \longrightarrow \xi_1'}{\xi_1\ \xi_2 \longrightarrow \xi_1'\ \xi_2}\ [\text{R-App}_1] \qquad \frac{\xi_2 \longrightarrow \xi_2'}{\xi_1\ \xi_2 \longrightarrow \xi_1\ \xi_2'}\ [\text{R-App}_2]$$

$$\frac{\xi_1 \longrightarrow \xi_2}{\lambda\beta :: \kappa.\xi_1 \longrightarrow \lambda\beta :: \kappa.\xi_2}\ [\text{R-Lam}] \qquad \frac{}{(\lambda\beta :: \kappa.\xi_1)\ \xi_2 \longrightarrow [\xi_2\ /\ \beta]\xi_1}\ [\text{R-Beta}]$$

$$\frac{}{(\xi_1 \sqcup \xi_2)\ \xi \longrightarrow \xi_1\ \xi \sqcup \xi_2\ \xi}\ [\text{R-Gamma}_1]$$

$$\frac{}{(\lambda\beta :: \kappa.\xi_1) \sqcup (\lambda\beta :: \kappa.\xi_2) \longrightarrow \lambda\beta :: \kappa.\xi_1 \sqcup \xi_2}\ [\text{R-Gamma}_2]$$

$$\frac{\xi_1 \longrightarrow \xi_1'}{\xi_1 \sqcup \xi_2 \longrightarrow \xi_1' \sqcup \xi_2}\ [\text{R-Join}_1] \qquad \frac{\xi_2 \longrightarrow \xi_2'}{\xi_1 \sqcup \xi_2 \longrightarrow \xi_1 \sqcup \xi_2'}\ [\text{R-Join}_2]$$

Figure 3.1.: $\lambda^{\sqcup}$-calculus: reduction rules

The following lemma demonstrates the correctness of the above reduction rules with respect to the denotational semantics.

Lemma 3.3. *Let ξ_1 and ξ_2 be dependency terms such that $\Sigma \vdash_s \xi_1 : \kappa$ and $\Sigma \vdash_s \xi_2 : \kappa$ hold for some Σ and κ.*
If $\xi_1 \to \xi_2$ holds, then $\Sigma \vdash \xi_1 \equiv \xi_2$.

Proof. By induction on $\xi_1 \to \xi_2$. Let ρ be an arbitrary environment compatible with Σ.

[R-Beta] Using lemma 2.16, we can derive

$$\begin{aligned} &[\![(\lambda\beta :: \kappa.\xi_1)\ \xi_2]\!]_\rho = [\![\lambda\beta :: \kappa.\xi_1]\!]_\rho\,([\![\xi_2]\!]_\rho) = (\lambda v \in V_\kappa.\ [\![\xi_1]\!]_{\rho[\beta\mapsto v]})([\![\xi_2]\!]_\rho) \\ =\ &[\![\xi_1]\!]_{\rho[\beta\mapsto [\![\xi_2]\!]_\rho]} = [\![[\xi_2\ /\ \beta]\xi_1]\!]_\rho \end{aligned}$$

[R-Gamma$_1$]

$$\begin{aligned} &[\![(\xi_1 \sqcup \xi_2)\ \xi]\!]_\rho = [\![\xi_1 \sqcup \xi_2]\!]_\rho\,([\![\xi]\!]_\rho) = ([\![\xi_1]\!]_\rho \sqcup [\![\xi_2]\!]_\rho)([\![\xi]\!]_\rho) \\ =&(\lambda v.\ [\![\xi_1]\!]_\rho\,(v) \sqcup [\![\xi_2]\!]_\rho\,(v))([\![\xi]\!]_\rho) = [\![\xi_1]\!]_\rho\,([\![\xi]\!]_\rho) \sqcup [\![\xi_2]\!]_\rho\,([\![\xi]\!]_\rho) \\ =&\ [\![\xi_1\ \xi]\!]_\rho \sqcup [\![\xi_2\ \xi]\!]_\rho = [\![\xi_1\ \xi \sqcup \xi_2\ \xi]\!]_\rho \end{aligned}$$

[R-Gamma$_2$]

$$\begin{aligned} &[\![(\lambda\beta :: \kappa.\xi_1) \sqcup (\lambda\beta :: \kappa.\xi_2)]\!]_\rho = [\![\lambda\beta :: \kappa.\xi_1]\!]_\rho \sqcup [\![\lambda\beta :: \kappa.\xi_2]\!]_\rho \\ =&(\lambda v \in V_\kappa.\ [\![\xi_1]\!]_{\rho[\beta\mapsto v]}) \sqcup (\lambda v \in V_\kappa.\ [\![\xi_2]\!]_{\rho[\beta\mapsto v]}) \\ =&\lambda v \in V_\kappa.\ [\![\xi_1]\!]_{\rho[\beta\mapsto v]} \sqcup [\![\xi_2]\!]_{\rho[\beta\mapsto v]} \\ =&\lambda v \in V_\kappa.\ [\![\xi_1 \sqcup \xi_2]\!]_{\rho[\beta\mapsto v]} \\ =&\ [\![\lambda\beta :: \kappa.\xi_1 \sqcup \xi_2]\!]_\rho \end{aligned}$$

[R-App$_1$] We have $\xi_1 \longrightarrow \xi_1'$ as a premise. Using the induction hypothesis, we can derive

$$[\![\xi_1\ \xi_2]\!]_\rho = [\![\xi_1]\!]_\rho\,([\![\xi_2]\!]_\rho) = [\![\xi_1']\!]_\rho\,([\![\xi_2]\!]_\rho) = [\![\xi_1'\ \xi_2]\!]_\rho$$

[R-App$_2$] We have $\xi_2 \longrightarrow \xi_2'$ as a premise. Using the induction hypothesis, we can derive

$$[\![\xi_1\ \xi_2]\!]_\rho = [\![\xi_1]\!]_\rho\,([\![\xi_2]\!]_\rho) = [\![\xi_1]\!]_\rho\,([\![\xi_2']\!]_\rho) = [\![\xi_1\ \xi_2']\!]_\rho$$

[R-Join$_{1/2}$] Analogous to the application cases.

[R-Lam] We have $\xi_1 \longrightarrow \xi_2$ as a premise. Using the induction hypothesis, we can derive

$$[\![\lambda\beta :: \kappa.\xi_1]\!]_\rho = \lambda v \in V_\kappa.\ [\![\xi_1]\!]_{\rho[\beta\mapsto v]} = \lambda v \in V_\kappa.\ [\![\xi_2]\!]_{\rho[\beta\mapsto v]} = [\![\lambda\beta :: \kappa.\xi_2]\!]_\rho$$

□

3.1.2. Canonical Forms

Equipped with the reduction rules and equivalences from the previous section we can now introduce canonical forms.

Definition 3.4. A $\lambda^\sqcup$-term c of sort $\kappa_1 \Rightarrow \cdots \Rightarrow \kappa_n \Rightarrow \star$ is in *canonical form* if it has the following shape.

$$\lambda\beta_1 : \kappa_1.\ldots.\lambda\beta_n : \kappa_n.\ k_1 \sqcup \cdots \sqcup k_m$$

where each of the $k_1, \ldots, k_m$ is either a lattice element ℓ or a saturated function application $\beta\ c_1 \cdots c_{a_\beta}$ where β is a variable with arity a_β and $c_1, \ldots, c_{a_\beta}$ are canonical forms themselves.

We call the sub-terms $k_1, \ldots, k_n$ *atoms*.

Remark 3.5. There are a few things to note about canonical forms. First of all, they are η-long, i.e. it is not possible to η-expand a sub-term without introducing additional β-redexes. This is because all function applications are already fully saturated, lattice elements are always η-long and all sub-terms that are possibly of function sort are canonical forms themselves.

They are also $\beta\gamma$-normal since there are no applications to lambda terms or joins, and no joins of lambda-terms.

However, we do not call them *normal forms* because the atoms can still be reordered and thereby possibly further reduced due to the underlying algebraic structure. For example, reordering could allow further joining of lattice elements that have not been adjacent previously.

We originally planned to prove that every $\lambda^\sqcup$-term can be transformed into such a canonical form by applying the given reduction rules and equivalences. However, due to the fact that canonicalization is no longer required for the correct functioning of the reconstruction algorithm, this goal became secondary and has ultimately been dropped in favor of other parts of this thesis. We assume that such a proof will likely be similar to a proof of strong normalization of the simply typed lambda-calculus while additionally ensuring the particular shape of the normal form.

$$\frac{\mathrm{idx}(\beta, \Sigma_1) < \mathrm{idx}(\beta', \Sigma_2)}{\Sigma_1 \mid \Sigma_2 \vdash \beta <_C \beta'} \text{[CO-Var]}$$

$$\frac{\Sigma_1, \beta_1 :: \kappa_1, ..., \beta_n :: \kappa_n \mid \Sigma_2, \beta_1' :: \kappa_1, ..., \beta_n' :: \kappa_n \vdash (k_1, ..., k_m) <_C (k_1', ..., k_{m'}')}{\Sigma_1 \mid \Sigma_2 \vdash \lambda\beta_1 :: \kappa_1. ... \lambda\beta_n :: \kappa_n.k_1 \sqcup ... \sqcup k_m <_C \lambda\beta_1' :: \kappa_1. ... \lambda\beta_n' :: \kappa_n.k_1' \sqcup ... \sqcup k_{m'}'} \text{[CO-Abs]}$$

$$\frac{\ell_1 <_L \ell_2}{\Sigma_1 \mid \Sigma_2 \vdash \ell_1 <_C \ell_2} \text{[CO-Lat]} \qquad \frac{}{\Sigma_1 \mid \Sigma_2 \vdash \ell <_C \beta\ c_1\ \dots\ c_{a_\beta}} \text{[CO-Lat-App]}$$

$$\frac{\Sigma_1 \mid \Sigma_2 \vdash (\beta, c1, ..., c_{a_\beta}) <_C (\beta', c_1', ..., c_{a_{\beta'}}')}{\Sigma_1 \mid \Sigma_2 \vdash \beta\ c_1\ \dots\ c_{a_\beta} <_C \beta'\ c_1'\ \dots\ c_{a_{\beta'}}'} \text{[CO-App]}$$

$$\frac{\Sigma_1 \mid \Sigma_2 \vdash x_1 <_C y_1}{\Sigma_1 \mid \Sigma_2 \vdash (x_1, ..., x_m) <_C (y_1, ..., y_n)} \text{[CO-Lex}_1\text{]}$$

$$\frac{n \geqslant 1}{\Sigma_1 \mid \Sigma_2 \vdash () <_C (y_1, ..., y_n)} \text{[CO-Lex}_2\text{]}$$

$$\frac{\Sigma_1 \mid \Sigma_2 \vdash x_1 \equiv_\alpha y_1 \qquad \Sigma_1 \mid \Sigma_2 \vdash (x_2, ..., x_m) <_C (y_2, ..., y_n)}{\Sigma_1 \mid \Sigma_2 \vdash (x_1, ..., x_m) <_C (y_1, ..., y_n)} \text{[CO-Lex}_3\text{]}$$

Figure 3.2.: Canonical order on dependency terms ($\Sigma_1 \mid \Sigma_2 \vdash \xi_1 <_C \xi_2$)

3.1.3. Canonical Ordering

The canonical forms introduced in the previous subsection are unique up to the order of the operands of the $\sqcup$-terms and further reduction using the join operation of the underlying lattice.

In order to impose a unique order on the atoms in canonical forms, we need to fix an arbitrary (strict) total order $<_L$ on the elements of the lattice $\mathcal{L}$. For example, in the case of finite and countably infinite lattices, we can simply identify every element of $\mathcal{L}$ with a unique natural number through an injective function $\iota : \mathcal{L} \to \mathbb{N}$ and define $x <_L y \iff \iota(x) <_\mathbb{N} \iota(y)$ for all $x, y \in \mathcal{L}$.

We extend this strict total order to canonical forms and atoms using the rules in figure 3.2. A judgment $\Sigma_1 \mid \Sigma_2 \vdash \xi_1 <_C \xi_2$ states that ξ_1 under the environment Σ_1 is less than ξ_2 under the environment Σ_2. That also means that the canonical order is only defined on canonical forms of the same sort. The environments may only differ in the names of the variables, but not in the sorts they are mapped to. That means, variables are identified by their position in the environment which indicates the level of the lambda abstraction introducing a variable, similar to how De Bruijn indices work.

The inference rules of the order relation make use of judgments $\Sigma_1 \mid \Sigma_2 \vdash \xi_1 \equiv_\alpha \xi_2$ which hold whenever ξ_1 and $[\Sigma_1 / \Sigma_2]\xi_2$ are alpha-equivalent, i.e. ξ_1 and ξ_2 are alpha-equivalent after renaming all free variables in ξ_2 (given by Σ_2) to the corresponding name in Σ_1.

The strict total order $<_C$ also implies a non-strict order $\leqslant_C$. The judgment $\Sigma_1 \mid \Sigma_2 \vdash \xi_1 \leqslant_C \xi_2$ holds if and only if $\Sigma_1 \mid \Sigma_2 \vdash \xi_1 <_C \xi_2$ or $\Sigma_1 \mid \Sigma_2 \vdash \xi_1 \equiv_\alpha \xi_2$.

[CO-VAR] Variables are compared by their index in the respective environment. Since environments are extended on the right in λ-abstractions, this amounts to comparing variables by their De Bruijn index in the terms we initially started comparing. We cannot simply compare variables by name, because the order relation must be invariant under alpha-equivalence. Otherwise, equal terms such as $\lambda x.\, x$ and $\lambda y.\, y$ would not compare equal under the relation.

[CO-ABS] Normal forms are compared by first moving all abstractions in to the context and then lexicographically comparing the atoms in the bodies. Note that since both normal forms have the same type, they must have the same number of abstractions only differing in the names.

[CO-LAT] Lattice elements are compared by the (arbitrary) total order $<_L$ that was fixed in the beginning.

[CO-LAT-APP] This rule ensures that lattice atoms always compare less than application atoms.

[CO-APP] Two saturated applications are compared lexicographically. Note that if the first symbols are equal, they must be functions of the same sort, and hence all pairs of arguments of the first and second atom have the same sorts.

[CO-LEX$_1$] A sequence is lexicographically smaller than another if its first element is less than the other's first element.

[CO-LEX$_2$] The empty sequence is less than any non-empty sequence.

[CO-LEX$_3$] If the heads of two sequences are equal, their order is determined by their tails.

Definition 3.6. A $\lambda^{\sqcup}$-term $c = \lambda\beta_1 :: \kappa_1. \ldots .\lambda\beta_n :: \kappa_n.k_1 \sqcup \cdots \sqcup k_m$ in canonical form that is well-sorted under some environment Σ is said to be *canonically ordered* if

1. its atoms $k_1, \ldots, k_m$ are canonically ordered and
2. $\Sigma, \overline{\beta_i :: \kappa_i} \mid \Sigma, \overline{\beta_i :: \kappa_i} \vdash k_j \leqslant_C k_{j+1}$ for all $j \in \{1, \ldots, m-1\}$ and
3. it is irreducible, i.e. all atoms that are lattice elements have been joined, and there is only an atom that is equal to bottom if it is the only atom.

An atom k is canonically ordered if it is a lattice element, or if it is a saturated application $\beta\ c_1 \cdots c_{a_\beta}$ where each of the arguments $c_1, \ldots, c_{a_\beta}$ is canonically ordered.

At this point we would like to provide a theorem stating that two semantically equivalent $\lambda^{\sqcup}$-terms always reduce to the same canonically ordered forms. However, as mentioned earlier, this guarantee does not hold in the current formulation of the reduction system. Fixing this problem has proven to be a task beyond the scope of this thesis. Nonetheless, we will present some ideas in section 3.4.

Definition 3.7. We denote by $\llbracket \xi \rrbracket_\Sigma$ the canonically ordered term derived from ξ under the environment Σ.

We extend this definition to annotated types in the following way.

$$\begin{array}{lcl}
\llbracket \widehat{\text{unit}} \rrbracket_\Sigma & = & \widehat{\text{unit}} \\
\llbracket \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \rrbracket_\Sigma & = & \llbracket \widehat{\tau}_1 \rrbracket_\Sigma\langle\llbracket \xi_1 \rrbracket_\Sigma\rangle + \llbracket \widehat{\tau}_2 \rrbracket_\Sigma\langle\llbracket \xi_2 \rrbracket_\Sigma\rangle \\
\llbracket \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \rrbracket_\Sigma & = & \llbracket \widehat{\tau}_1 \rrbracket_\Sigma\langle\llbracket \xi_1 \rrbracket_\Sigma\rangle \times \llbracket \widehat{\tau}_2 \rrbracket_\Sigma\langle\llbracket \xi_2 \rrbracket_\Sigma\rangle \\
\llbracket \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \rrbracket_\Sigma & = & \llbracket \widehat{\tau}_1 \rrbracket_\Sigma\langle\llbracket \xi_1 \rrbracket_\Sigma\rangle \to \llbracket \widehat{\tau}_2 \rrbracket_\Sigma\langle\llbracket \xi_2 \rrbracket_\Sigma\rangle \\
\llbracket \forall \beta :: \kappa . \widehat{\tau} \rrbracket_\Sigma & = & \forall \beta :: \kappa . \llbracket \widehat{\tau} \rrbracket_{\Sigma, \beta :: \kappa}
\end{array}$$

Remark 3.8. The environment Σ in the above definition is necessary because we need to know the types of free variables in order to correctly perform η-expansion. This information has been left implicit in the preceding definitions.

3.2. Ensuring Modularity

When designing the type reconstruction algorithm we have two goals in mind. Firstly, every program typeable in the underlying type system should have a valid typing derivation in the type and effect system. This means our analysis should be a *conservative extension* of the underlying type system.

Secondly, types assigned by the analysis should be as general as possible in order to achieve modularity. Concretely, this means that a function's type must be general enough to be able to adapt to arguments with arbitrary annotations.

These two goals give rise to the notion of *fully flexible* and *fully parametric* types defined by Holdermans and Hage [10]. Koot calls these types *conservative* and *pattern* types respectively. We will adopt the latter notation.

Informally, an annotated type is a pattern type if it can be instantiated to any conservative type of the same shape and a conservative type is an analysis of an expression that is able to cope with any arguments it might depend on.

These types are conservative in a sense because they make the least assumptions about their arguments and therefore are a conservative estimate compared to other typings with less degrees of freedom.

3.2.1. Pattern Types

For a pattern type to be instantiable to any conservative type, we first need to make sure that all dependency annotations occurring in it can be instantiated to the corresponding dependency terms in a matching conservative type.

This leads to the following definition of a *pattern* in the $\lambda^{\sqcup}$-calculus. It is based on the similar definition by Koot which in turn is a special case of an earlier definition of a pattern in higher-order unification theory [3, 17].

Definition 3.9. A $\lambda^{\sqcup}$-term is a *pattern* if it is of the form $f\ \beta_1\ \cdots\ \beta_n$ where f is a free variable and $\beta_1, \ldots, \beta_n$ are distinct bound variables.

$$\frac{\beta \notin \overline{\alpha_i}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\text{unit}} \mathbin{\&} \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star} \text{[P-UNIT]}$$

$$\frac{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1 \mathbin{\&} \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_2 \mathbin{\&} \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \notin \overline{\alpha_i}, \overline{\beta_j}, \overline{\gamma_k}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[P-SUM]}$$

$$\frac{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1 \mathbin{\&} \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_2 \mathbin{\&} \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \notin \overline{\alpha_i}, \overline{\beta_j}, \overline{\gamma_k}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[P-PROD]}$$

$$\frac{\emptyset \vdash_p \widehat{\tau}_1 \mathbin{\&} \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_p \widehat{\tau}_2 \mathbin{\&} \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \notin \overline{\alpha_i}, \overline{\beta_j}, \overline{\gamma_k}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[P-ARR]}$$

Figure 3.3.: Pattern types ($\Sigma \vdash_p \widehat{\tau} \mathbin{\&} \xi \rhd \Sigma'$)

A unification problem of the form $\forall\beta_1 \cdots \beta_n.f\ \beta_1 \cdots \beta_n = \xi$ where the left-hand side is a pattern is called *pattern unification*.

Remark 3.10. A pattern unification problem $\forall\beta_1 \cdots \beta_n.f\ \beta_1 \cdots \beta_n = \xi$ has a unique most general solution, namely the substitution $[f \mapsto \lambda\beta_1. \cdots \lambda\beta_n.\xi]$ [3].

The definition of a pattern is then extended to annotated types using the rules from figure 3.3. Our definition is more precise than the ones from the previous work it is based on in that it makes explicit which variables are expected to be bound and which are free. We require that all variables with different names in the definition of these rules are distinct from each other.

Definition 3.11. An type and effect pair $\widehat{\tau} \mathbin{\&} \xi$ is a *pattern type* under the sort environment Σ if the judgment $\Sigma \vdash_p \widehat{\tau} \mathbin{\&} \xi \rhd \Sigma'$ holds for some Σ'. We call the variables in Σ *argument variables* and the variables in Σ' *pattern variables*.

In such a judgment $\Sigma \vdash_p \widehat{\tau} \mathbin{\&} \xi \rhd \Sigma'$, the context Σ contains the types of variables that have already been bound earlier on the same chain of function arrows[1]. They correspond to the annotations of preceding arguments. On the other hand, the pattern variables Σ' are those free variables of $\widehat{\tau}$ that are not part of Σ and that will later be assigned in order to instantiate a pattern to a conservative type.

One small but important detail in this definition is that, strictly speaking, the ξ in a pattern type is not always a pattern as defined in definition 3.9. Even though it is of the form $f\ \beta_1 \cdots \beta_n$, and the $\beta_1, \ldots, \beta_n$ are always bound higher up in the type, f might also be bound if the pattern occurs as an argument of another function pattern.

However, as we will later see when discussing the type reconstruction algorithm, f will be instantiated with a fresh variable, and thereby become free before pattern unification is performed.

[1] By a chain of function arrows we mean a type of the form $\widehat{\tau}_1 \to \ldots \to \widehat{\tau}_n$.

Example 3.12. A simple pattern type with the pattern variables $\beta :: \star \Rightarrow \star$ and $\beta' :: \star \Rightarrow \star \Rightarrow \star$ is

$$\forall \beta_1 :: \star.\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall \beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle$$

as witnessed by the following derivation tree (where sort signatures have been omitted as they are the same as in the above type signature).

$$\dfrac{\emptyset \vdash_p \widehat{\text{unit}}\ \&\ \beta_1 \rhd \beta_1 \qquad \dfrac{\emptyset \vdash_p \widehat{\text{unit}}\ \&\ \beta_2 \rhd \beta_2 \qquad \beta_1, \beta_2 \vdash_p \widehat{\text{unit}}\ \&\ \beta'\ \beta_1\ \beta_2 \rhd \beta'}{\beta_1 \vdash_p \forall \beta_2.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle\ \&\ \beta\ \beta_1 \rhd \beta, \beta'}}{\emptyset \vdash_p \forall \beta_1.\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall \beta_2.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle\ \&\ \beta'' \rhd \beta'', \beta, \beta'}$$

Note that since β_1 is quantified on the same function arrow chain, it is passed on to the second function arrow. However, it is not propagated into the second argument.

In general, annotations on the return type may depend on the annotations of all previous arguments while annotations of the arguments may not. This prevents any dependency between the annotations of arguments and guarantees that they are as permissive as possible.

This is also why pattern variables in a covariant position are passed on to the next higher level while pattern variables in arguments are quantified in the enclosing function arrow. This allows the caller of a function to instantiate the effects of the parameters to the actual arguments.

Example 3.13. As another example, consider the underlying type (unit $\to$ unit) $\to$ unit and a matching pattern type

$$\forall \beta_1 :: \star.\forall \beta_2 :: \star \Rightarrow \star.(\forall \beta_3 :: \star.\text{unit}\langle\beta_3\rangle \to \text{unit}\langle\beta_2\ \beta_3\rangle)\langle\beta_1\rangle \to \text{unit}\langle\beta_4\ \beta_1\ \beta_2\rangle$$

which contains a single pattern variable $\beta_4 :: \star \Rightarrow (\star \Rightarrow \star) \Rightarrow \star$. This is an example of a higher-order effect operator. It arises because a general analysis of a higher order function is abstracted over the analysis of its argument, which itself is represented as an effect operator.

The pattern variables β_1 and β_2 of the function parameter are quantified on the outer function arrow. Informally, this is because the pattern must be valid for all possible arguments it might receive, hence the universal quantification.

The following lemma characterizes how the free variables in a pattern type relate to the argument and pattern variables of the corresponding judgment.

Lemma 3.14. *If for an annotated type $\widehat{\tau}$, a dependency term ξ and sort environments Σ, Σ' the judgment $\Sigma \vdash_p \widehat{\tau}\ \&\ \xi \rhd \Sigma'$ holds, then*

1. $\text{dom}(\Sigma) \cap \text{dom}(\Sigma') = \emptyset$,
2. $\text{fav}(\widehat{\tau}) \cup \text{fav}(\xi) = \text{dom}(\Sigma) \cup \text{dom}(\Sigma')$, *and*
3. $\Sigma, \Sigma' \vdash_s \xi : \star$ *and* $\Sigma, \Sigma' \vdash_{\text{wft}} \widehat{\tau}$.

Proof. See appendix, page 118. □

In particular, this lemma lets us conclude that for patterns occurring in argument positions, i.e. where the sort environment Σ is empty, the free variables of the pattern type are exactly the pattern variables of that type.

Moreover, pattern types are unique up to alpha-equivalence, as the following lemma demonstrates.

Lemma 3.15. *Suppose $\widehat{\tau}$ and $\widehat{\tau}'$ are annotated types and ξ and ξ' are dependency terms such that $\lfloor\widehat{\tau}\rfloor = \lfloor\widehat{\tau}'\rfloor$ and $\Sigma \vdash_p \widehat{\tau} \,\&\, \xi \rhd \Sigma'$ and $\Sigma \vdash_p \widehat{\tau}' \,\&\, \xi' \rhd \Sigma''$ for some Σ, Σ' and Σ''. Then Σ' and Σ'' are equal up to renaming, $\widehat{\tau} \equiv_\alpha [\Sigma' \,/\, \Sigma'']\widehat{\tau}'$ and $\xi \equiv_\alpha [\Sigma' \,/\, \Sigma'']\xi'$, i.e. they are alpha-equivalent up to the names of the pattern variables.*[2]

Proof. See appendix, page 120. □

3.2.2. Conservative Types

As we stated earlier, a conservative function type makes the least assumptions over its arguments. Formally, this means that arguments of conservative functions are pattern types. We will later see that a pattern type can be instantiated to any conservative type of the same shape. On the other hand, non-functional conservative types are not constrained in their annotations. These characteristics are captured by the following definition based on *conservative types* by Koot and *fully flexible types* by Holdermans and Hage.

Definition 3.16. An annotated type $\widehat{\tau}$ is *conservative* if

1. $\widehat{\tau} = \widehat{\text{unit}}$, or
2. $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ and both $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative, or
3. $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle$ and both $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative, or
4. $\widehat{\tau} = \forall\overline{\beta_j :: \kappa_j}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ and both
   a) $\emptyset \vdash_p \widehat{\tau}_1 \,\&\, \xi_1 \rhd \overline{\beta_j :: \kappa_j}$ and
   b) $\widehat{\tau}_2$ is conservative.

A type and effect pair $\widehat{\tau} \,\&\, \xi$ is *conservative* if $\widehat{\tau}$ is *conservative.*

A type and effect environment $\widehat{\Gamma}$ is *conservative* if for all $x \in \text{dom}(\widehat{\Gamma})$, $\widehat{\Gamma}(x)$ is conservative.

Example 3.17. The following type signature for the function f is a conservative type that takes the function type from example 3.12 as an argument.

$$\begin{aligned} f : {} & \forall\beta :: \star \Rightarrow \star.\forall\beta' :: \star \Rightarrow \star \Rightarrow \star.\forall\beta_3 :: \star. \\ & \quad (\forall\beta_1 :: \star.\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle)\langle\beta_3\rangle \\ & \quad \to \widehat{\text{unit}}\langle\beta_3 \sqcup \beta\ \bot \sqcup \beta'\ \bot\ \ell\rangle \,\&\, \bot \end{aligned}$$

[2] The notation $[\Sigma'/\Sigma'']$ is used to denote a substitution that replaces every variable in Σ'' with the corresponding variable from Σ'

Note that the pattern variables of the argument have been bound in the top-level function type. This allows callers of f to instantiate these patterns accordingly. Suppose there is another function g of the following (conservative) type.

$$g : \forall(\beta_1 :: \star).\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall(\beta_2 :: \star).\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta_2\rangle)\langle\bot\rangle \mathbin{\&} \bot$$

Since the type of g and the formal parameter of f have the same shape, we can derive the type of the application $f\ g$ by instantiating the pattern variables with the substitution $[\beta \mapsto \lambda\beta_1.\bot, \beta' \mapsto \lambda\beta_1.\lambda\beta_2.\beta_2, \beta_3 \mapsto \bot]$ obtained by performing pattern unification. Applying this substitution to the return annotation of the function f yields $\bot \sqcup (\lambda\beta_1.\,\bot)\ \bot \sqcup (\lambda\beta_1.\,\lambda\beta_2.\,\beta_2)\ \bot\ \ell$. Further normalization of this dependency term results in ℓ.

The following lemma establishes that a pattern type is always conservative.

Lemma 3.18. *Let $\widehat{\tau}$ be an annotated type, ξ a dependency term and Σ, Σ' sort environments such that $\Sigma \vdash_p \widehat{\tau} \mathbin{\&} \xi \triangleright \Sigma'$ holds. Then $\widehat{\tau}$ is conservative.*

Proof. By induction on the derivation of $\Sigma \vdash_p \widehat{\tau} \mathbin{\&} \xi \triangleright \Sigma'$.

[P-Unit] We have $\widehat{\tau} = \widehat{\text{unit}}$ which is conservative by definition.

[P-Sum] We have $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$. By induction, both $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative, and therefore $\widehat{\tau}$ is conservative.

[P-Prod] Analogous to the previous case.

[P-Arr] We have $\widehat{\tau} = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ which is conservative because $\emptyset \vdash_p \widehat{\tau}_1 \& \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ holds by assumption of [P-Arr] and $\widehat{\tau}_2$ is conservative by induction.

□

Moreover, substitutions preserve conservativeness.

Lemma 3.19. *Let $\widehat{\tau}$ be a conservative type and θ a substitution on annotation variables. Then $\theta\widehat{\tau}$ is also conservative.*

Proof. By induction on $\widehat{\tau}$. Since $\widehat{\tau}$ is conservative, one of the following cases holds

$\widehat{\tau} = \widehat{\textbf{unit}}$ Trivial, since $\theta\widehat{\text{unit}} = \widehat{\text{unit}}$.

$\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ By induction, $\theta\widehat{\tau}_1$ and $\theta\widehat{\tau}_2$ are conservative. Therefore, $\theta(\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle) = \theta\widehat{\tau}_1\langle\theta\xi_1\rangle + \theta\widehat{\tau}_2\langle\theta\xi_2\rangle$ is also conservative.

$\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle$ Analogous to the previous case.

$\widehat{\tau} = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ We have $\emptyset \vdash_p \widehat{\tau}_1 \mathbin{\&} \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\widehat{\tau}_2$ is conservative. By lemma 3.14, $\text{fav}(\widehat{\tau}_1) \cup \text{fav}(\xi_1) = \overline{\beta_j}$. Define $\theta' := \theta \restriction_{\text{dom}(\theta)\setminus\overline{\beta_j}}$. Then, $\theta\widehat{\tau} = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \theta'\widehat{\tau}_2\langle\theta'\xi_2\rangle$. By induction, $\theta'\widehat{\tau}_2$ is conservative. Hence $\theta\widehat{\tau}$, is also conservative.

□

$$\frac{\beta \text{ fresh}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \text{unit} : \widehat{\text{unit}} \;\&\; \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star} \text{[C-UNIT]}$$

$$\frac{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 : \widehat{\tau}_1 \;\&\; \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_2 : \widehat{\tau}_2 \;\&\; \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \text{ fresh}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 + \tau_2 : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[C-SUM]}$$

$$\frac{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 : \widehat{\tau}_1 \;\&\; \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_2 : \widehat{\tau}_2 \;\&\; \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \text{ fresh}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 \times \tau_2 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[C-PROD]}$$

$$\frac{\vdash_c \tau_1 : \widehat{\tau}_1 \;\&\; \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}} \qquad \overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_c \tau_2 : \widehat{\tau}_2 \;\&\; \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}} \qquad \beta \text{ fresh}}{\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 \to \tau_2 : \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\gamma_k :: \kappa_{\gamma_k}}} \text{[C-ARR]}$$

Figure 3.4.: Type completion ($\Sigma \vdash_c \tau : \widehat{\tau} \;\&\; \xi \rhd \Sigma'$)

3.2.3. Type Completion

We can extend the previous definition of pattern types to the type completion relation shown in figure 3.4. It relates every underlying type τ with a pattern type $\widehat{\tau}$ such that $\widehat{\tau}$ erases to τ. It is defined through judgments $\Sigma \vdash_c \tau : \widehat{\tau} \;\&\; \xi \rhd \Sigma'$ with the meaning that under the sort environment Σ, τ is completed to the annotated type $\widehat{\tau}$ and the effect ξ containing the pattern variables Σ'. The completion relation can also be interpreted as a function taking Σ and τ as arguments and returning $\widehat{\tau}$, ξ and Σ'.

The following lemma establishes basic correctness properties of the type completion relation. In particular, the completed type is always a pattern type.

Lemma 3.20. *Suppose there are Σ, τ, $\widehat{\tau}$, ξ and Σ' such that $\Sigma \vdash_c \tau : \widehat{\tau} \;\&\; \xi \rhd \Sigma'$ holds, then*

1. *$\tau = \lfloor \widehat{\tau} \rfloor$, and*
2. *$\widehat{\tau} \;\&\; \xi$ is a pattern type, i.e. $\Sigma \vdash_p \widehat{\tau} \;\&\; \xi \rhd \Sigma'$ holds*

Proof. By induction on the derivation tree of $\Sigma \vdash_c \tau : \widehat{\tau} \;\&\; \xi \rhd \Sigma'$.

[C-UNIT] By definition of the rule [C-UNIT], $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \text{unit}$, $\widehat{\tau} = \widehat{\text{unit}}$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star$.

By definition, $\lfloor \widehat{\text{unit}} \rfloor = \text{unit}$ and by rule [P-UNIT], $\Sigma \vdash_p \widehat{\tau} \;\&\; \xi \rhd \Sigma'$ holds.

[C-SUM] By definition of the rule [C-SUM], $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \tau_1 + \tau_2$, $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}$.

The premises are $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 : \widehat{\tau}_1 \;\&\; \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_2 : \widehat{\tau}_2 \;\&\; \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}}$.

By induction, we have $\lfloor \widehat{\tau}_1 \rfloor = \tau_1$ and $\lfloor \widehat{\tau}_2 \rfloor = \tau_1$, and therefore $\lfloor \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \rfloor = \lfloor \widehat{\tau}_1 \rfloor + \lfloor \widehat{\tau}_2 \rfloor = \tau_1 + \tau_2$. Moreover, we get $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1 \;\&\; \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_2 \;\&\; \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}}$ from which we can conclude $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \beta\ \overline{\alpha_i} \rhd \beta :: \overline{\kappa_{\alpha_i}} \Rightarrow \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}$ by applying rule [P-SUM].

[C-PROD] Analogous to [C-SUM].

[C-ARR] In this case we have $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \tau_1 \to \tau_2$, $\widehat{\tau} = \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \overline{\gamma_k :: \kappa_{\gamma_k}}$.

The premises are $\emptyset \vdash_c \tau_1 : \widehat{\tau}_1 \& \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_c \tau_2 : \widehat{\tau}_2 \& \xi_2 \triangleright \overline{\gamma_k :: \kappa_{\gamma_k}}$.

By induction, we have $\lfloor \widehat{\tau}_1 \rfloor = \tau_1$ and $\lfloor \widehat{\tau}_2 \rfloor = \tau_1$, and therefore

$$\lfloor \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \rfloor = \lfloor \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \rfloor = \lfloor \widehat{\tau}_1 \rfloor \to \lfloor \widehat{\tau}_2 \rfloor = \tau_1 \to \tau_2.$$

Moreover, we get $\emptyset \vdash_p \widehat{\tau}_1 \ \&\ \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_p \widehat{\tau}_2 \ \&\ \xi_2 \triangleright \overline{\gamma_k :: \kappa_{\gamma_k}}$ from which we can conclude by [P-ARR]

$$\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \ \&\ \beta\ \overline{\alpha_i} \triangleright \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \overline{\gamma_k :: \kappa_{\gamma_k}}.$$

□

Lastly, we revisit the examples from the previous sections and show how a pattern type can be mechanically derived from an underlying type.

Example 3.21. In example 3.12 we presented a pattern type for the underlying type unit → unit → unit. Using the type completion relation, we can derive the pattern type without having to guess. This is because the components $\widehat{\tau}$, ξ and Σ' in a judgment $\Sigma \vdash_c \tau : \widehat{\tau} \,\&\, \xi \triangleright \Sigma'$ are uniquely determined by Σ and τ from looking at the syntax alone.

$$\dfrac{\emptyset \vdash_c \mathrm{unit} : \widehat{\mathrm{unit}} \ \&\ \beta_1 \triangleright \beta_1 \qquad \dfrac{\emptyset \vdash_c \mathrm{unit} : \widehat{\mathrm{unit}} \ \&\ \beta_2 \triangleright \beta_2 \qquad \beta_1, \beta_2 \vdash_c \mathrm{unit} : \widehat{\mathrm{unit}} \ \&\ \beta'\ \beta_1\ \beta_2 \triangleright \beta'}{\beta_1 \vdash_c \mathrm{unit} \to \mathrm{unit} : \forall \beta_2.\widehat{\mathrm{unit}}\langle\beta_2\rangle \to \widehat{\mathrm{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle \ \&\ \beta\ \beta_1 \triangleright \beta, \beta'}}{\emptyset \vdash_c \mathrm{unit} \to \mathrm{unit} \to \mathrm{unit} : \forall \beta_1.\widehat{\mathrm{unit}}\langle\beta_1\rangle \to (\forall \beta_2.\widehat{\mathrm{unit}}\langle\beta_2\rangle \to \widehat{\mathrm{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle \ \&\ \beta_3 \triangleright \beta, \beta', \beta_3}$$

The resulting pattern type contains three pattern variables, $\beta :: \star \Rightarrow \star$, $\beta' :: \star \Rightarrow \star \Rightarrow \star$ and $\beta_3 :: \star$. Since the initial sort environment was empty, these are also the only free variables of the pattern type.

A more involved example is the completion of the underlying type ((unit × unit → unit + unit) → unit) → unit. We omit the derivation tree due to its size.

$$\begin{array}{l}
\forall \beta_5\ \beta_4'. \\
\quad (\forall \beta_4\ \beta_1'\ \beta_2'\ \beta_3'. \\
\quad\quad (\forall \beta_1\ \beta_2\ \beta_3.(\mathrm{unit}\langle\beta_1\rangle \times \mathrm{unit}\langle\beta_2\rangle)\langle\beta_3\rangle \\
\quad\quad\quad\quad\quad\quad \to (\mathrm{unit}\langle\beta_1'\ \beta_1\ \beta_2\ \beta_3\rangle + \mathrm{unit}\langle\beta_2'\ \beta_1\ \beta_2\ \beta_3\rangle)\langle\beta_3'\ \beta_1\ \beta_2\ \beta_3\rangle)\langle\beta_4\rangle \\
\quad\quad \to \mathrm{unit}\langle\beta_4'\ \beta_4\ \beta_1'\ \beta_2'\ \beta_3'\rangle)\langle\beta_5\rangle \\
\quad \to \mathrm{unit}\langle\beta_6\ \beta_5\ \beta_4'\rangle \ \&\ \beta_7
\end{array}$$

Here, only $\beta_6 :: \star \Rightarrow (\star \Rightarrow (\star \Rightarrow \star \Rightarrow \star \Rightarrow \star) \Rightarrow (\star \Rightarrow \star \Rightarrow \star \Rightarrow \star) \Rightarrow (\star \Rightarrow \star \Rightarrow \star \Rightarrow \star) \Rightarrow \star) \Rightarrow \star$ and $\beta_7 :: \star$ are pattern variables because the return annotation of the topmost function arrow and the annotation of the function itself are the only things in

this type that may instantiated in order to adapt to a conservative type. Functions that are consumed as arguments must always be as general as possible.

The complicated sort of β_6 arises from the fact that the outermost function might call its argument, which in turn needs to be able to adapt to any function it gets applied to.

3.2.4. Least Types

Based on the type completion relation we can define least type completions. These are conservative types that are subtypes of all other conservative types of the same shape. Therefore, all annotations occurring in positive positions on the top level function arrow chain must also be least. We do not need to consider arguments here because those are by definition equal up to alpha-conversion due to being pattern types.

Since our annotations are based on bounded lattices, we know by definition that there are least elements. The following definition gives a way of constructing terms in the $\lambda^{\sqcup}$-calculus corresponding to these least elements.

Definition 3.22 (least annotations)**.** We define the *least annotation term* of sort κ recursively as

$$\begin{aligned} \bot_\star &= \bot \\ \bot_{\kappa_1 \Rightarrow \kappa_2} &= \lambda\beta : \kappa_1 . \bot_{\kappa_2}. \end{aligned}$$

The following lemma shows that these terms indeed correspond to the least elements of the lattices.

Lemma 3.23. *For all sorts κ and all sort environments Σ, least annotations terms $\bot_\kappa$ are*

1. *well-sorted, i.e. $\Sigma \vdash_{\mathrm{s}} \bot_\kappa : \kappa$ and*
2. *least w.r.t. the subsumption relation $\Sigma \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$.*

Proof. 1. We show by induction on κ that $\Sigma \vdash_{\mathrm{s}} \bot_\kappa : \kappa$ holds for all Σ.

$\kappa = \star$ By definition $\Sigma \vdash_{\mathrm{s}} \bot : \star$.

$\kappa = \kappa_1 \Rightarrow \kappa_2$ By the induction hypothesis, $\Sigma, \beta :: \kappa_1 \vdash_{\mathrm{s}} \bot_{\kappa_2} : \kappa_2$. Therefore we can conclude $\Sigma \vdash_{\mathrm{s}} \lambda\beta :: \kappa_1 . \bot_{\kappa_2} : \kappa_1 \Rightarrow \kappa_2$.

2. We show by induction on κ that $[\![\bot_\kappa]\!]_\rho = \bot \in V_\kappa$, i.e. that it is the bottom value of lattice V_κ for all environments ρ.

Suppose ρ is an arbitrary environment.

$\kappa = \star$ By definition, $[\![\bot_\kappa]\!]_\rho = [\![\bot]\!]_\rho = \bot \in V_\star$.

$\kappa = \kappa_1 \Rightarrow \kappa_2$ $[\![\bot_{\kappa_1 \Rightarrow \kappa_2}]\!]_\rho = [\![\lambda\beta :: \kappa_1 . \bot_{\kappa_2}]\!]_\rho = \lambda v \in V_{\kappa_1} . [\![\bot_{\kappa_2}]\!]_{\rho[\beta \mapsto v]} = \lambda v \in V_{\kappa_1} . \bot$ which is exactly the bottom value of the pointwise extension lattice $V_{\kappa_1} \to V_{\kappa_2}$.

Consequentially, for all environments ρ and annotation terms ξ of sort κ we have $[\![\bot_\kappa]\!]_\rho = \bot \sqsubseteq [\![\xi]\!]_\rho$. Since this holds in particular for all environments compatible with an arbitrary Σ, we can conclude $\Sigma \vdash_{\text{sub}} \bot_\kappa \sqsubseteq \xi$.

$\square$

Based on our definition of least annotations, we can define the least type completions as follows.

Definition 3.24 (least types)**.** We define the least completion of type τ (see figure 3.4) by substituting all free variables in the completion with the least annotation of the corresponding sort, i.e.

$$\bot_\tau = [\overline{\bot_{\kappa_i} \,/\, \beta_i}]\widehat{\tau} \text{ for } \emptyset \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \overline{\beta_i :: \kappa_i}.$$

Note that we can ignore the effect part of the type completion in the above definition because it is always $\bot_\star$. No matter which completion rules applies, there is some $\beta \in \overline{\beta_i}$ such that $\xi = \beta$ because the initial sort environment is always empty. Since we instantiate all $\overline{\beta_i}$ to the corresponding bottom value, we instantiate β to $\bot$ in particular.

We proceed by showing that least type completions are well-formed, conservative types of the correct shape.

Lemma 3.25. *For all types τ of the underlying type system,*

1. *$\bot_\tau$ is conservative,*
2. *$\Sigma \vdash_{\text{wft}} \bot_\tau$ for all sort environments Σ, i.e. $\bot_\tau$ is well-formed in any environment, and*
3. *$\lfloor \bot_\tau \rfloor = \tau$.*

Proof. Recall the definition of $\bot_\tau$:

$$\bot_\tau = [\overline{\bot_{\kappa_i} \,/\, \beta_i}]\widehat{\tau} \text{ for } \emptyset \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \overline{\beta_i :: \kappa_i}.$$

We define the substitution $\theta = [\overline{\bot_{\kappa_i} \,/\, \beta_i}] : \text{dom}(\overline{\beta_i :: \kappa_i}) \to \textbf{AnnTm}$. By lemma 3.23, the terms in the image of θ are always well-sorted.

By lemma 3.20, we have $\emptyset \vdash_p \widehat{\tau} \,\&\, \xi \rhd \overline{\beta_i :: \kappa_i}$ and $\lfloor \widehat{\tau} \rfloor = \tau$. Since θ only applies to nested annotations, but does not change the shape of the type, we can conclude $\lfloor \theta\widehat{\tau} \rfloor = \tau$. Note that $\text{fav}(\theta\beta) = \emptyset$ for all β and $\text{fav}(\widehat{\tau}) = \overline{\beta_i}$ by lemma 3.14.

By lemma 3.18, $\widehat{\tau}$ is conservative and $\overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}$. By lemma 2.28, $\emptyset \vdash_{\text{wft}} \theta\widehat{\tau}$ and by lemma 3.19, $\theta\widehat{\tau}$ is conservative. By lemma 2.22 $\Sigma \vdash_{\text{wft}} \theta\widehat{\tau}$ holds for any Σ. Since $\bot_\tau = \theta\widehat{\tau}$, this completes the proof. $\square$

We demonstrate that the given definition of least type completions indeed leads to least types with respect to the subtyping relation. For this, we need a slightly more general statement.

Lemma 3.26. *Suppose there are Σ, τ, $\widehat{\tau}$, ξ and Σ' such that $\Sigma \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \Sigma'$ holds. Furthermore, let $\theta : \mathrm{dom}(\Sigma') \to$ **AnnTm** denote a substitution defined by $\theta(\beta) = \bot_{\Sigma'(\beta)}$. Then, for all conservative $\widehat{\tau}'$ and sort environments Σ'' such that $\lfloor \widehat{\tau}' \rfloor = \tau$ and $\Sigma'' \vdash_{\mathrm{wft}} \widehat{\tau}'$, we have*

1. $\Sigma'' \vdash_{\mathrm{sub}} \theta\widehat{\tau} \leqslant \widehat{\tau}'$ *and*
2. $\Sigma'' \vdash_{\mathrm{sub}} \theta\xi \sqsubseteq \bot$.

Proof. See appendix, page 121. □

The previous lemma established that substituting the pattern variables of a pattern type in an arbitrary environment with the least annotations of the right sort always yields the least type under that environment. We now relate this statement to the definition of $\bot_\tau$.

Lemma 3.27. *For all types τ, the least type completions $\bot_\tau$ is least w.r.t. the subtyping relation $\leqslant$, i.e. for all conservative types $\widehat{\tau}$ with $\lfloor \widehat{\tau} \rfloor = \tau$ and sort environments Σ such that $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}$, we have $\Sigma \vdash_{\mathrm{sub}} \bot_\tau \leqslant \widehat{\tau}$.*

Proof. Let Σ be an arbitrary sort environment and $\widehat{\tau}'$ be a conservative type such that $\lfloor \widehat{\tau}' \rfloor = \tau$. Furthermore, there are ξ and $\widehat{\tau}$ such that $\emptyset \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \Sigma'$ holds. By lemma 3.26, we have $\Sigma \vdash_{\mathrm{sub}} \theta\widehat{\tau} \leqslant \widehat{\tau}$ where θ is the substitution as defined in the lemma. We note that this exactly the same substitution that is used in the definition of $\bot_\tau$, thereby completing the proof. □

Example 3.28. As an example, let us consider again the type ((unit × unit → unit + unit) → unit) → unit. The least completion is obtained by instantiating all its pattern variables with least annotations of the correct sort, resulting in the following type (with further reduction of dependency terms).

$$
\begin{array}{l}
\forall \beta_5\ \beta_4'.\\
\quad (\forall \beta_4\ \beta_1'\ \beta_2'\ \beta_3'.\\
\qquad (\forall \beta_1\ \beta_2\ \beta_3.(\mathrm{unit}\langle\beta_1\rangle \times \mathrm{unit}\langle\beta_2\rangle)\langle\beta_3\rangle\\
\qquad\qquad\qquad \to (\mathrm{unit}\langle\beta_1'\ \beta_1\ \beta_2\ \beta_3\rangle + \mathrm{unit}\langle\beta_2'\ \beta_1\ \beta_2\ \beta_3\rangle)\langle\beta_3'\ \beta_1\ \beta_2\ \beta_3\rangle)\langle\beta_4\rangle\\
\qquad \to \mathrm{unit}\langle\beta_4'\ \beta_4\ \beta_1'\ \beta_2'\ \beta_3'\rangle)\langle\beta_5\rangle\\
\quad \to \mathrm{unit}\langle\bot\rangle \,\&\, \bot
\end{array}
$$

3.3. Reconstruction Algorithm

Having defined all the preliminaries, we can now move on to the type reconstruction algorithm that is performing the actual analysis. At its core lies the algorithm $\mathcal{R}$ shown in figure 3.5. The input of the algorithm is a triple $(\widehat{\Gamma}, \Sigma, t)$ consisting of a well-typed source term t, a type and effect environment $\widehat{\Gamma}$ providing the types and effects of the free term variables in t and a sort environment Σ mapping each free annotation variable

in scope to its sort. It returns a triple $\widehat{t} : \widehat{\tau} \,\&\, \xi$ consisting of an elaborated term $\widehat{t}$ in the target language (that erases to the source term t), an annotated type $\widehat{\tau}$ and an effect ξ such that $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ holds. In the definition of $\mathcal{R}$, we write Γ instead of $\widehat{\Gamma}$ because we are only dealing with one kind of type environment.

The algorithm relies on the invariant that all types in the type environment and the inferred type must be conservative. In the version by Koot, all inferred effects (including those nested as annotations in types) had to be canonically ordered as well. But as it turned out that this canonically ordered form was not enough for deciding semantic equality, we lifted this requirement. We still marked those places in the algorithm where canonicalization would have occurred, but the actual effects of this operation do not matter as long as the dependency terms remain equivalent.

Moreover, we will later present a partial proof that $\mathcal{R}$ computes the least conservative type and effect that can be assigned to the source term, under some restrictions.

The two simplest cases of the algorithm are variables and the unit constructor. The former are simply looked up in the environment, and the latter receives the least possible effect $\bot$.

In the case of annotations $\mathrm{ann}_\ell(t)$, the required annotation ℓ is joined with the effect of the term t. This ensures that the resulting effect is at least as large as t, but not larger than necessary.

Similarly, the implementation for $\mathrm{seq}\ t_1\ t_2$ simply joins the effects of the expression t_1 being evaluated and the expression t_2 being returned.

The effects of product and sum constructors become the annotations in the corresponding types. In the latter case, the part of the sum that is not given is assigned the least possible type and effect. Note that regardless of the effects of the components, the effect of the newly created constructor is always bottom.

When eliminating a product, it must first be evaluated and therefore the top level effect is always propagated to the result and combined with the effect of the component we are projecting. The effect of the unused component is thrown away.

In contrast to the construction of a product, the analysis of projections must deal with arbitrary annotations on the product. Even though products are always created with the effect $\bot$, that effect could be enlarged directly through annotations $\mathrm{ann}_\ell(t)$ or indirectly through joining two branches of a case statement.

Case elimination of sums is a bit more involved. The resulting effect is the least upper bound of the effects of both branches and the effect of the sum itself. The types and effects of the alternatives of the sum are made known to the branches when reconstructing their types. Whether or not the effect of a value stored in a sum is part of the resulting effect therefore depends on whether the corresponding case branch makes use of this value.

Furthermore, we must ensure that all nested annotations of the types of the branches are compatible. This is done by defining the least upper bound $\sqcup$ on types as shown in figure 3.6.

The algorithm for computing the least upper bound of types requires that both types are conservative, have the same shape and use the same names for bound variables. The latter can be trivially ensured by performing α-conversion while the former two

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEffEnv} \times \mathbf{SortEnv} \times \mathbf{Tm} \to \widehat{\mathbf{Tm}} \times \widehat{\mathbf{Ty}} \times \mathbf{AnnTm} \\
\mathcal{R}(\Gamma; \Sigma; x) = x : \Gamma(x) \\
\mathcal{R}(\Gamma; \Sigma; ()) = () : \widehat{\text{unit}} \;\&\; \bot \\
\mathcal{R}(\Gamma; \Sigma; \text{ann}_\ell(t)) = \\
\quad \textbf{let } \widehat{t} : \widehat{\tau} \;\&\; \xi = \mathcal{R}(\Gamma; \Sigma; t) \\
\quad \textbf{in } \text{ann}_\ell(\widehat{t}) : \widehat{\tau} \;\&\; [\![\xi \sqcup \ell]\!]_\Sigma \\
\mathcal{R}(\Gamma; \Sigma; \text{seq } t_1 \; t_2) = \\
\quad \textbf{let } \widehat{t_1} : \widehat{\tau}_1 \;\&\; \xi_1 = \mathcal{R}(\Gamma; \Sigma; t_1) \\
\qquad \widehat{t_2} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma; \Sigma; t_2) \\
\quad \textbf{in } \text{seq } \widehat{t_1} \; \widehat{t_2} : \widehat{\tau}_2 \;\&\; [\![\xi_1 \sqcup \xi_2]\!]_\Sigma \\
\mathcal{R}(\Gamma; \Sigma; (t_1, t_2)) = \\
\quad \textbf{let } \widehat{t_1} : \widehat{\tau}_1 \;\&\; \xi_1 = \mathcal{R}(\Gamma; \Sigma; t_1) \\
\qquad \widehat{t_2} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma; \Sigma; t_2) \\
\quad \textbf{in } (\widehat{t_1}, \widehat{t_2}) : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \;\&\; \bot \\
\mathcal{R}(\Gamma; \Sigma; \text{inl}_{\tau_2}(t)) = \\
\quad \textbf{let } \widehat{t} : \widehat{\tau}_1 \;\&\; \xi_1 = \mathcal{R}(\Gamma; \Sigma; t) \\
\quad \textbf{in } \text{inl}_{\tau_2}(\widehat{t}) : \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \;\&\; \bot \\
\mathcal{R}(\Gamma; \Sigma; \text{inr}_{\tau_1}(t)) = \\
\quad \textbf{let } \widehat{t} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma; \Sigma; t) \\
\quad \textbf{in } \text{inr}_{\tau_1}(\widehat{t}) : \bot_{\tau_1}\langle \bot \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \;\&\; \bot \\
\mathcal{R}(\Gamma; \Sigma; \text{proj}_i(t)) = \\
\quad \textbf{let } \widehat{t} : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \;\&\; \xi = \mathcal{R}(\Gamma; \Sigma; t) \\
\quad \textbf{in } \text{proj}_i(\widehat{t}) : \widehat{\tau}_i \;\&\; [\![\xi \sqcup \xi_i]\!]_\Sigma \\
\mathcal{R}(\Gamma; \Sigma; \textbf{ case } t_1 \textbf{ of } \{ \text{inl}(x) \to t_2; \; \text{inr}(y) \to t_3 \}) = \\
\quad \textbf{let } \widehat{t_1} : \widehat{\tau}\langle \xi \rangle + \widehat{\tau}'\langle \xi' \rangle \;\&\; \xi_1 = \mathcal{R}(\Gamma; \Sigma; t_1) \\
\qquad \widehat{t_2} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma, x : \widehat{\tau} \;\&\; \xi; \Sigma; t_2) \\
\qquad \widehat{t_3} : \widehat{\tau}_3 \;\&\; \xi_3 = \mathcal{R}(\Gamma, y : \widehat{\tau}' \;\&\; \xi'; \Sigma; t_3) \\
\quad \textbf{in case } \widehat{t_1} \textbf{ of } \{ \text{inl}(x) \to \widehat{t_2}; \text{inr}(y) \to \widehat{t_3} \} \\
\qquad : [\![\widehat{\tau}_2 \sqcup \widehat{\tau}_3]\!]_\Sigma \;\&\; [\![\xi_1 \sqcup \xi_2 \sqcup \xi_3]\!]_\Sigma \\
\mathcal{R}(\Gamma; \Sigma; \lambda x : \tau_1 . t) = \\
\quad \textbf{let } \widehat{\tau}_1 \;\&\; \beta \rhd \overline{\beta_i :: \kappa_i} = \mathcal{C}([\,]; \tau_1) \\
\qquad \Gamma' = \Gamma, x : \widehat{\tau}_1 \;\&\; \beta \\
\qquad \Sigma' = \Sigma, \overline{\beta_i :: \kappa_i} \\
\qquad \widehat{t} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma'; \Sigma'; t) \\
\quad \textbf{in } \Lambda \overline{\beta_i :: \kappa_i} . \lambda x : \widehat{\tau}_1 \;\&\; \beta . \widehat{t} \\
\qquad : \forall \overline{\beta_i :: \kappa_i} . \widehat{\tau}_1\langle \beta \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \;\&\; \bot \\
\mathcal{R}(\Gamma; \Sigma; t_1 \; t_2) = \\
\quad \textbf{let } \widehat{t_1} : \widehat{\tau}_1 \;\&\; \xi_1 = \mathcal{R}(\Gamma; \Sigma; t_1) \\
\qquad \widehat{t_2} : \widehat{\tau}_2 \;\&\; \xi_2 = \mathcal{R}(\Gamma; \Sigma; t_2) \\
\qquad \widehat{\tau}_2'\langle \beta \rangle \to \widehat{\tau}\langle \xi \rangle \rhd \overline{\beta_i} = \mathcal{I}(\widehat{\tau}_1) \\
\qquad \theta = [\beta \mapsto \xi_2] \circ \mathcal{M}([\,]; \widehat{\tau}_2'; \widehat{\tau}_2) \\
\quad \textbf{in } \widehat{t_1} \; \langle \overline{\theta \beta_i} \rangle \; \widehat{t_2} : [\![\theta \widehat{\tau}]\!]_\Sigma \;\&\; [\![\xi_1 \sqcup \theta \xi]\!]_\Sigma \\
\mathcal{R}(\Gamma; \Sigma; \mu x : \tau . t) = \\
\quad \textbf{do } i; \widehat{\tau}_0 \;\&\; \xi_0 \leftarrow 0; \bot_\tau \;\&\; \bot \\
\qquad \textbf{repeat } \widehat{t}_{i+1} : \widehat{\tau}_{i+1} \;\&\; \xi_{i+1} \\
\qquad\qquad \leftarrow \mathcal{R}(\Gamma, x : \widehat{\tau}_i \;\&\; \xi_i; \Sigma; t) \\
\qquad\quad i \leftarrow i + 1 \\
\qquad \textbf{until } (\widehat{\tau}_{i-1} \equiv \widehat{\tau}_i \wedge \xi_{i-1} \equiv \xi_i) \\
\qquad \textbf{return } (\mu x : \widehat{\tau}_i \;\&\; \xi_i . \widehat{t}) : \widehat{\tau}_i \;\&\; \xi_i
\end{array}
$$

Figure 3.5.: Type reconstruction algorithm ($\mathcal{R}$)

$$
\begin{array}{lll}
\multicolumn{3}{l}{\sqcup : \widehat{\mathbf{Ty}} \times \widehat{\mathbf{Ty}} \to \widehat{\mathbf{Ty}}} \\
\widehat{\text{unit}} & \sqcup \; \widehat{\text{unit}} & = \widehat{\text{unit}} \\
(\widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle) & \sqcup \; (\widehat{\tau}_1'\langle \xi_1' \rangle + \widehat{\tau}_2'\langle \xi_2' \rangle) & = (\widehat{\tau}_1 \sqcup \widehat{\tau}_1')\langle \xi_1 \sqcup \xi_1' \rangle + (\widehat{\tau}_2 \sqcup \widehat{\tau}_2')\langle \xi_2 \sqcup \xi_2' \rangle \\
(\widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle) & \sqcup \; (\widehat{\tau}_1'\langle \xi_1' \rangle \times \widehat{\tau}_2'\langle \xi_2' \rangle) & = (\widehat{\tau}_1 \sqcup \widehat{\tau}_1')\langle \xi_1 \sqcup \xi_1' \rangle \times (\widehat{\tau}_2 \sqcup \widehat{\tau}_2')\langle \xi_2 \sqcup \xi_2' \rangle \\
(\widehat{\tau}_1\langle \beta \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle) & \sqcup \; (\widehat{\tau}_1\langle \beta \rangle \to \widehat{\tau}_2'\langle \xi_2' \rangle) & = \widehat{\tau}_1\langle \beta \rangle \to (\widehat{\tau}_2 \sqcup \widehat{\tau}_2')\langle \xi_2 \sqcup \xi_2' \rangle \\
(\forall \beta :: \kappa . \widehat{\tau}) & \sqcup \; (\forall \beta :: \kappa . \widehat{\tau}') & = \forall \beta :: \kappa . \widehat{\tau} \sqcup \widehat{\tau}'
\end{array}
$$

Figure 3.6.: Least upper bound of types ($\sqcup$)

$$\begin{aligned}
&\mathcal{C} : \mathbf{SortEnv} \to \widehat{\mathbf{Ty}} \times \mathbf{AnnTm} \times \mathbf{SortEnv} \\
&\mathcal{C}(\Sigma; \tau) = \widehat{\tau} \,\&\, \xi \rhd \overline{\beta_i :: \kappa_i} \text{ } \mathbf{where} \text{ } \Sigma \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \overline{\beta_i :: \kappa_i}
\end{aligned}$$

Figure 3.7.: Completion algorithm ($\mathcal{C}$)

$$\begin{aligned}
&\mathcal{I} : \widehat{\mathbf{Ty}} \to \widehat{\mathbf{Ty}} \times \mathbf{SortEnv} \\
&\mathcal{I}(\forall \beta :: \kappa.\widehat{\tau}) = \mathbf{let} \ \widehat{\tau}' \rhd \Sigma = \mathcal{I}(\widehat{\tau}) \\
&\qquad\qquad\qquad\qquad \beta' \textit{ be fresh} \\
&\qquad\qquad\qquad\quad \mathbf{in} \ [\beta \mapsto \beta'](\widehat{\tau}') \rhd \beta' :: \kappa, \Sigma \\
&\mathcal{I}(\widehat{\tau}) \qquad\qquad = \widehat{\tau} \rhd [\,]
\end{aligned}$$

Figure 3.8.: Instantiation algorithm ($\mathcal{I}$)

requirements are fulfilled by how this function is used in $\mathcal{R}$.

The restriction to conservative types allows us to ignore functions arguments because these are always required to be pattern types, which are unique up to α-equivalence. This alleviates the need for computing a corresponding greatest lower bound of types because the algorithm only traverses covariant positions.

The handling of λ-abstractions needs the type completion algorithm $\mathcal{C}$ as shown in figure 3.7. It defers its work to the type completion relation defined earlier (see figure 3.4) which can be interpreted in a functional way.

The underlying type of the function argument is completed to a pattern type. The function body is analyzed in the presence of the newly introduced pattern variables. Note that this pattern type is also conservative, thereby preserving the invariant that the context only holds conservative types.

The inferred annotated type of the lambda abstraction universally quantifies over all pattern variables and the quantification is reflected on the term level through annotation abstractions $\Lambda\beta :: \kappa.t$.

In order to analyze function applications, we need two more auxiliary algorithms. The first one is the instantiation procedure $\mathcal{I}$ (see figure 3.8) which instantiates all top-level quantifiers with fresh annotation variables.

Secondly, we need the matching algorithm $\mathcal{M}$ (see figure 3.9) which is used to instantiate a pattern type with a conservative type of the same shape. It returns a substitution obtained by performing pattern unification on corresponding annotations.

Example 3.29. In order to provide some more insight into the inner workings of the matching algorithm, we apply it to the pattern type

$$\widehat{\tau}_1 = \forall \beta_1 :: \star.\widehat{\text{unit}}\langle \beta_1 \rangle \to (\forall \beta_2 :: \star.\widehat{\text{unit}}\langle \beta_2 \rangle \to \widehat{\text{unit}}\langle \beta' \ \beta_1 \ \beta_2 \rangle)\langle \beta \ \beta_1 \rangle$$

and the following conservative type of the same shape

$$\widehat{\tau}_2 = \forall \beta_1 :: \star.\widehat{\text{unit}}\langle \beta_1 \rangle \to (\forall \beta_2 :: \star.\widehat{\text{unit}}\langle \beta_2 \rangle \to \widehat{\text{unit}}\langle \beta_1 \sqcup \beta_2 \rangle)\langle \bot \rangle.$$

$$
\begin{array}{l}
\mathcal{M} : \mathbf{SortEnv} \times \widehat{\mathbf{Ty}} \times \widehat{\mathbf{Ty}} \to \mathbf{AnnSubst} \\
\mathcal{M}(\Sigma; \widehat{\text{unit}}; \quad \widehat{\text{unit}}) = [\,] \\
\mathcal{M}(\Sigma; \widehat{\tau}'_1\langle\beta\ \overline{\beta_i}\rangle + \widehat{\tau}'_2\langle\beta'\ \overline{\beta_i}\rangle; \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle) = [\beta \mapsto \lambda\overline{\beta_i :: \Sigma(\beta_i)}.\xi_1, \beta' \mapsto \lambda\overline{\beta_i :: \Sigma(\beta_i)}.\xi_2] \\
\qquad \circ\, \mathcal{M}(\Sigma; \widehat{\tau}'_1; \widehat{\tau}_1) \circ \mathcal{M}(\Sigma; \widehat{\tau}'_2; \widehat{\tau}_2) \\
\mathcal{M}(\Sigma; \widehat{\tau}'_1\langle\beta\ \overline{\beta_i}\rangle \times \widehat{\tau}'_2\langle\beta'\ \overline{\beta_i}\rangle; \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle) = [\beta \mapsto \lambda\overline{\beta_i :: \Sigma(\beta_i)}.\xi_1, \beta' \mapsto \lambda\overline{\beta_i :: \Sigma(\beta_i)}.\xi_2] \\
\qquad \circ\, \mathcal{M}(\Sigma; \widehat{\tau}'_1; \widehat{\tau}_1) \circ \mathcal{M}(\Sigma; \widehat{\tau}'_2; \widehat{\tau}_2) \\
\mathcal{M}(\Sigma; \widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}'_2\langle\beta'\ \overline{\beta_i}\rangle; \quad \widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi\rangle) = [\beta' \mapsto \lambda\overline{\beta_i :: \Sigma(\beta_i)}.\xi] \circ \mathcal{M}(\Sigma; \widehat{\tau}'_2; \widehat{\tau}_2) \\
\mathcal{M}(\Sigma; \forall\beta :: \kappa.\widehat{\tau}'; \quad \forall\beta :: \kappa.\widehat{\tau}) = \mathcal{M}(\Sigma, \beta :: \kappa; \widehat{\tau}'; \widehat{\tau})
\end{array}
$$

Figure 3.9.: Matching algorithm ($\mathcal{M}$)

Performing stepwise expansion results in the following derivation.

$$
\begin{array}{l}
\mathcal{M}([\,]; \forall\beta_1 :: \star.\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle; \\
\qquad \forall\beta_1 :: \star.\widehat{\text{unit}}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle) \\
= \mathcal{M}(\beta_1 :: \star; \widehat{\text{unit}}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle)\langle\beta\ \beta_1\rangle; \\
\qquad \widehat{\text{unit}}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle) \\
= [\beta \mapsto \lambda\beta_1 :: \star.\bot] \circ \mathcal{M}(\beta_1 :: \star; \forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle; \\
\quad \forall\beta_2 :: \star.\widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta_1 \sqcup \beta_2\rangle) \\
= [\beta \mapsto \lambda\beta_1 :: \star.\bot] \circ \mathcal{M}(\beta_1 :: \star, \beta_2 :: \star; \widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta'\ \beta_1\ \beta_2\rangle; \\
\quad \widehat{\text{unit}}\langle\beta_2\rangle \to \widehat{\text{unit}}\langle\beta_1 \sqcup \beta_2\rangle) \\
= [\beta \mapsto \lambda\beta_1 :: \star.\bot] \circ [\beta' \mapsto \lambda\beta_1 :: \star.\lambda\beta_2 :: \star.\beta_1 \sqcup \beta_2] \circ \mathcal{M}(\beta_1 :: \star, \beta_2 :: \star; \widehat{\text{unit}}; \widehat{\text{unit}}) \\
= [\beta \mapsto \lambda\beta_1 :: \star.\bot] \circ [\beta' \mapsto \lambda\beta_1 :: \star.\lambda\beta_2 :: \star.\beta_1 \sqcup \beta_2] \circ [\,]
\end{array}
$$

Note that similar to the least upper bound algorithm presented above, the matching procedure only traverses the covariant parts of function arrows.

We have now covered the preliminaries required to describe the analysis of function applications. The first step is to separately analyze the function being applied and the argument. Then, all top-level quantifiers of the function type are instantiated (which are exactly those quantifying over the pattern variables of the function argument). Matching the pattern type of the formal parameter with the conservative type of the actual argument then results in a substitution of the pattern variables.

The function application term is extended with annotation applications $t\ \langle\xi\rangle$ reflecting the instantiations of the quantified variable through the aforementioned substitution. This substitution is also applied to the annotated type and the effect of the functions return type in order to instantiate the "abstract" analysis of the function body to a concrete analysis specialized to the given argument.

The last defining clause of $\mathcal{R}$ is dealing with recursion. It performs a Kleene-Mycroft-iteration in order to infer a polymorphically recursive type and effect for the fixpoint construct. This is the part of the analysis that requires a computable function for deciding semantic equality of annotations. The comparison operator $\equiv$ could be implemented

in terms of the denotational semantics as outlined in the beginning of this chapter. In the next section we will discuss the issues we encountered when dealing with the Mycroft-iteration.

3.4. Problems with Earlier Approaches

Now that the canonicalization procedure and the reconstruction have been explained, we can look into the termination issues we encountered in more detail.

The problem is that there is a certain class of programs that makes the resulting type annotations grow indefinitely because none of the reduction rules currently in use apply. The most simple such program we found is the following.

$$\mu f : (\text{unit} \to \text{unit}) \to \text{unit} \to \text{unit}.\lambda g : \text{unit} \to \text{unit}.\lambda x : \text{unit}.g\ (f\ g\ x)$$

In order to find the annotated type of this term, Mycroft-iteration repeatedly infers the type of the body of the fixpoint using the previous result as the type for f in the environment. This results in the following steps.

1. The iteration is started with the least type $\bot_{(\text{unit}\to\text{unit})\to\text{unit}\to\text{unit}}$, which is

   $\forall\beta_1\ \beta_2.(\forall\beta_3.\widehat{\text{unit}}\langle\beta_3\rangle \to \widehat{\text{unit}}\langle\beta_1\ \beta_3\rangle)\langle\beta_2\rangle \to \forall\beta_4.(\widehat{\text{unit}}\langle\beta_4\rangle \to \widehat{\text{unit}}\langle\bot\rangle)\langle\bot\rangle \,\&\, \bot.$

   The return type and effect of the recursive call $f\ g\ x$ is therefore $\widehat{\text{unit}} \,\&\, \bot$. Hence, the call to g is instantiated with $\bot$ and the resulting effect of $g\ x$ is $\beta_1\ \bot \sqcup \beta_2$, in accordance with the [T-App] rule.

2. This leads to the next approximation for the type of the fixpoint, namely

   $\forall\beta_1\ \beta_2.(\forall\beta_3.\widehat{\text{unit}}\langle\beta_3\rangle \to \widehat{\text{unit}}\langle\beta_1\ \beta_3\rangle)\langle\beta_2\rangle \to \forall\beta_4.(\widehat{\text{unit}}\langle\beta_4\rangle \to \widehat{\text{unit}}\langle\beta_1\ \bot \sqcup \beta_2\rangle)\langle\bot\rangle \,\&\, \bot.$

   By the same argument as in the first step, the annotation of the return value grows to $\beta_1\ (\beta_1\ \bot \sqcup \beta_2) \sqcup \beta_2$.

The annotation continues to grow, adding more and more layers of β_1 because at the time when the normalization happens, β_1 is unknown and neither β- nor γ-reduction applies.

While using the enumerative approach to semantic equality at least made it possible to guarantee termination of the type reconstruction algorithm, the number of combinations that is currently checked is too high for practical use. For now, we see two speculative starting points for further research into alleviating this issue.

1. Currently, the universes $V_{\kappa_1 \Rightarrow \kappa_2}$ representing dependency terms of functional sorts consist of all monotone functions between the two lattices. However, there could be stronger properties that hold for such functions, due to the limited way in which they can be constructed in the $\lambda^{\sqcup}$-calculus.

   Such stronger guarantees could then serve as the foundation for additional reduction rules that ultimately make a syntactic approach to semantic equality viable again.

2. On the other hand, there could be possibilities for speeding up the brute force approach. The algorithm currently in use simply enumerates all environments and computes and compares the denotations of the terms in question for each of them.

   Similar to the first point, additional properties known about the universes or about the denotation function itself might have the potential to be used in pruning the search space.

Besides the direct approach to higher-ranked polyvariance that was taken by Koot and which we chose to adapt, there is also the earlier constraint based system by Holdermans and Hage that has been used for performing flow analysis.

Unfortunately, we cannot further investigate how the issue at hand would manifest itself in this constraint based system because their algorithm [10] is incorrect in its current form. All the constraints that are generated by the algorithm are free of cycles. Consequently, it never actually finds a fixpoint as demonstrated by the following example.

Example 3.30. The function in the following term cycles through its arguments, which causes each of the initial parameters to flow into the condition of the if-expression at some point.

$$(\mathit{fix}\ (\lambda f.\, \lambda x.\, \lambda y.\, \lambda z.\, \mathbf{if}\ x\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ f\ z\ x\ y))\ \mathit{True}\ \mathit{True}\ \mathit{True}$$

Labeling the program points where values are produced or consumed leads to the term

$$\begin{array}{l} ((((\mathit{fix}\ (\lambda f.\, (\lambda x.\, (\lambda y.\, (\lambda z. \\ \quad (\mathbf{if}\ x\ \mathbf{then}\ \mathit{True}\ \{1\}\ \mathbf{else}\ (((f\ z)\ \{2\}\ x)\ \{3\}\ y)\ \{4\})\ \{5\}) \\ \quad \{6\})\{7\})\{8\})\ \{9\})\ \{10\}\ \mathit{True}\ \{11\})\ \{12\}\ \mathit{True}\ \{13\})\ \{14\}\ \mathit{True}\ \{15\})\ \{16\} \end{array}$$

We would expect that the set of values that flow to the if-expression (5) consists of all three arguments (11, 13 and 15), but the flow analysis algorithm only outputs the set $\{11, 15\}$. In fact, even variants of this example with more arguments are always analyzed to just have two arguments flowing to the if-expression.

3.5. Correctness of the Algorithm

In order to demonstrate the correctness of the analysis algorithm presented in this chapter we have to show that for every well-typed underlying term it produces an analysis (i.e. annotated types and effects) that can be derived in the annotated type system (see figure 2.9). That is to say, algorithm $\mathcal{R}$ is sound w.r.t. the annotated type system.

3.5.1. Soundness

Definition 3.31. We say a type and effect environment $\widehat{\Gamma}$ is well-formed under an environment Σ, if $\widehat{\Gamma}$ is conservative and for all bindings $x : \widehat{\tau}\ \&\ \xi$ in $\widehat{\Gamma}$ we have $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}$ and $\Sigma \vdash_{\mathrm{s}} \xi : \star$.

Since the reconstruction algorithm applies canonicalization after every step, the following two lemmas are needed to show that this still leads to a valid derivation. It does not matter what $\llbracket \xi \rrbracket_\Sigma$ evaluates to, as long as it is semantically equivalent to ξ.

Lemma 3.32. *If we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$, then we also have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \llbracket \xi \rrbracket_\Sigma$.*

Proof. A fundamental property of canonicalization is that $\llbracket \xi \rrbracket_\Sigma$ is semantically equivalent to ξ under the environment Σ, that is to say $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \llbracket \xi \rrbracket_\Sigma$ and $\Sigma \vdash_{\text{sub}} \llbracket \xi \rrbracket_\Sigma \sqsubseteq \xi$. The conclusion follows from a single application of [T-Sub]. ☐

Lemma 3.33. *If we have $\Sigma \vdash_{\text{wft}} \widehat{\tau}$, then $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \llbracket \widehat{\tau} \rrbracket_\Sigma$ and $\Sigma \vdash_{\text{sub}} \llbracket \widehat{\tau} \rrbracket_\Sigma \leqslant \widehat{\tau}$.*

Proof. By induction on the derivation tree of $\Sigma \vdash_{\text{wft}} \widehat{\tau}$ using the fact that canonical forms preserve semantics. ☐

Next, we establish the correctness of the matching algorithm $\mathcal{M}$ that is used in the type reconstruction algorithm.

Lemma 3.34. *Let $\widehat{\tau}'$ be a pattern type such that $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}' \,\&\, \beta \;\overline{\alpha_i} \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\widehat{\tau}$ be a conservative type such that $\lfloor \widehat{\tau} \rfloor = \lfloor \widehat{\tau}' \rfloor$ and $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\text{wft}} \widehat{\tau}$. Then there is a substitution $\theta = \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \widehat{\tau}'; \widehat{\tau})$ such that $\theta \widehat{\tau}' = \widehat{\tau}$ and $\Sigma \vdash_{\text{s}} \theta \beta_j : \kappa_{\beta_j}$ for all $\beta_j \in \text{dom}(\theta) = \overline{\beta_j} \setminus \{\beta\}$.*

Proof. See appendix, page 122. ☐

Similarly, we show some useful properties of the join $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ of two annotated types.

Lemma 3.35. *Let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ be conservative types and Σ a sort environment such that $\lfloor \widehat{\tau}_1 \rfloor = \lfloor \widehat{\tau}_2 \rfloor = \tau$ for some τ, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$.*

Then $\lfloor \widehat{\tau}_1 \sqcup \widehat{\tau}_2 \rfloor = \tau$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1 \sqcup \widehat{\tau}_2$ and $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ is conservative.

Proof. See appendix, page 124. ☐

Based on this knowledge, we can now show that algorithm $\mathcal{R}$ is sound w.r.t. to the declarative type system.

Theorem 3.36. *Let t be a source term, Σ a sort environment and $\widehat{\Gamma}$ a type and effect environment well-formed under Σ such that $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \widehat{t} : \widehat{\tau} \,\&\, \xi$ for some $\widehat{t}$, $\widehat{\tau}$ and ξ.*

Then, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}$, $\Sigma \vdash_{\text{s}} \xi : \star$ and $\widehat{\tau}$ is conservative.

Proof. See appendix, page 125. ☐

3.5.2. Completeness

The next step is to show that our analysis succeeds in deriving an annotated type and effect for any well-typed source term. The crucial part here is the termination of the fixpoint iteration.

In order to show the convergence of the fixpoint iteration, we start by defining an equivalence relation on type and effect pairs.

Definition 3.37. Let τ be an underlying type and let Σ be a sort environment. We define

1. the set of conservative annotated type and effect pairs with corresponding underlying type τ well-formed under Σ,

$$\begin{aligned}&\mathbf{TyEff}_\tau(\Sigma)\\&= \left\{\widehat{\tau} \mathbin{\&} \xi \in \widehat{\mathbf{Ty}} \times \mathbf{AnnTm} \mid \Sigma \vdash_{\text{wft}} \widehat{\tau} \wedge \Sigma \vdash_{\text{s}} \xi : \star \wedge \widehat{\tau} \text{ conservative} \wedge \lfloor\widehat{\tau}\rfloor = \tau\right\}\end{aligned}$$

2. the equivalence relation $\equiv_{\tau;\Sigma} \subseteq \mathbf{TyEff}_\tau(\Sigma) \times \mathbf{TyEff}_\tau(\Sigma)$ given by

$$\begin{aligned}&\widehat{\tau}_1 \mathbin{\&} \xi_1 \equiv_{\tau;\Sigma} \widehat{\tau}_2 \mathbin{\&} \xi_2\\\Longleftrightarrow\ &\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \wedge \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_1 \wedge \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2 \wedge \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_1,\end{aligned}$$

3. $\mathbf{TyEff}_\tau(\Sigma)/\equiv_{\tau;\Sigma}$ to be the quotient of the set of well-formed annotated type and effect pairs by the equivalence relation $\equiv_{\tau;\Sigma}$,

4. $[\widehat{\tau} \mathbin{\&} \xi]_\Sigma$ to denote the equivalence class of $\widehat{\tau} \mathbin{\&} \xi \in \mathbf{TyEff}_{\lfloor\widehat{\tau}\rfloor}(\Sigma)$.

When the underlying type τ is obvious from the context, we write $\equiv_\Sigma$ instead of $\equiv_{\tau;\Sigma}$.

Lemma 3.38. *$\equiv_\Sigma$ is an equivalence relation.*

Proof. We need to show that $\equiv_\Sigma$ is reflexive, symmetric and transitive.

Reflexivity For all $\widehat{\tau} \mathbin{\&} \xi \in \mathbf{TyEff}_\tau(\Sigma)$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}$ by [SUB-REFL] and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi$ by the reflexivity of subsumption. Therefore $\widehat{\tau} \mathbin{\&} \xi \equiv_\Sigma \widehat{\tau}\ \xi$.

Symmetry Given $\widehat{\tau}_1 \mathbin{\&} \xi_1$ and $\widehat{\tau}_2 \mathbin{\&} \xi_2$ such that $\widehat{\tau}_1 \mathbin{\&} \xi_1 \equiv_\Sigma \widehat{\tau}_2 \mathbin{\&} \xi_2$, then we also have $\widehat{\tau}_2 \mathbin{\&} \xi_2 \equiv_\Sigma \widehat{\tau}_1 \mathbin{\&} \xi_1$ by the symmetry of $\wedge$.

Transitivity Given $\widehat{\tau}_1 \mathbin{\&} \xi_1$, $\widehat{\tau}_2 \mathbin{\&} \xi_2$ and $\widehat{\tau}_3 \mathbin{\&} \xi_3$ such that $\widehat{\tau}_1 \mathbin{\&} \xi_1 \equiv_\Sigma \widehat{\tau}_2 \mathbin{\&} \xi_2$ and $\widehat{\tau}_2 \mathbin{\&} \xi_2 \equiv_\Sigma \widehat{\tau}_3 \mathbin{\&} \xi_3$. Then we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_1$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_3$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_2$. By [SUB-TRANS], we can infer $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_1$. By the transitivity of subsumption, we have $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_3$ and $\Sigma \vdash_{\text{sub}} \xi_3 \sqsubseteq \xi_1$. Hence $\widehat{\tau}_1 \mathbin{\&} \xi_1 \equiv_\Sigma \widehat{\tau}_3 \mathbin{\&} \xi_3$.

□

An important ingredient in the completeness proof is the fact that the set of equivalence classes defined above is finite.

Lemma 3.39. $\mathbf{TyEff}_\tau(\Sigma)/\equiv_{\tau;\Sigma}$ *is finite.*

Proof. Suppose not. Then there is an infinite number of type and effect pairs $(\widehat{\tau}_i \mathbin{\&} \xi_i)_{i\in\mathbb{N}}$ such that none of them is equivalent to any other pair. Since the type and effect pairs are all of the same shape, they can only differ in the (potentially nested) annotations. Hence, there must be an infinite number of dependency terms $(\xi'_i)_{i\in\mathbb{N}}$ occurring in the

same position that are not semantically equivalent to any other (because there are only finitely many positions where an annotation can occur in a type). For every pair of such terms ξ'_i, ξ'_j, there is an environment $\rho_{i,j}$ such that $[\![\xi'_i]\!]_{\rho_{i,j}} \neq [\![\xi'_j]\!]_{\rho_{i,j}}$.

In the proof of lemma 3.1 we established that there can only be a finite number of environments compatible with any given sort environment. Therefore, there is an environment ρ and an infinite subset $S \subseteq \mathbb{N}$ such that for all $i, j \in S$ with $i \neq j$ we have $[\![\xi'_i]\!]_\rho \neq [\![\xi'_j]\!]_\rho$. However, by assumption, the underlying lattice is finite. Contradiction. □

Next, we define an order relation on equivalence classes based on the subtyping and subsumption relations.

Definition 3.40. We define a relation $\sqsubseteq_{\tau;\Sigma}$ on $\mathbf{TyEff}_\tau(\Sigma)/\equiv_{\tau;\Sigma}$ given by

$$[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqsubseteq_{\tau;\Sigma} [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma \iff \Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \wedge \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2.$$

Lemma 3.41. *$\sqsubseteq_\Sigma$ is well-defined and a partial order.*

Proof. We first show that the definition of $\sqsubseteq_{\tau;\Sigma}$ is independent of the chosen representative. Let $\mathcal{E}_1, \mathcal{E}_2 \in \mathbf{TyEff}_\tau(\Sigma)/\equiv_{\tau;\Sigma}$ denote two arbitrary equivalence classes and let $\widehat{\tau}_1 \,\&\, \xi_1, \widehat{\tau}'_1 \,\&\, \xi'_1 \in \mathcal{E}_1$ and $\widehat{\tau}_2 \,\&\, \xi_2, \widehat{\tau}'_2 \,\&\, \xi'_2 \in \mathcal{E}_2$ be arbitrary representatives of theses classes. We have $\widehat{\tau}_1 \,\&\, \xi_1 \equiv_{\tau;\Sigma} \widehat{\tau}'_1 \,\&\, \xi'_1$ and $\widehat{\tau}_2 \,\&\, \xi_2 \equiv_{\tau;\Sigma} \widehat{\tau}'_2 \,\&\, \xi'_2$.

Suppose that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2$ hold. By definition 3.37, we additionally have (among others) $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}_1 \wedge \Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi_1$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2 \wedge \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2$. By [Sub-Trans], $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ and by the transitivity of subsumption we have $\Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi'_2$. Hence, $\sqsubseteq_\Sigma$ is well-defined.

In order to show that $\sqsubseteq_\Sigma$ is a partial order, we show that the relation is reflexive, transitive and anti-symmetric.

Reflexivity By [Sub-Refl] and the reflexivity of subsumption, $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}$ and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi$ for all $\widehat{\tau} \,\&\, \xi \in \mathbf{TyEff}_\tau(\Sigma)$. Hence, $[\widehat{\tau} \,\&\, \xi]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau} \,\&\, \xi]_\Sigma$.

Transitivity Given $\widehat{\tau}_1 \,\&\, \xi_1$, $\widehat{\tau}_2 \,\&\, \xi_2$ and $\widehat{\tau}_3 \,\&\, \xi_3$ such that $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma$ and $[\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_3 \,\&\, \xi_3]_\Sigma$. Then we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_3$ as well as $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_2$. By [Sub-Trans], we can infer $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_3$. By the transitivity of subsumption, we have $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_3$. Hence $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_3 \,\&\, \xi_3]_\Sigma$.

Antisymmetry Suppose that $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma$ and $[\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma$. Then, by definition $\widehat{\tau}_1 \,\&\, \xi_1 \equiv_\Sigma \widehat{\tau}_2 \,\&\, \xi_2$. Therefore, $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma = [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma$.

□

Definition 3.42. We define a pointwise join operation on type and effect pairs as follows:

$$(\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) := \widehat{\tau}_1 \sqcup \widehat{\tau}_2 \,\&\, \xi_1 \sqcup \xi_2.$$

Lemma 3.43. *The join operation $\sqcup$ on type and effect pairs is*

1. *associative, i.e. for all* $\widehat{\tau}_1 \,\&\, \xi_1, \widehat{\tau}_2 \,\&\, \xi_2, \widehat{\tau}_3 \,\&\, \xi_3 \in \mathbf{TyEff}_\tau(\Sigma)$ *we have* $((\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2)) \sqcup (\widehat{\tau}_3 \,\&\, \xi_3) \equiv_\Sigma (\widehat{\tau}_1 \,\&\, \xi_1) \sqcup ((\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_3 \,\&\, \xi_3))$*,*
2. *commutative, i.e. for all* $\widehat{\tau}_1 \& \xi_1, \widehat{\tau}_2 \& \xi_2 \in \mathbf{TyEff}_\tau(\Sigma)$ *we have* $(\widehat{\tau}_1 \& \xi_1) \sqcup (\widehat{\tau}_2 \& \xi_2) \equiv_\Sigma (\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_1 \,\&\, \xi_1)$*, and*
3. *idempotent, i.e. for all* $\widehat{\tau}_1 \& \xi_1 \in \mathbf{TyEff}_\tau(\Sigma)$ *we have* $(\widehat{\tau}_1 \& \xi_1) \sqcup (\widehat{\tau}_1 \& \xi_1) \equiv_\Sigma \widehat{\tau}_1 \& \xi_1$*.*

Proof. All three properties follow from the associativity, commutativity and idempotence of the underlying lattice. We show an exemplary proof for commutativity, the other two can be done similarly.

Let $\widehat{\tau}_1 \,\&\, \xi_1, \widehat{\tau}_2 \,\&\, \xi_2 \in \mathbf{TyEff}_\tau(\Sigma)$ be arbitrary. We show $(\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) \equiv_\Sigma (\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_1 \,\&\, \xi_1)$ by induction on $\widehat{\tau}_1$. In all cases, we have $\Sigma \vdash \xi_1 \sqcup \xi_2 \equiv \xi_2 \sqcup \xi_1$ by the commutativity of the underlying lattice.

$\widehat{\tau}_1 = \widehat{\textbf{unit}}$ In this case, $\widehat{\tau}_2 = \widehat{\text{unit}}$ and therefore $\widehat{\tau}_1 \sqcup \widehat{\tau}_2 = \widehat{\text{unit}} = \widehat{\tau}_2 \sqcup \widehat{\tau}_1$ as well. The equivalence follows from reflexivity.

$\widehat{\tau}_1 = \forall \beta :: \kappa.\widehat{\tau}_1'$ Then $\widehat{\tau}_2 = \forall \beta' :: \kappa'.\widehat{\tau}_2'$ for some $\widehat{\tau}_2'$. We can assume $\beta' = \beta$ due to alpha-equivalence. By induction, we have $(\widehat{\tau}_1' \,\&\, \xi_1) \sqcup (\widehat{\tau}_2' \,\&\, \xi_2) \equiv_{\Sigma,\beta::\kappa} (\widehat{\tau}_2' \,\&\, \xi_2) \sqcup (\widehat{\tau}_1' \,\&\, \xi_1)$.

We have $(\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) = \forall \beta :: \kappa.\widehat{\tau}_1' \sqcup \widehat{\tau}_2' \,\&\, \xi_1 \sqcup \xi_2 \equiv_\Sigma \forall \beta :: \kappa.\widehat{\tau}_2' \sqcup \widehat{\tau}_1' \,\&\, \xi_2 \sqcup \xi_1 = (\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_1 \,\&\, \xi_1)$ by [Sub-Forall].

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle \xi_1' \rangle + \widehat{\tau}_1''\langle \xi_1'' \rangle$ Then $\widehat{\tau}_2 = \widehat{\tau}_2'\langle \xi_2' \rangle + \widehat{\tau}_2''\langle \xi_2'' \rangle$ for some $\widehat{\tau}_2'$, $\widehat{\tau}_2''$, ξ_2' and ξ_2''. By induction, we have $(\widehat{\tau}_1' \& \xi_1') \sqcup (\widehat{\tau}_2' \& \xi_2') \equiv_\Sigma (\widehat{\tau}_2' \& \xi_2') \sqcup (\widehat{\tau}_1' \& \xi_1')$ and $(\widehat{\tau}_1'' \& \xi_1'') \sqcup (\widehat{\tau}_2'' \& \xi_2'') \equiv_\Sigma (\widehat{\tau}_2'' \& \xi_2'') \sqcup (\widehat{\tau}_1'' \& \xi_1'')$. Therefore, we must also have $(\widehat{\tau}_1 \& \xi_1) \sqcup (\widehat{\tau}_2 \& \xi_2) \equiv_\Sigma (\widehat{\tau}_2 \& \xi_2) \sqcup (\widehat{\tau}_1 \& \xi_1)$ by [Sub-Sum].

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle \xi_1' \rangle \times \widehat{\tau}_1''\langle \xi_1'' \rangle$ Analogously to the previous case.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle \xi_1' \rangle \to \widehat{\tau}_1''\langle \xi_1'' \rangle$ Similarly to the previous case, but noting that the contravariant positions of the function arrows must be the same.

□

The following lemma shows that the join operation interacts with the subtyping and subsumption relation in a way we would expect from a lattice.

Lemma 3.44. *For all* $\widehat{\tau}_1 \,\&\, \xi_1, \widehat{\tau}_2 \,\&\, \xi_2 \in \mathbf{TyEff}_\tau(\Sigma)$*, we have*

$$\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \wedge \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2 \iff (\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) \equiv_\Sigma \widehat{\tau}_2 \,\&\, \xi_2.$$

Proof. Suppose that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \wedge \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2$ holds. We show $(\widehat{\tau}_1 \& \xi_1) \sqcup (\widehat{\tau}_2 \& \xi_2) \equiv_\Sigma \widehat{\tau}_2 \,\&\, \xi_2$ by induction on $\widehat{\tau}_1$.

$\widehat{\tau}_1 = \widehat{\textbf{unit}}$ Then $\widehat{\tau}_2 = \widehat{\text{unit}}$ as well. Because we have $\Sigma \vdash \xi_1 \sqcup \xi_2 \equiv \xi_2$ due to the lattice properties, it follows that $(\widehat{\text{unit}} \,\&\, \xi_1) \sqcup (\widehat{\text{unit}} \,\&\, \xi_2) \equiv_\Sigma \widehat{\text{unit}} \,\&\, \xi_2$ must also hold.

$\widehat{\tau}_1 = \forall\beta :: \kappa.\widehat{\tau}_1'$ The $\widehat{\tau}_2 = \forall\beta :: \kappa.\widehat{\tau}_2'$ for some $\widehat{\tau}_2'$. By lemma 2.30, we have $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$. By induction, we have $(\widehat{\tau}_1' \,\&\, \xi_1) \sqcup (\widehat{\tau}_2' \,\&\, \xi_2) \equiv_\Sigma \widehat{\tau}_2' \,\&\, \xi_2$. By applying [Sub-Forall], we get $\forall\beta :: \kappa.\widehat{\tau}_1' \sqcup \widehat{\tau}_2' \,\&\, \xi_1 \sqcup \xi_2 \equiv_\Sigma \forall\beta :: \kappa.\widehat{\tau}_2' \,\&\, \xi_2$.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle$ Then $\widehat{\tau}_2 = \widehat{\tau}_2'\langle\xi_2'\rangle + \widehat{\tau}_2''\langle\xi_2''\rangle$ for some $\widehat{\tau}_2'$, $\widehat{\tau}_2''$, ξ_2' and ξ_2''. By lemma 2.30, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$, $\Sigma \vdash_{\text{sub}} \xi_1' \sqsubseteq \xi_2'$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$ and $\Sigma \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$.

By induction, we have $(\widehat{\tau}_1' \,\&\, \xi_1') \sqcup (\widehat{\tau}_2' \,\&\, \xi_2') \equiv_\Sigma (\widehat{\tau}_2' \,\&\, \xi_2')$ and $(\widehat{\tau}_1'' \,\&\, \xi_1'') \sqcup (\widehat{\tau}_2'' \,\&\, \xi_2'') \equiv_\Sigma \widehat{\tau}_2'' \,\&\, \xi_2''$. Therefore, we must also have $(\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) \equiv_\Sigma \widehat{\tau}_2 \,\&\, \xi_2$ by [Sub-Sum].

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle \times \widehat{\tau}_1''\langle\xi_1''\rangle$ Analogously to the previous case.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_1''\rangle$ Similarly to the previous case, but noting that the contravariant positions of the function arrows must be the same.

□

We also need this lemma relating the substitutions obtained when matching a subtype of another type with the same pattern type.

Lemma 3.45. *If we have $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$, $\theta_1 = \mathcal{M}(\Sigma'; \widehat{\tau}; \widehat{\tau}_1)$ and $\theta_2 = \mathcal{M}(\Sigma'; \widehat{\tau}; \widehat{\tau}_2)$, then $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$ for all $\beta \in \text{dom}(\theta_1) = \text{dom}(\theta_2)$.*

Proof. By induction on the derivation of $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$.

[Sub-Refl] Trivial, since $\widehat{\tau}_1 = \widehat{\tau}_2$ and therefore also $\theta_1 = \theta_2$.

[Sub-Trans] We have $\widehat{\tau}'$ such that $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'$ and $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau}_2$. By induction, we get $\theta_1 = \mathcal{M}(\Sigma'; \widehat{\tau}; \widehat{\tau}_1)$, $\theta' = \mathcal{M}(\Sigma'; \widehat{\tau}; \widehat{\tau}')$, $\theta_2 = \mathcal{M}(\Sigma'; \widehat{\tau}; \widehat{\tau}_2)$ such that $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \sqsubseteq \theta'(\beta)$ and $\Sigma \vdash_{\text{sub}} \theta'(\beta) \leqslant \theta_2(\beta)$ for all β. We apply [Sub-Trans] once for every variable and get $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \leqslant \theta_2(\beta)$ for all β.

[Sub-Forall] We have $\widehat{\tau}_1 = \forall\beta :: \kappa.\widehat{\tau}_1'$ and $\widehat{\tau}_2 = \forall\beta :: \kappa.\widehat{\tau}_2'$ such that $\Sigma, \Sigma', \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$. Therefore, $\theta_1 = \mathcal{M}(\Sigma', \beta :: \kappa; \widehat{\tau}; \widehat{\tau}_1')$ and $\theta_2 = \mathcal{M}(\Sigma', \beta :: \kappa; \widehat{\tau}; \widehat{\tau}_2')$.

By induction, $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$ for all $\beta \in \text{dom}(\theta_1) = \text{dom}(\theta_2)$.

[Sub-Sum] We have $\widehat{\tau} = \widehat{\tau}'\langle\beta\rangle + \widehat{\tau}''\langle\beta'\rangle$, $\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle$ and $\widehat{\tau}_2 \;\cdot\!+\!\cdot\; (\widehat{\tau}_2'\langle\xi_2'\rangle)\; (\widehat{\tau}_2''\langle\xi_2''\rangle)$ such that $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$, $\Sigma, \Sigma' \vdash_{\text{sub}} \xi_1' \sqsubseteq \xi_2'$, $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$ and $\Sigma, \Sigma' \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$.

Suppose that $\Sigma' = \overline{\beta_i :: \kappa_{\beta_i}}$. We get

$$\theta_1 = [\beta \mapsto \lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_1', \beta' \mapsto \lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_1''] \circ \mathcal{M}(\Sigma'; \widehat{\tau}'; \widehat{\tau}_1') \circ \mathcal{M}(\Sigma; \widehat{\tau}''; \widehat{\tau}_1'')$$

and

$$\theta_2 = [\beta \mapsto \lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_2', \beta' \mapsto \lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_2''] \circ \mathcal{M}(\Sigma'; \widehat{\tau}'; \widehat{\tau}_2') \circ \mathcal{M}(\Sigma; \widehat{\tau}''; \widehat{\tau}_2'').$$

For β, we have to show $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$, i.e. for all compatible environments ρ we have $[\![\lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_1']\!]_\rho \sqsubseteq [\![\lambda\overline{\beta_i :: \kappa_{\beta_i}}.\xi_2']\!]_\rho$. We need to compare the resulting

functions pointwise. But by assumption, we already have $[\![\xi_1']\!]_{\rho'} \sqsubseteq [\![\xi_2']\!]_{\rho'}$ for all environments compatible with Σ, Σ', and therefore in particular for all possible arguments that could be passed to these functions. The same holds for β' using a similar reasoning and the remaining subsumption statements follow by induction.

[Sub-Prod] Analogously to the previous case.

[Sub-Arr] Analogously to the previous cases, but only recursing on the covariant part of the function arrow, since the contravariant positions are necessarily equal.

□

The following lemma shows that the subtyping relation between conservative types is preserved by certain kinds of substitutions.

Lemma 3.46. *Let Σ and Σ' be sort environments with disjoint domains and let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ be conservative types. If we have $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ then we also have $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}_1 \leqslant \theta_2\widehat{\tau}_2$ for all substitutions $\theta_1, \theta_2 : \text{dom}(\Sigma') \to \textbf{AnnTm}$ such that $\Sigma \vdash_{\text{sub}} \theta_1(\beta) \sqsubseteq \theta_2(\beta)$, $\Sigma \vdash_{\text{s}} \theta_1(\beta) : \Sigma'(\beta)$ and $\Sigma \vdash_{\text{s}} \theta_2(\beta) : \Sigma'(\beta)$ for all $\beta \in \text{dom}(\Sigma')$.*

Proof. By induction on the conservative type $\widehat{\tau}_1$. By lemma 2.29, we know that $\widehat{\tau}_2$ has the same shape as $\widehat{\tau}_1$.

$\widehat{\tau}_1 = \widehat{\textbf{unit}}$ In this case, $\widehat{\tau}_2 = \widehat{\text{unit}}$ as well. Since $\theta_1\widehat{\text{unit}} = \widehat{\text{unit}} = \theta_2\widehat{\text{unit}}$, $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}_1 \leqslant \theta_2\widehat{\tau}_2$ holds by assumption.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle$ We must have $\widehat{\tau}_2 = \widehat{\tau}_2'\langle\xi_2'\rangle + \widehat{\tau}_2''\langle\xi_2''\rangle$ for some $\widehat{\tau}_2'$, $\widehat{\tau}_2''$, ξ_2' and ξ_2''. By lemma 2.30, $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$, $\Sigma, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$, $\Sigma, \Sigma' \vdash_{\text{sub}} \xi_1' \sqsubseteq \xi_2'$ and $\Sigma, \Sigma' \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$.

Then, by induction $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}_1' \leqslant \theta_2\widehat{\tau}_2'$, $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}_1'' \leqslant \theta_2\widehat{\tau}_2''$ and by lemma 2.17, $\Sigma \vdash_{\text{sub}} \theta_1\xi_1' \sqsubseteq \theta_1\xi_2'$ and $\Sigma \vdash_{\text{sub}} \theta_1\xi_1'' \sqsubseteq \theta_2\xi_2''$. Hence, we can derive $\Sigma \vdash_{\text{sub}} \theta_1(\widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_1''\langle\xi_1''\rangle) \leqslant \theta_2(\widehat{\tau}_2'\langle\xi_2'\rangle + \widehat{\tau}_2''\langle\xi_2''\rangle)$.

$\widehat{\tau}_1 = \widehat{\tau}_1'\langle\xi_1'\rangle \times \widehat{\tau}_1''\langle\xi_1''\rangle$ This case can be proven analogously to the previous case.

$\widehat{\tau}_1 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_1''\rangle$ This case needs special care in handling the contravariant argument position of the function type. We have $\widehat{\tau}_2 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_2'\langle\xi_2'\rangle \to \widehat{\tau}_2''\langle\xi_2''\rangle$. As $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative, we have $\emptyset \vdash_p \widehat{\tau}_1' \& \xi_1' \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\emptyset \vdash_p \widehat{\tau}_2' \& \xi_2' \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$.

We assume that the variables $\overline{\beta_j}$ are disjoint from those in Σ and Σ', because we could always obtain an equivalent type through renaming. By lemma 3.14, $\text{fav}(\widehat{\tau}_1') \cup \text{fav}(\xi_1') \subseteq \overline{\beta_j}$ and $\text{fav}(\widehat{\tau}_2') \cup \text{fav}(\xi_2') \subseteq \overline{\beta_j}$. Therefore, none of the free variables of $\widehat{\tau}_1'$ and ξ_1' are free in $\widehat{\tau}_1$, and the same holds for $\widehat{\tau}_2$.

As the substitutions do not apply to the free variables of the argument positions,

$$\theta_1\widehat{\tau}_1 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1'\langle\xi_1'\rangle \to \theta_1\widehat{\tau}_1''\langle\theta_1\xi_1''\rangle \text{ and } \theta_2\widehat{\tau}_2 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_2'\langle\xi_2'\rangle \to \theta_2\widehat{\tau}_2''\langle\theta_2\xi_2''\rangle.$$

We can assume $\widehat{\tau}_1' = \widehat{\tau}_2'$ and $\xi_1' = \xi_2'$ due to lemma 3.15.

Through repeated application of lemma 2.30, we can infer $\Sigma, \Sigma', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$ and $\Sigma, \Sigma', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$. The contexts in the judgments above can be reordered to $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}}, \Sigma' \vdash_{\text{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$ and $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}}, \Sigma' \vdash_{\text{sub}} \xi_1'' \sqsubseteq \xi_2''$.

By induction, respectively lemma 2.17, we get $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \theta_1\widehat{\tau}_1'' \leqslant \theta_2\widehat{\tau}_2''$ and $\Sigma, \overline{\beta_j :: \kappa_j} \vdash_{\text{sub}} \theta_1\xi_1'' \sqsubseteq \theta_2\xi_2''$. The judgments $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}_2' \leqslant \widehat{\tau}_1'$ as well as $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \xi_2' \sqsubseteq \xi_1'$ follow by removing the variables in Σ' from the context, as they are not free in the respective types and effects (using lemma 2.31).

Lastly, we can derive $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}_1 \leqslant \theta_2\widehat{\tau}_2$ by [Sub-Arr] and [Sub-Forall].

□

Note that, in contrast to lemma 2.32, the proof of the foregoing lemma crucially relies on the fact that conservative types are invariant in the argument types of function arrows.

We now prove the following monotonicity lemma stating that increasing a type or effect in the environment (w.r.t. the subtyping or subsumption relations) passed to $\mathcal{R}$ also leads to greater or equal type and effect being computed.

Lemma 3.47. *Let Σ be a sort environment, let $\widehat{\Gamma_1}$, $\widehat{\Gamma_2}$ be type and effect environments with identical domains well-formed under Σ and let t be a source term such that we have $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; t) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; t) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$.*

If for all $x \in \operatorname{dom}(\widehat{\Gamma_1})$ with $\widehat{\Gamma_1}(x) = \widehat{\tau} \,\&\, \xi$ and $\widehat{\Gamma_2}(x) = \widehat{\tau}' \,\&\, \xi'$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi'$, then also $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2$.

Proof. See appendix, page 129. □

We have now covered the preliminaries required for proving the completeness of our type reconstruction algorithm.

Theorem 3.48 (Completeness)**.** *Given a source term t, a sort environment Σ, a type and effect environment $\widehat{\Gamma}$ well-formed under Σ, and an underlying type τ such that $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t : \tau$, then there are $\widehat{t}$, $\widehat{\tau}$ and ξ such that $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \widehat{t} : \widehat{\tau} \,\&\, \xi$ and $\lfloor\widehat{\tau}\rfloor = \tau$, $\lfloor\widehat{t}\rfloor = t$.*

Proof. See appendix, page 133. □

As a corollary from the foregoing theorems, our analysis is a conservative extension of the underlying type system.

Corollary 3.49 (Conservative Extension)**.** *Let t be a source term, τ be a type and Γ a type environment such that $\Gamma \vdash_{\text{t}} t : \tau$. Then there are Σ, $\widehat{\Gamma}$, $\widehat{t}$, $\widehat{\tau}$, ξ such that $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ with $\lfloor\widehat{t}\rfloor = t$, $\lfloor\widehat{\tau}\rfloor = \tau$ and $\lfloor\widehat{\Gamma}\rfloor = \Gamma$.*

Proof. Construct $\widehat{\Gamma}$ from Γ by replacing every binding $x : \tau'$ with a binding $x : \bot_{\tau'} \,\&\, \bot_{\star}$. Set Σ to be the empty sort context. By lemma 3.25, $\bot_{\tau'}$ is conservative, $\Sigma \vdash_{\text{wft}} \bot_{\tau'}$ and $\lfloor\bot_{\tau'}\rfloor = \tau'$ for all such τ'. The latter also implies $\lfloor\widehat{\Gamma}\rfloor = \Gamma$ and therefore $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t : \tau$. By lemma 3.23, $\Sigma \vdash_{\text{s}} \bot_{\star} : \star$. Hence, $\widehat{\Gamma}$ is well-formed under Σ.

This allows us to apply theorem 3.48, so there must be $\widehat{t}$, $\widehat{\tau}$ and ξ such that $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \widehat{t} : \widehat{\tau} \,\&\, \xi$ with $\lfloor\widehat{\tau}\rfloor = \tau$ and $\lfloor\widehat{t}\rfloor = t$. Since $\widehat{\Gamma}$ is well-formed under Σ, we can conclude from theorem 3.36 that $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ holds. □

3.5.3. Best Analyses

Now that we have established that the algorithm $\mathcal{R}$ can always compute a valid annotated type for every well-typed source term, we need to show that the types and effects computed by $\mathcal{R}$ are always the best possible ones in the following sense.

Definition 3.50 (Best Analysis)**.** Let Σ be a sort environment, $\widehat{\Gamma}$ be a type and effect environment well-formed under Σ and t a source term.

A triple $\widehat{t} : \widehat{\tau} \,\&\, \xi$ is a *best analysis* for t under the environments Σ and $\widehat{\Gamma}$ if $\widehat{\tau}$ is conservative, $\lfloor \widehat{t} \rfloor = t$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ and for all triples $\widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ such that $\widehat{\tau}'$ is conservative, $\lfloor \widehat{t'} \rfloor = t$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ we have that $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi'$.

As proving the best analyses theorem unfortunately turned out to be harder than expected, it has not been finished to date. Nonetheless, we managed to provide a partial proof sketch for a restricted variant of said theorem.

We start by showing several parts required for proving that the fixpoints computed by $\mathcal{R}$ are least. Our proof will use the equivalence classes we have introduced in the previous part. We begin by defining the join operation for equivalence classes and show that this definition is in fact well-defined.

Definition 3.51. We define the join of two equivalence classes of type and effect pairs as follows:

$$[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma := [\widehat{\tau}_1 \sqcup \widehat{\tau}_2 \,\&\, \xi_1 \sqcup \xi_2]..$$

Lemma 3.52. $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma$ *is well-defined.*

Proof. Let $\widehat{\tau}_1 \,\&\, \xi_1 \equiv_\Sigma \widehat{\tau}_1' \,\&\, \xi_1'$ and $\widehat{\tau}_2 \,\&\, \xi_2 \equiv_\Sigma \widehat{\tau}_2' \,\&\, \xi_2'$ be arbitrary.

By definition of $\equiv_\Sigma$, we have in particular $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1'$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_1'$ as well as $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_2'$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_2'$. Using lemma 3.43 and lemma 3.44, we can derive

$$\begin{aligned} &((\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2)) \sqcup ((\widehat{\tau}_1' \,\&\, \xi_1') \sqcup (\widehat{\tau}_2' \,\&\, \xi_2')) \\ \equiv_\Sigma &((\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_1' \,\&\, \xi_1')) \sqcup ((\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_2' \,\&\, \xi_2')) \\ \equiv_\Sigma &(\widehat{\tau}_1' \,\&\, \xi_1') \sqcup (\widehat{\tau}_2' \,\&\, \xi_2') \end{aligned}$$

and therefore $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \sqcup \widehat{\tau}_2 \leqslant \widehat{\tau}_1' \sqcup \widehat{\tau}_2'$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqcup \xi_2 \sqsubseteq \xi_1' \sqcup \xi_2'$.

By a similar argument, we also have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1' \sqcup \widehat{\tau}_2' \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \xi_1' \sqcup \xi_2' \sqsubseteq \xi_1 \sqcup \xi_2$, and thus $\widehat{\tau}_1 \sqcup \widehat{\tau}_2 \,\&\, \xi_1 \sqcup \xi_2 = (\widehat{\tau}_1 \,\&\, \xi_1) \sqcup (\widehat{\tau}_2 \,\&\, \xi_2) \equiv_\Sigma (\widehat{\tau}_1' \,\&\, \xi_1') \sqcup (\widehat{\tau}_2' \,\&\, \xi_2') = \widehat{\tau}_1' \sqcup \widehat{\tau}_2' \,\&\, \xi_1' \sqcup \xi_2'$. ☐

We show that the partially ordered set of equivalence classes has a lower bound and least upper bounds.

Lemma 3.53. *The partially ordered set* $(\mathbf{TyEff}_\tau(\Sigma)/\equiv_\Sigma, \sqsubseteq_\Sigma)$ *has 1. a least element* $[\bot_\tau \,\&\, \bot_\star]_\Sigma$ *and 2. least upper bounds* $[\widehat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\widehat{\tau}_2 \,\&\, \xi_2]_\Sigma$.

Proof. For the first part, let $\widehat{\tau} \,\&\, \xi \in \mathbf{TyEff}_\tau(\Sigma)$ be arbitrary. By lemma 3.23, $\Sigma \vdash_{\text{sub}} \bot_\star \sqsubseteq \xi$ and by lemma 3.26, $\Sigma \vdash_{\text{sub}} \bot_\tau \leqslant \widehat{\tau}$. Hence, $[\bot_\tau \,\&\, \bot_\star]_\Sigma$ is indeed a least element.

For the second part, let $[\hat{\tau}_1 \,\&\, \xi_1]_\Sigma, [\hat{\tau}_2 \,\&\, \xi_2]_\Sigma$ be arbitrary. Then, by lemma 3.43 and lemma 3.44, $[\hat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\hat{\tau}_2 \,\&\, \xi_2]_\Sigma$ is an upper bound. Suppose that there is another upper bound $[\hat{\tau} \,\&\, \xi]_\Sigma \sqsubseteq_\Sigma [\hat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\hat{\tau}_2 \,\&\, \xi_2]_\Sigma$. Then

$$\begin{aligned}\hat{\tau} \,\&\, \xi \equiv_\Sigma (\hat{\tau} \,\&\, \xi) \sqcup (\hat{\tau}_2 \,\&\, \xi_2) \equiv_\Sigma ((\hat{\tau} \,\&\, \xi) \sqcup (\hat{\tau}_1 \,\&\, \xi_1)) \sqcup (\hat{\tau}_2 \,\&\, \xi_2) \\ \equiv_\Sigma (\hat{\tau} \,\&\, \xi) \sqcup ((\hat{\tau}_1 \,\&\, \xi_1) \sqcup (\hat{\tau}_2 \,\&\, \xi_2)) \equiv_\Sigma (\hat{\tau}_1 \,\&\, \xi_1) \sqcup (\hat{\tau}_2 \,\&\, \xi_2).\end{aligned}$$

Hence, $[\hat{\tau} \,\&\, \xi]_\Sigma = [\hat{\tau}_1 \,\&\, \xi_1]_\Sigma \sqcup [\hat{\tau}_2 \,\&\, \xi_2]_\Sigma$. ☐

The following lemma provides us with a method for obtaining least fixpoints of a monotone function on the set of equivalence classes.

Lemma 3.54. *Let* $f : (\mathbf{TyEff}_\tau(\Sigma)/\equiv_\Sigma, \sqsubseteq_\Sigma) \to (\mathbf{TyEff}_\tau(\Sigma)/\equiv_\Sigma, \sqsubseteq_\Sigma)$ *be a monotone function. Then there is an* $n \in \mathbb{N}$ *such that* $\mathcal{F} = f^n([\bot_\tau \,\&\, \bot_\star]_\Sigma)$ *(where* f^n *denotes the* n*-fold application of* f*) is the least fixpoint of* f*.*

Proof. We define the sequence $([\hat{\tau}_i \,\&\, \xi_i]_\Sigma)_{i \in \mathbb{N}}$ by

$$\begin{aligned}[\hat{\tau}_0 \,\&\, \xi_0]_\Sigma &= [\bot_\tau \,\&\, \bot_\star]_\Sigma \\ [\hat{\tau}_{i+1} \,\&\, \xi_{i+1}]_\Sigma &= f([\hat{\tau}_i \,\&\, \xi_i]_\Sigma).\end{aligned}$$

Since f is monotone and $\mathbf{TyEff}_\tau(\Sigma)/\equiv_\Sigma$ is finite by lemma 3.39, there must be an $n \in \mathbb{N}$ such that for all $i \geqslant n$, $[\hat{\tau}_i \,\&\, \xi_i]_\Sigma = [\hat{\tau}_{i+1} \,\&\, \xi_{i+1}]_\Sigma$. We set $\mathcal{F} = [\hat{\tau}_n \,\&\, \xi_n]_\Sigma$ which is clearly a fixpoint.

Now suppose that there is another fixpoint $\mathcal{F}'$. We show by induction that $[\hat{\tau}_i \& \xi_i]_\Sigma \sqsubseteq_\Sigma \mathcal{F}'$ for all i.

$i = 0$ $[\hat{\tau}_i \,\&\, \xi_i]_\Sigma = [\bot_\tau \,\&\, \bot_\star]_\Sigma$ is the least element.

$i \to i+1$ By induction, we have $[\hat{\tau}_i \,\&\, \xi_i]_\Sigma \sqsubseteq_\Sigma \mathcal{F}'$, and by monotonicity $[\hat{\tau}_{i+1} \,\&\, \xi_{i+1}]_\Sigma = f([\hat{\tau}_i \,\&\, \xi_i]_\Sigma) \sqsubseteq_\Sigma f(\mathcal{F}') = \mathcal{F}'$

In particular, $\mathcal{F} \sqsubseteq_\Sigma \mathcal{F}'$. ☐

The main issue is the lack of knowledge about $\hat{t}'$, except that its type is conservative. However, just because the outer type is conservative, this does not mean its subterms also have conservative types. Therefore, we additionally assume that the other target term follows a certain structure which ensures that the relevant subterms are conservative as well. This facilitates the comparison between the result of $\mathcal{R}$ and the term this result is related to. The following definition formalizes this structural assumption.

Definition 3.55. A well-typed target term $\hat{t}$ is called *conservative*, if

- $\hat{t} = x$, or
- $\hat{t} = ()$, or
- $\hat{t} = \mathrm{ann}_\ell(\hat{t}')$ and $\hat{t}'$ is conservative, or

- $\widehat{t} = \text{seq}\ \widehat{t_1}\ \widehat{t_2}$ and $\widehat{t_1}$ and $\widehat{t_2}$ are conservative, or
- $\widehat{t} = (\widehat{t_1}, \widehat{t_2})$ and $\widehat{t_1}$ and $\widehat{t_2}$ are conservative, or
- $\widehat{t} = \text{proj}_i(\widehat{t'})$ and $\widehat{t'}$ is conservative, or
- $\widehat{t} = \text{inl}_\tau(\widehat{t'})$ and $\widehat{t'}$ is conservative, or
- $\widehat{t} = \text{inr}_\tau(\widehat{t'})$ and $\widehat{t'}$ is conservative, or
- $\widehat{t} = \textbf{case}\ \widehat{t_1}\ \textbf{of}\ \{\text{inl}(x_1) \to \widehat{t_2}; \text{inr}(x_2) \to \widehat{t_3}\}$ and $\widehat{t_1}, \widehat{t_2}, \widehat{t_3}$ are conservative, or
- $\widehat{t} = \Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t_2}$ such that $\emptyset \vdash_p \widehat{\tau}_1 \,\&\, \xi_1 \triangleright \overline{\beta_i :: \kappa_i}$ and $\widehat{t_2}$ is conservative, or
- $\widehat{t} = \widehat{t_1}\ \langle\overline{\xi_i}\rangle\ \widehat{t_2}$ and $\widehat{t_1}$ and $\widehat{t_2}$ are conservative, or
- $\widehat{t} = \mu x : \widehat{\tau}' \,\&\, \xi'.\widehat{t'}$ and both $\widehat{\tau}'$ and $\widehat{t'}$ are conservative

The definition restricts the locations where annotation abstractions and applications can occur. We conjecture that for every well-typed conservative term, there is a conservative type less than or equal to its original type. There is a partial proof sketch, particularly for the interesting cases of fixpoints, lambda abstractions and applications.

Conjecture 3.56. *If $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ holds for a conservative environment $\widehat{\Gamma}$ compatible with Σ and $\widehat{t}$ is conservative, then there is a conservative $\widehat{\tau}'$ such that $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau}' \,\&\, \xi$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau}$ hold.*

Proof. By induction on $\widehat{t}$ where the following cases are complete:

$\widehat{t} = x$ By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\widehat{\Gamma}(x) = \widehat{\tau}' \,\&\, \xi'}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} x : \widehat{\tau}' \,\&\, \xi'}\ [\text{T-Var}] \qquad \begin{array}{l}\Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \\ \Sigma \vdash_{\text{sub}} \xi' \sqsubseteq \xi\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} x : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

By assumption, $\widehat{\tau}'$ is conservative and we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} x : \widehat{\tau}' \,\&\, \xi$.

$\widehat{t} = \Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t_2}$ By repeatedly applying lemma 2.35 to $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$, we eventually get a derivation

$$\dfrac{\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma} \vdash_{\text{te}} \lambda x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t_2} : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot}\ [\text{T-Abs}]$$

followed by interleaved applications of [T-Sub] and [T-AnnAbs].

By assumption, $\widehat{\tau}_1$ is a pattern type and therefore conservative. Hence, we can apply the induction hypothesis and get a conservative $\widehat{\tau}_2'$ such that $\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{sub}} \widehat{\tau}_2' \leqslant \widehat{\tau}_2$ and $\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2' \,\&\, \xi_2$.

Moreover, since we have $\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2'\langle\xi_2\rangle \leqslant \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ by [Sub-Arr], we can also derive $\Sigma \vdash_{\text{sub}} \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau}$ using the subtyping judgments from the above derivation sequence, interleaved with [Sub-Forall].

By disregarding the subtyping in the above derivation sequence and only retaining the subeffecting part, we can eventually derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi$. The type is conservative by definition.

$\widehat{t} = \mu x : \widehat{\tau}' \,\&\, \xi'.\widehat{t'}$ By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi'}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau}' \,\&\, \xi'.\widehat{t'} : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-Fix}] \qquad \begin{array}{c}\Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \\ \Sigma \vdash_{\text{sub}} \xi' \sqsubseteq \xi\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau}' \,\&\, \xi'.\widehat{t'} : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By assumption, $\widehat{\tau}'$ is conservative and we can derive the desired $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau}' \,\&\, \xi$.

The proof for applications could be done as follows.

$\widehat{t} = \widehat{t_1}\ \langle\overline{\xi_i}\rangle\ \widehat{t_2}$ The idea in this case is to use lemma 2.35 to expose the typing derivation and to use the induction hypothesis for obtaining conservative types for both $\widehat{t_1}$ and $\widehat{t_2}$.

We could then build a substitution from the annotation applications and apply it to the conservative return type of the function. This preserves the conservativeness and the resulting type is still a subtype of the original type.

Presumably, the remaining cases can be proven in a similar fashion. □

While the following lemma is not the result we initially aimed to prove, we hope that it can serve as a starting point for a more general theorem by combining it with some sort of proof normalization for target terms. Again, the proof is incomplete, and more to be understood as a proof sketch at some points.

Conjecture 3.57 (Best Analyses)**.** *Let Σ be a sort environment, $\widehat{\Gamma}$ be a well-formed type and effect environment under Σ and t a source term. $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \widehat{t} : \widehat{\tau} \,\&\, \xi$ is a best analysis for t under Σ and $\widehat{\Gamma}$ under the assumption that the other term is conservative, according to the preceding definition.*

Proof. By theorem 3.48, there are $\widehat{t}$, $\widehat{\tau}$ and ξ with $\lfloor\widehat{t}\rfloor = t$ such that $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \widehat{t} : \widehat{\tau} \,\&\, \xi$. Additionally, by theorem 3.36 $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$ and $\widehat{\tau}$ is conservative.

It remains to show that for all triples $\widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ such that $\widehat{\tau}'$ is conservative, $\lfloor\widehat{t'}\rfloor = t$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi'$. Additionally, we assume that $\widehat{t'}$ is conservative.

Proof by induction on t. The following cases can be considered as complete.

$t = x$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; x) = x : \widehat{\Gamma}(x)$ and $\widehat{\Gamma}(x) = \widehat{\tau} \,\&\, \xi$.

By assumption, $\widehat{t'} = x$ using $\lfloor\widehat{t'}\rfloor = x$ to eliminate other possibilities. By lemma 2.35, any other valid type and effect $\widehat{\tau}' \,\&\, \xi'$ must be larger than or equal to $\widehat{\tau} \,\&\, \xi$. Thus, $x : \widehat{\tau} \,\&\, \xi$ is the best analysis.

$t = ()$ Since $\widehat{\tau}'$ is conservative and $\lfloor \widehat{\tau}' \rfloor = \text{unit}$, $\widehat{\tau}' = \widehat{\text{unit}}$. By definition, $\mathcal{R}(\widehat{\Gamma}; \Sigma; ()) = () : \widehat{\text{unit}} \,\&\, \bot$. Since $\bot$ is the least element, $\Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi'$ always holds. We have $\Sigma \vdash_{\text{sub}} \widehat{\text{unit}} \leqslant \widehat{\text{unit}}$ by [SUB-REFL].

$t = \textbf{ann}_\ell(t_1)$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{ann}_\ell(t_1)) = \text{ann}_\ell(\widehat{t_1}) : \widehat{\tau}_1 \,\&\, \llbracket \ell \sqcup \xi_1 \rrbracket_\Sigma$ such that $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$.

By assumption, $\widehat{t'} = \text{ann}_\ell(\widehat{t'_1})$ for some conservative $\widehat{t'_1}$ and by lemma 2.35 and conjecture 3.56, there is a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'_1} : \widehat{\tau}'_1 \,\&\, \xi'_1 \qquad \Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi'_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(\widehat{t'}) : \widehat{\tau}'_1 \,\&\, \xi'_1}\,[\text{T-ANN}] \qquad \begin{array}{c}\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}' \\ \Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi'\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{ann}_\ell(\widehat{t'}) : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-SUB}]$$

with $\widehat{\tau}'_1$ being conservative.

By induction, $\widehat{t_1} : \widehat{\tau} \,\&\, \xi_1$ is a best analysis for t_1, hence $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1$. By transitivity, we then also have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'$. Since we also have $\Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi'$, we can conclude $\Sigma \vdash_{\text{sub}} \ell \sqcup \xi_1 \sqsubseteq \xi'$.

$t = \textbf{inl}_{\tau_2}(t_1)$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{inl}_{\tau_2}(t_1)) = \text{inl}_{\tau_2}(\widehat{t_1}) : \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \,\&\, \bot$ where $\widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_1)$.

By assumption, $\widehat{t'} = \text{inl}_{\tau_2}(\widehat{t'_1})$ with $\widehat{t'_1}$ conservative, and by lemma 2.35 and conjecture 3.56, there are conservative $\widehat{\tau}'_1$, $\widehat{\tau}'_2$ such that the following derivation for $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ is valid.

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'_1} : \widehat{\tau}'_1 \,\&\, \xi'_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{inl}_{\tau_2}(\widehat{t'_1}) : \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}'_2\langle \xi'_2 \rangle \,\&\, \bot}\,[\text{T-INL}] \qquad \begin{array}{c}\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}'_2\langle \xi'_2 \rangle \leqslant \widehat{\tau}' \\ \Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi'\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{inl}_{\tau_2}(\widehat{t'_1}) : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-SUB}]$$

By induction, $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}'_1$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi'_1$. By lemma 3.26, $\Sigma \vdash_{\text{sub}} \bot_{\tau_2} \leqslant \widehat{\tau}'_2$. Clearly, $\Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi'_2$. Thus, we can derive $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \leqslant \widehat{\tau}'_1\langle \xi'_1 \rangle + \widehat{\tau}'_2\langle \xi'_2 \rangle$ by [T-SUM].

By transitivity, we get $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi'$ trivially.

$t = \lambda x : \tau_1 . t_2$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \lambda x : \tau_1 . t') = \Lambda \overline{\beta_i :: \kappa_i} . \lambda x : \widehat{\tau}_1 \,\&\, \xi_1 . \widehat{t_2} : \forall \overline{\beta_i :: \kappa_i} . \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \,\&\, \bot$ such that $\emptyset \vdash_c \tau_1 : \widehat{\tau}_1 \,\&\, \xi_1 \rhd \overline{\beta_i :: \kappa_i}$ holds and $\widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2 = \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1; \Sigma, \overline{\beta_i :: \kappa_i}; t_2)$.

By assumption, we have $\widehat{t'} = \Lambda \overline{\beta_i :: \kappa_i} . \lambda x : \widehat{\tau}'_1 \,\&\, \xi'_1 . \widehat{t'_2}$ such that $\widehat{t'_2}$ is conservative and $\emptyset \vdash_p \widehat{\tau}'_1 \,\&\, \xi'_1 \rhd \overline{\beta_i :: \kappa_i}$ holds. Since pattern types are unique, we can assume that $\widehat{t'}$ quantifies over the same annotation variables $\overline{\beta_i}$. By lemma 2.35 and conjecture 3.56, there is a conservative type $\widehat{\tau}'_2$ such that $\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma}, x : \widehat{\tau}'_1 \,\&\, \xi'_1 \vdash_{\text{te}} \widehat{\tau}'_2 : \xi'_2 \,\&\, \cdot$ holds.

By induction, we then get $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi'_2$. As the argument position can be treated invariantly, this allows us to derive $\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}'_2\langle\xi'_2\rangle$. Using [SUB-FORALL], we arrive at the desired conclusion.

$t = \mu x : \tau.t_1$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \mu x : \tau.t_1) = \mu x : \widehat{\tau} \mathbin{\&} \xi.\widehat{t_1} : \widehat{\tau} \mathbin{\&} \xi$ for some $\widehat{\tau}$ and ξ. Using lemmas 3.47 and 3.54 we can express this result as the least fixpoint $[\widehat{\tau} \mathbin{\&} \xi]_\Sigma$ of the monotone function

$$f([\widehat{\tau}_1 \mathbin{\&} \xi_1]_\Sigma) = [\widehat{\tau}'_1 \mathbin{\&} \xi'_1]_\Sigma \text{ where } \widehat{t_1} : \widehat{\tau}'_1 \mathbin{\&} \xi'_1 = \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}_1 \mathbin{\&} \xi_1; \Sigma; t_1)$$

Note that due to the monotonicity of $\mathcal{R}$, the result of f does not depend on the choice of a particular representative of the equivalence class. This is because any representative is both a sub- and a supertype of all others, and so are the results of $\mathcal{R}$ by monotonicity. Thus, they are all in the same equivalence class as well.

By assumption, we have $\widehat{t'} = \mu x : \widehat{\tau}' \mathbin{\&} \xi'.\widehat{t'_1}$ such that $\widehat{t'_1}$ is conservative, and by lemma 2.35 and conjecture 3.56, there is the following derivation.

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}'' \mathbin{\&} \xi'' \vdash_{\text{te}} \widehat{t'_1} : \widehat{\tau}'' \mathbin{\&} \xi''}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau}'' \mathbin{\&} \xi''.\widehat{t'_1} : \widehat{\tau}'' \mathbin{\&} \xi''}\,[\text{T-FIX}] \quad \begin{array}{c}\Sigma \vdash_{\text{sub}} \widehat{\tau}'' \leqslant \widehat{\tau}' \\ \Sigma \vdash_{\text{sub}} \xi'' \sqsubseteq \xi'\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x : \widehat{\tau}'' \mathbin{\&} \xi''.\widehat{t'_1} : \widehat{\tau}' \mathbin{\&} \xi'}\,[\text{T-SUB}]$$

We define the sequence $(\widehat{\tau}''_i \mathbin{\&} \xi''_i)_{i\in\mathbb{N}}$ by

$$\begin{array}{rcl} [\widehat{\tau}''_0 \mathbin{\&} \xi''_0]_\Sigma & = & [\widehat{\tau}'' \mathbin{\&} \xi'']_\Sigma \\ [\widehat{\tau}''_{i+1} \mathbin{\&} \xi''_{i+1}]_\Sigma & = & f([\widehat{\tau}''_i \mathbin{\&} \xi''_i]_\Sigma) \end{array}$$

Claim: $[\widehat{\tau}''_{i+1} \mathbin{\&} \xi''_{i+1}]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}''_i \mathbin{\&} \xi''_i]_\Sigma$ for all i.

Proof: By induction on i.

$i = 0$ By using the outer induction hypothesis, $\mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}''_0 \mathbin{\&} \xi''_0; \Sigma; t_1)$ is a best analysis for t_1. Therefore, $[\widehat{\tau}''_1 \mathbin{\&} \xi''_1]_\Sigma = f([\widehat{\tau}''_0 \mathbin{\&} \xi''_0]_\Sigma) \sqsubseteq_\Sigma [\widehat{\tau}''_0 \mathbin{\&} \xi''_0]_\Sigma$.

$i \to i + 1$ By induction, we have

$$[\widehat{\tau}''_{i+2} \mathbin{\&} \xi''_{i+2}]_\Sigma = f([\widehat{\tau}''_{i+1} \mathbin{\&} \xi''_{i+1}]_\Sigma) \sqsubseteq_\Sigma f([\widehat{\tau}''_i \mathbin{\&} \xi''_i]_\Sigma) = [\widehat{\tau}''_{i+1} \mathbin{\&} \xi''_{i+1}]_\Sigma$$

■

Since the number of equivalence classes is finite, there must be some index j such that $[\widehat{\tau}''_j \mathbin{\&} \xi''_j]_\Sigma$ is a fixpoint of f. By lemma 3.54, $[\widehat{\tau} \mathbin{\&} \xi]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}''_j \mathbin{\&} \xi''_j]_\Sigma$ and by transitivity, $[\widehat{\tau} \mathbin{\&} \xi]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}'' \mathbin{\&} \xi'']_\Sigma$.

Thus, $\Sigma \vdash_{\text{sub}} \widehat{\tau} \leqslant \widehat{\tau}'$ and $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi'$ hold by transitivity.

The following case consists of a sketch for proving the result for function applications.

$t = t_1\ t_2$ The crucial part of this case is to show that the annotation applications computed by the matching algorithm are subsumed by those used in the other analysis $\widehat{t'}$. This can likely be shown by inspecting the subtyping relation obtained by applying the induction hypothesis to the argument term.

We think that the remaining cases, i.e. pairs, projections, seq and case-expressions, can be proven analogously. □

4. Implementation

This section briefly outlines the prototype implementation based on the algorithm presented in the previous chapter. It has been realized using Haskell with version 8.0.2 of GHC[1].

4.1. Additional Features

While the only base type in the analysis described in the previous chapters is unit, the prototype has additional built-in support for booleans and integers. Therefore, there is additional syntax for boolean literals (*true* and *false*), and for integer literals. Boolean values can be deconstructed using if-expressions $\textbf{if}\ c\ \textbf{then}\ t_1\ \textbf{else}\ t_2$.

The typing rules for literals can be straightforwardly derived from [T-Unit] and the typing rule for if-expressions is just [T-Case] without the additional variable bindings.

Note that there are no builtin operators, since these can be represented by providing a suitable initial type environment containing functions with the right type annotations.

Example 4.1. A basic set of functions for working with integers could consist of the following signatures.

$$
\begin{aligned}
&\mathit{plus} &&: \forall\beta_1.\mathrm{int}\langle\beta_1\rangle \to (\forall\beta_2.\mathrm{int}\langle\beta_2\rangle \to \mathrm{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot \\
&\mathit{minus} &&: \forall\beta_1.\mathrm{int}\langle\beta_1\rangle \to (\forall\beta_2.\mathrm{int}\langle\beta_2\rangle \to \mathrm{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot \\
&\mathit{mult} &&: \forall\beta_1.\mathrm{int}\langle\beta_1\rangle \to (\forall\beta_2.\mathrm{int}\langle\beta_2\rangle \to \mathrm{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot \\
&\mathit{eq} &&: \forall\beta_1.\mathrm{int}\langle\beta_1\rangle \to (\forall\beta_2.\mathrm{int}\langle\beta_2\rangle \to \mathrm{bool}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot \\
&\mathit{gt} &&: \forall\beta_1.\mathrm{int}\langle\beta_1\rangle \to (\forall\beta_2.\mathrm{int}\langle\beta_2\rangle \to \mathrm{bool}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot
\end{aligned}
$$

We can observe that all these functions follow the same pattern. There are two inputs and the result depends on both.

Interestingly, these types are valid for any choice of a concrete dependency lattice because they only mention the join operation and bottom. There are of course examples for builtin functions where this is not the case. Consider a function

$$\mathit{readInt} : \forall\beta_1.\widehat{\mathrm{unit}}\langle\beta_1\rangle \to \mathrm{int}\langle\mathbf{D}\rangle \ \&\ \mathbf{S}$$

that reads an integer from the standard input. In this case, we must provide a specific type signature for binding-time analysis in order to assign the non-bottom annotation **D** to the return value.

This method can also be used for providing primitive operations that cannot be expressed in the language. Consider an example in security analysis where we want to

[1] The Glasgow Haskell Compiler, `https://www.haskell.org/ghc/`

be able to declassify data if it meets certain requirements. This is not possible with the constructs provided by the language where we have only means for increasing the security level. We could however provide a function in the initial environment such as

$$declassify : \forall \beta_1 . A\langle \beta_1 \rangle \to A\langle \beta_1 \rangle + A\langle \bot \rangle \; \& \; \bot$$

which takes an argument of some type A and depending on whether it meets certain criteria returns it without classification ($\bot$), or with the same classification β_1 as before.

It is important to note though, that while such type signatures can be artificially added to the environment, there is a good reason why it is not possible to construct these within the language. Functions like these can be used to break noninterference, i.e. it is no longer evident from the type of an expression whether or not it may use certain variables from the context.

However, this is not a problem in our noninterference theorem because it only relates closed terms obtained by substituting the only free variable. Therefore, it only applies to those terms where there is actually a closed term that can be used for the substitution.

4.2. Project Structure

The project itself is structured into several modules that correspond to the different sections of this thesis.

We are effectively dealing with three different but related languages, the source language, the target language and the annotation language (the $\lambda^{\sqcup}$-calculus). Their respective implementations reside in the `DependencyAnalysis.Source`, `DependencyAnalysis.Target` and `DependencyAnalysis.LambdaJoin` namespaces, while using common definitions from the `DependencyAnalysis.Common` namespace. Each of these namesspaces contains an `AST` module containing the data types used for representing the abstract syntax trees. Furthermore, there are `Parser` and `Pretty` modules for languages that need to be parsed, respectively pretty printed.

Additionally, the $\lambda^{\sqcup}$-namespace also contains the modules `Normalization` implementing the canonicalization presented in the previous chapter, `Equality` providing functions for testing $\lambda^{\sqcup}$-terms for semantic or syntactic equality, and `Sorting` implementing a sort-inference for dependency terms.

The actual reconstruction algorithm $\mathcal{R}$ is realized in the `DependencyAnalysis.Reconstruction` module. So far, it assumes that it is provided with a well-typed underlying term. It makes use of the implementation of the completion rules provided in the module `DependencyAnalysis.Completion`.

4.3. REPL

While the analysis itself is developed as a library, the project also contains an executable providing a simple *read-eval-print-loop* (REPL) for analyzing source terms and computing type completions of underlying types. Only binding-time analysis and security analysis using the four element lattice presented in an earlier example have been

implemented. However, the implementation of the analysis is abstracted over the choice of a particular lattice. Therefore, new concrete analyses can be added independent of the reconstruction algorithm.

It needs to be considered that the current implementation uses the brute-force approach to deciding semantic equality (as outlined at the beginning of chapter 3) for the lack of a better alternative. Thus, using larger lattices will likely result in very slow computations.

The concrete syntax resembles the abstract syntax used in this thesis, using an ASCII representation of the symbolic notation. It is shown in figure 4.1 in an EBNF-like presentation. The join operation in the inferred annotations and effects is represented by the `+` symbol. The two values of the binding-time lattice are mapped to `S` (static) and `D` (dynamic) whereas the security lattice uses `L` (low), `M1`, `M2` (two unrelated medium classifications) and `H` (high).

In order to provide meaningful operations for the additional base types besides unit, the type and effect environment is prepopulated with the functions from example 4.1 as well as analogously defined binary functions `and` and `or` for booleans and additional order predicates `lt`, `leq`, `geq` and `neq`.

⟨*type*⟩ ::= ⟨*stype*⟩ | ⟨*stype*⟩ `->` ⟨*type*⟩ (function type)
⟨*stype*⟩ ::= ⟨*ptype*⟩ | ⟨*ptype*⟩ `+` ⟨*ptype*⟩ (sum type)
⟨*ptype*⟩ ::= ⟨*btype*⟩ | ⟨*btype*⟩ `*` ⟨*btype*⟩ (product type)
⟨*btype*⟩ ::= `bool` | `int` | `unit` | `(`⟨*type*⟩`)` (base types)

⟨*ident*⟩ ::= [a − z][a − zA − Z0 − 9_']∗

⟨*latticeval*⟩ ::= *depends on lattice*

⟨*term*⟩ ::= ⟨*ident*⟩ (variable)
| `()` (unit constructor)
| `fun` ⟨*ident*⟩ `:` ⟨*type*⟩ `=>` ⟨*term*⟩ (λ-abstraction)
| `fix` ⟨*ident*⟩ `:` ⟨*type*⟩ `=>` ⟨*term*⟩ (fixpoint)
| ⟨*term*⟩ ⟨*term*⟩ (application)
| `(`⟨*term*⟩`,` ⟨*term*⟩`)` (pair constructor)
| `fst(`⟨*term*⟩`)` | `snd(`⟨*term*⟩`)` (pair projections)
| `true` | `false` (boolean constructors)
| `if` ⟨*term*⟩ `then` ⟨*term*⟩ `else` ⟨*term*⟩ (bool elimination)
| inl<⟨*type*⟩>(⟨*term*⟩) | inr<⟨*type*⟩>(⟨*term*⟩) (sum constructors)
| `case` ⟨*term*⟩ `of` { `inl(`⟨*ident*⟩`) ->` ⟨*term*⟩ (sum elimination)
`; inr(`⟨*ident*⟩`) ->` ⟨*term*⟩ }
| `seq(`⟨*term*⟩`,` ⟨*term*⟩`)` (forcing)
| ann<⟨*latticeval*⟩>(⟨*term*⟩) (annotations)

Figure 4.1.: Concrete syntax of the source language

5. Evaluation

In this chapter we evaluate the type reconstruction algorithm and the type system itself by means of some examples. At the end of chapter 2 we have already seen how the underlying lattice can be chosen in order to specialize the algorithm to a certain analysis.

The examples here use the additional types provided by our implementation, as presented in the previous chapter.

5.1. Construction and Elimination

We start with a few simple examples demonstrating that the effects inferred for constructors and eliminators are sensible. We use the terms *constructors* and *eliminators* deliberately here, because the basic concept generalizes over product types, sum types and function types.

Whenever something is constructed, be it a product, a sum or a lambda abstraction, the outermost effect that is assigned is $\bot$. This is because the analysis aims to produce the best possible and thereby least annotations for a given source program.

On the other hand, when eliminating through projection, case distinction or function application, the effect of the term that is eliminated is included in the result. This is a necessary requirement for the analysis results to be sound.

Consider the case of binding-time analysis, and suppose we have a variable of function type $f : \forall\beta.\text{int}\langle\beta\rangle \to \text{int}\langle\beta\rangle \& \mathbf{D}$. We see that it preserves the annotations of its arguments, i.e. if we apply f to a static value, the return annotation is also instantiated to be static. The function itself, however, is dynamic. And therefore, the whole result of the function application must also be dynamic, because we cannot know which particular function has been assigned to f.

As elimination always introduces a dependency in the program, this can uncover subtleties arising when functions only differ in their termination behavior. For example, compare $\lambda p : \text{int} \times \text{int}.p$ with $\lambda p : \text{int} \times \text{int}.(\text{proj}_1(p), \text{proj}_2(p))$.

In a call-by-value language, these two functions would be (extensionally) equivalent. However, with non-strict evaluation, p might be a non-terminating computation. In that case, applying the former function would diverge, while the latter function at least produces the pair constructor. This is also reflected in the annotated types that are inferred for the above functions,

$$\forall\beta_0, \beta_1, \beta_2 :: \star.(\text{int}\langle\beta_0\rangle \times \text{int}\langle\beta_1\rangle)\langle\beta_2\rangle \to (\text{int}\langle\beta_0\rangle \times \text{int}\langle\beta_1\rangle)\langle\beta_2\rangle \;\&\; \mathbf{S}$$

for the former and

$$\forall\beta_0, \beta_1, \beta_2 :: \star.(\text{int}\langle\beta_0\rangle \times \text{int}\langle\beta_1\rangle)\langle\beta_2\rangle \to (\text{int}\langle\beta_0 \sqcup \beta_2\rangle \times \text{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\mathbf{S}\rangle \;\&\; \mathbf{S}$$

for the latter. In particular, the annotation of the product in the second type signature is **S**. Therefore, it can not depend on the input of the function.

5.2. Polymorphic Recursion

One class of functions where the analysis benefits from polymorphic recursion are those that permute their arguments on recursive calls.

Example 5.1. This example is a slightly modified version of the motivating example of Dussart, Henglein, and Mossin [4] for polymorphic recursion in binding-time analysis.

$$\mu f : \text{bool} \to \text{bool} \to \text{bool}.\lambda x : \text{bool}.\lambda y : \text{bool}.\ \textbf{if}\ x\ \textbf{then}\ \mathit{true}\ \textbf{else}\ f\ y\ x$$

In an analysis with monomorphic recursion, this would force the analysis to assign the same annotations to both parameters. It must be large enough (in terms of the lattice order) to accommodate for both arguments. This is due to the permutation of the arguments in the else branch, where the same instantiation of f must be used as in the initial call.

On the other hand, an analysis with polymorphic recursion is allowed to use a different instantiation for f in that case. Our algorithm hence infers the following most general type.

$$\forall \beta_1 :: \star.\widehat{\text{bool}}\langle\beta_1\rangle \to (\forall \beta_2 :: \star.\widehat{\text{bool}}\langle\beta_2\rangle \to \widehat{\text{bool}}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle \ \&\ \bot$$

We see that the result of the function indeed depends on the annotations of both arguments, as both end up in the condition of the if-expression at some point. Yet, both arguments are completely unrestricted, and unrelated in their annotations.

The instantiation of the recursive call is explicitly visible in the target language term generated by the analysis.

$$\begin{aligned}&\mu f : (\forall \beta_1 :: \star.\widehat{\text{bool}}\langle\beta_1\rangle \to (\forall \beta_2 :: \star.\widehat{\text{bool}}\langle\beta_2\rangle \to \widehat{\text{bool}}\langle\beta_1 + \beta_2\rangle)\langle\bot\rangle)\ \&\ \bot.\\ &\quad \Lambda\beta_1 :: \star.\lambda x : \widehat{\text{bool}}\ \&\ \beta_1.\Lambda\beta_2 :: \star.\lambda y : \widehat{\text{bool}}\ \&\ \beta_2.\ \textbf{if}\ x\ \textbf{then}\ \mathit{true}\ \textbf{else}\ f\ \langle\beta_2\rangle\ y\ \langle\beta_1\rangle\ x\end{aligned}$$

A type that is polymorphic in the annotations of the arguments is assigned to the fixpoint binding and the quantification is explicit on the term level. We can observe that β_2 is passed to the formal parameter β_1.

In contrast, a type system with monomorphic recursion would only admit a weaker type, possibly similar to the following, where the annotations of both arguments must be the same.

$$\forall \beta_1 :: \star.\widehat{\text{bool}}\langle\beta_1\rangle \to (\widehat{\text{bool}}\langle\beta_1\rangle \to \widehat{\text{bool}}\langle\beta_1\rangle)\langle\bot\rangle \ \&\ \bot$$

A real world example where the annotations of the arguments eventually end up in both formal parameters is Euclid's algorithm for computing the greatest common divisor.

Example 5.2. We can formulate the *gcd* algorithm in our language as follows, assuming that the context is already populated with the function *minus* for performing subtraction and the order predicates *eq* (for equality) and *gt* (for greater than). What these auxiliary definitions would look like is outlined in the previous chapter (see example 4.1).

$$\begin{aligned}&\mu gcd : \text{int} \to \text{int} \to \text{int}.\lambda a : \text{int}.\lambda b : \text{int}.\\&\quad(\textbf{if } eq\ a\ b \textbf{ then } a \textbf{ else if } gt\ a\ b \textbf{ then } gcd\ (minus\ a\ b)\ b \textbf{ else } gcd\ a\ (minus\ b\ a))\end{aligned}$$

In one branch of the inner if-expression, *gcd*'s first parameter is instatiated with the join of the annotations of both arguments (due to the type signature of *minus*). In the other branch, this is the case for the second parameter of *gcd*.

Again, this becomes visible in the annotated type signature and the elaborated term.

$$\begin{aligned}&gcd : \forall\beta_1 :: \star.\text{int}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\text{int}\langle\beta_2\rangle \to \text{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle\ \&\ \bot\\&gcd = \mu gcd : \forall\beta_1 :: \star.\text{int}\langle\beta_1\rangle \to (\forall\beta_2 :: \star.\text{int}\langle\beta_2\rangle \to \text{int}\langle\beta_1 \sqcup \beta_2\rangle)\langle\bot\rangle\ \&\ \bot.\\&\quad\Lambda\beta_1 :: \star.\lambda a : \text{int}\ \&\ \beta_1.\Lambda\beta_2 :: \star.\lambda b : \text{int}\ \&\ \beta_1.\\&\qquad(\textbf{if } eq\ \langle\beta_1\rangle\ a\ \langle\beta_2\rangle\ b \textbf{ then } a \textbf{ else}\\&\qquad\quad\textbf{if } gt\ \langle\beta_1\rangle\ a\ \langle\beta_2\rangle\ b \textbf{ then } gcd\ \langle\beta_1 \sqcup \beta_2\rangle\ (minus\ \langle\beta_1\rangle\ a\ \langle\beta_2\rangle\ b)\ \langle\beta_2\rangle\ b\\&\qquad\qquad\qquad\qquad\qquad\qquad\textbf{else}\ \ gcd\ \langle\beta_1\rangle\ a\ \langle\beta_1 \sqcup \beta_2\rangle\ (minus\ \langle\beta_2\rangle\ b\ \langle\beta_1\rangle\ a))\end{aligned}$$

5.3. Higher-Ranked Polyvariance

This section discusses several examples where the analysis benefits from higher-ranked polyvariance. In particular, we compare our results to existing analyses that are not higher-ranked. The situations in which higher-ranked analyses lead to advantages are those where arguments of function types are used more than once.

In the following example we illustrate the gains of using our higher-ranked system in the context of binding-time analysis by comparing it to the polyvariant, but not higher-ranked binding time analysis by Zhang [26]. Note that Zhang's system is constraint based, while we are using a more direct formulation.

Example 5.3. A simple example to start with is a function that applies a function to both components of a pair.

$$\begin{aligned}&both : (\text{int} \to \text{int}) \to \text{int} \times \text{int} \to \text{int} \times \text{int}\\&both = \lambda f : \text{int} \to \text{int}.\lambda p : \text{int} \times \text{int}.(f\ (\text{proj}_1(p)), f\ (\text{proj}_2(p)))\end{aligned}$$

Suppose in the context of binding-time analysis that *both* is used to apply a statically known function to a pair whose the first component is always computable at compile time, but whose second component is dynamic.

For simplicity's sake, we will use the identity function as an argument to *both*.

$$\begin{aligned}&id : \text{int} \to \text{int}\\&id = \lambda x : \text{int}.x\end{aligned}$$

The non-higher-ranked analysis would assign the following types to our example functions.

$$\begin{aligned}
&\mathit{id} \quad : \forall \beta_1\ \beta_2.(\beta_1 \sqsubseteq \beta_2, \beta_3 \sqsubseteq \beta_1, \beta_3 \sqsubseteq \beta_2) \Rightarrow \mathrm{int}\langle\beta_1\rangle \xrightarrow{\beta_3} \mathrm{int}\langle\beta_2\rangle \\
&\mathit{both} : \forall \beta_1 \ldots \beta_{10}.(\beta_3 \sqsubseteq \beta_1, \beta_3 \sqsubseteq \beta_2, \beta_4 \sqsubseteq \beta_3, \beta_4 \sqsubseteq \beta_8, \beta_7 \sqsubseteq \beta_5, \beta_7 \sqsubseteq \beta_6, \beta_8 \sqsubseteq \beta_7 \\
&\qquad\quad , \beta_8 \sqsubseteq \beta_{11}, \beta_{11} \sqsubseteq \beta_9, \beta_{11} \sqsubseteq \beta_{10}, \beta_5 \sqsubseteq \beta_1, \beta_6 \sqsubseteq \beta_1, \beta_2 \sqsubseteq \beta_9, \beta_2 \sqsubseteq \beta_{10}) \\
&\quad \Rightarrow (\mathrm{int}\langle\beta_1\rangle \xrightarrow{\beta_3} \mathrm{int}\langle\beta_2\rangle) \xrightarrow{\beta_4} (\mathrm{int}\langle\beta_5\rangle \times \mathrm{int}\langle\beta_6\rangle)\langle\beta_7\rangle \xrightarrow{\beta_8} (\mathrm{int}\langle\beta_9\rangle \times \mathrm{int}\langle\beta_{10}\rangle)\langle\beta_{11}\rangle
\end{aligned}$$

The limiting factor in the expressiveness of the analysis is the fact that the annotations β_1 and β_2 of the return value of the function argument to *both* are chosen wherever *both* is called. In order for this instantiation to be valid for all occurrences of the argument f in the body of *both*, the constraints $\beta_5 \sqsubseteq \beta_1$, $\beta_6 \sqsubseteq \beta_1$, $\beta_2 \sqsubseteq \beta_9$ and $\beta_2 \sqsubseteq \beta_{10}$ must be fulfilled. The first two constraints state that the annotation of the function's argument must be large enough to be able to take both components of the pair as input. The latter two ensure that the annotations of the returned pair are at least as large as whatever the function returns.

When we look at the partial application *both id*, we have the additional constraint $\beta_1 \sqsubseteq \beta_2$ from the type of *id*. This forces β_9 and β_{10} to be larger than both β_5 and β_6. Consider the call *both id p* for some pair $p : \mathrm{int}\langle\mathbf{S}\rangle \times \mathrm{int}\langle\mathbf{D}\rangle \ \&\ \mathbf{S}$. Then, $\beta_5 = \mathbf{S}$ and $\beta_6 = \mathbf{D}$, but even though we are applying the identity function, the whole call has the type $\mathrm{int}\langle\mathbf{D}\rangle \times \mathrm{int}\langle\mathbf{D}\rangle$.

In our higher-ranked analysis, we can infer the following conservative type for the identity function.

$$\begin{aligned}
&\mathit{id} : \forall \beta :: \star.\mathrm{int}\langle\beta\rangle \to \mathrm{int}\langle\beta\rangle \ \&\ \bot \\
&\mathit{id} = \Lambda\beta :: \star.\lambda x : \mathrm{int} \ \&\ \beta.x
\end{aligned}$$

Because *id* is not a higher-order function, we have not yet gained more precision at this point. However, we obtain the following target term and annotated type for the function *both*.

$$\begin{aligned}
&\mathit{both} : \forall \beta_1 :: \star.\forall \beta_2 :: \star \Rightarrow \star.(\forall \beta :: \star.\mathrm{int}\langle\beta\rangle \to \mathrm{int}\langle\beta_2\ \beta\rangle)\langle\beta_1\rangle \\
&\quad \to (\forall \beta_3, \beta_4, \beta_5 :: \star.(\mathrm{int}\langle\beta_3\rangle \times \mathrm{int}\langle\beta_4\rangle)\langle\beta_5\rangle \\
&\quad \to (\mathrm{int}\langle\beta_2\ (\beta_3 \sqcup \beta_5) \sqcup \beta_1\rangle \times \mathrm{int}\langle\beta_2\ (\beta_4 \sqcup \beta_5) \sqcup \beta_1\rangle)\langle\mathbf{S}\rangle)\langle\mathbf{S}\rangle \ \&\ \mathbf{S} \\
&\mathit{both} = \Lambda\beta_1 :: \star.\Lambda\beta_2 :: \star \Rightarrow \star.\lambda f : (\forall \beta :: \star.\mathrm{int}\langle\beta\rangle \to \mathrm{int}\langle\beta_2\ \beta\rangle). \\
&\quad \Lambda\beta_3 :: \star.\Lambda\beta_4 :: \star.\Lambda\beta_5 :: \star.\lambda p : \mathrm{int}\langle\beta_3\rangle \times \mathrm{int}\langle\beta_4\rangle. \\
&\qquad (f\ \langle\beta_3 \sqcup \beta_5\rangle\ (\mathrm{proj}_1(p)), f\ \langle\beta_4 \sqcup \beta_5\rangle\ (\mathrm{proj}_2(p)))
\end{aligned}$$

The function parameter f can be instantiated separately for each component because our analysis assigns it a type that universally quantifies over the annotation of its argument. It is evident from the type signature that the components of the resulting pair only depend on the corresponding components of the input pair, and the function and the input pair itself. They do not depend on the respective other component of the input.

If we again consider the call *both id p*, we obtain $\beta_2 = \lambda\beta :: \star.\beta$, $\beta_1 = \beta_3 = \beta_5 = \mathbf{S}$ and $\beta_4 = \mathbf{D}$ through pattern unification. Normalization of the resulting dependency terms results in the expected return type $\mathrm{int}\langle\mathbf{S}\rangle \times \mathrm{int}\langle\mathbf{D}\rangle$.

The generality provided by the higher-ranked analysis extends to an arbitrarily deep nesting of function arrows. The following example demonstrates this for two levels of arrows.

Example 5.4. Consider the following function that takes a function argument which again requires a function.

$$\begin{aligned}&\mathit{foo} : ((\text{int} \to \text{int}) \to \text{int}) \to \text{int} \times \text{int}\\&\mathit{foo} = \lambda f : (\text{int} \to \text{int}) \to \text{int}.(f\ (\lambda x : \text{int}.x), f\ (\lambda x : \text{int}.0))\end{aligned}$$

The higher-ranked analysis infers the following type and target term (where we omitted the type in the argument of the lambda term because it essentially repeats what is already visible in the top level type signature).

$$\begin{aligned}&\mathit{foo} : \forall\beta_4 :: \star.\forall\beta_3 :: \star \Rightarrow (\star \Rightarrow \star) \Rightarrow \star.\\&\quad(\forall\beta_2 :: \star.\forall\beta_1 :: \star \Rightarrow \star.(\forall\beta_0 :: \star.\text{int}\langle\beta_0\rangle \to \text{int}\langle\beta_1\ \beta_0\rangle)\langle a_2\rangle \to \text{int}\langle\beta_3\ \beta_2\ \beta_1\rangle)\langle\beta_4\rangle\\&\quad\to (\text{int}\langle\beta_3\ \mathbf{S}\ (\lambda\beta_5 :: \star.\beta_5) \sqcup \beta_4\rangle \times \text{int}\langle\beta_3\ \mathbf{S}\ (\lambda\beta_6 :: \star.\mathbf{S}) \sqcup \beta_4\rangle)\langle\mathbf{S}\rangle \ \&\ \mathbf{S}\\&\mathit{foo} = \Lambda\beta_4 :: \star.\Lambda\beta_3 :: \star \Rightarrow (\star \Rightarrow \star) \Rightarrow \star.\lambda f : \cdots .\\&\quad(f\ \langle\mathbf{S}\rangle\ \langle\lambda\beta_0 :: \star.\beta_0\rangle\ (\Lambda\beta_5 :: \star.\lambda x : \text{int}\ \&\ \beta_5.x)\\&\quad, f\ \langle\mathbf{S}\rangle\ \langle\lambda\beta_0 :: \star.\mathbf{S}\rangle\ \ (\Lambda\beta_6 :: \star.\lambda x : \text{int}\ \&\ \beta_6.1))\end{aligned}$$

Since the type of f is a pattern type, the argument to f is also a pattern type by definition. Therefore, the analysis of f depends on the analysis of the function passed to it. This gives rise to the *higher-order effect operator* β_3. Thus, f can be applied to any function with a conservative type of the right shape. As our type reconstruction algorithm always infers conservative types, the type of f is as general as possible.

This is reflected in the body of the lambda where in both cases f is instantiated with the effect corresponding to the function passed to it. The result of this instantiation can be observed in the returned product type where β_3 is applied to the effect operators $\lambda\beta_0 :: \star.\beta_0$ and $\lambda\beta_0 :: \star.\mathbf{S}$ corresponding to the respective functions used as arguments to f.

Only when we finally apply foo, the resulting annotations can be evaluated. We consider three simple functions that differ in their behavior.

$$\begin{aligned}&\mathit{bar}_1 : \forall a_2 :: \star.\forall a_1 :: \star \Rightarrow \star.(\forall a0 :: \star.\text{int}\langle a0\rangle \to \text{int}\langle a_1\ a0\rangle)\langle a_2\rangle \to \text{int}\langle\mathbf{S}\rangle \ \&\ \mathbf{S}\\&\mathit{bar}_1 = \Lambda a_2 :: \star.\Lambda a_1 :: \star \Rightarrow \star.\lambda f : \cdots .0\\&\mathit{bar}_2 : \forall a_2 :: \star.\forall a_1 :: \star \Rightarrow \star.(\forall a0 :: \star.\text{int}\langle a0\rangle \to \text{int}\langle a_1\ a0\rangle)\langle a_2\rangle \to \text{int}\langle a_1\ \mathbf{S} \sqcup a_2\rangle \ \&\ \mathbf{S}\\&\mathit{bar}_2 = \Lambda a_2 :: \star.\Lambda a_1 :: \star \Rightarrow \star.\lambda f : \cdots .f\ 0\\&\mathit{bar}_3 : \forall a_2 :: \star.\forall a_1 :: \star \Rightarrow \star.(\forall a0 :: \star.\text{int}\langle a0\rangle \to \text{int}\langle a_1\ a0\rangle)\langle a_2\rangle \to \text{int}\langle a_1\ \mathbf{D} \sqcup a_2\rangle \ \&\ \mathbf{S}\\&\mathit{bar}_3 = \Lambda a_2 :: \star.\Lambda a_1 :: \star \Rightarrow \star.\lambda f : \cdots .f\ (\text{ann}_{\mathbf{D}}(0))\end{aligned}$$

We obtain the following types when applying *foo* to the above functions.

$\mathit{foo}\ \mathit{bar}_1 : \textbf{int}\langle\mathbf{S}\rangle \times \textbf{int}\langle\mathbf{S}\rangle \ \&\ \mathbf{S}$ The effect operator β_3 is instantiated to the constant function $\lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\mathbf{S}$. Therefore, the components are annotated with **S** regardless of the arguments to β_3.

$foo\ bar_2 : \textbf{int}\langle \mathbf{S} \rangle \times \textbf{int}\langle \mathbf{S} \rangle \ \&\ \mathbf{S}$ Even though $\beta_3 = \lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\beta_1\ \mathbf{S} \sqcup \beta_2$ makes use of its arguments, the particular effect operators it is applied to still cause the result to be static. We have

$$(\lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\beta_1\ \mathbf{S} \sqcup \beta_2)\ \mathbf{S}\ (\lambda\beta :: \star.\beta) = (\lambda\beta :: \star.\beta)\ \mathbf{S} \sqcup \mathbf{S} = \mathbf{S}$$

and

$$(\lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\beta_1\ \mathbf{S} \sqcup \beta_2)\ \mathbf{S}\ (\lambda\beta :: \star.\mathbf{S}) = (\lambda\beta :: \star.\mathbf{S})\ \mathbf{S} \sqcup \mathbf{S} = \mathbf{S}.$$

$foo\ bar_3 : \textbf{int}\langle \mathbf{D} \rangle \times \textbf{int}\langle \mathbf{S} \rangle \ \&\ \mathbf{S}$ Now $\beta_3 = \lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\beta_1\ \mathbf{D} \sqcup \beta_2$ because bar_3 applies its argument to a value with dynamic binding time.

This causes the first component of the returned pair to be deemed dynamic as well. This becomes evident when we reduce the resulting annotation:

$$(\lambda\beta_2 :: \star.\lambda\beta_1 :: \star \Rightarrow \star.\beta_1\ \mathbf{D} \sqcup \beta_2)\ \mathbf{S}\ (\lambda\beta :: \star.\beta) = (\lambda\beta :: \star.\beta)\ \mathbf{D} \sqcup \mathbf{S} = \mathbf{D}$$

where $\lambda\beta :: \star.\beta$ is of course the effect operator belonging to the identity function passed to $bar3$.

On the other hand, in the second component bar_3 is applied to a constant function. Thus, regardless of the argument's dynamic binding time, the resulting binding time is static.

In the rank-1 type and effect system, we can assign the following type to *foo*:

$$\begin{aligned}&\forall \beta_1 \mathinner{..} \beta_8 :: \star.(\beta_1 \sqsubseteq \beta_2, \mathbf{S} \sqsubseteq \beta_2, \beta_3 \sqsubseteq \beta_2, \beta_5 \sqsubseteq \beta_4, \beta_4 \sqsubseteq \beta_6, \beta_4 \sqsubseteq \beta_7)\\ &\quad \Rightarrow ((\text{int}\langle\beta_1\rangle \to \text{int}\langle\beta_2\rangle)\langle\beta_3\rangle \to \text{int}\langle\beta_4\rangle)\langle\beta_5\rangle \to (\text{int}\langle\beta_6\rangle \times \text{int}\langle\beta_7\rangle)\langle\beta_8\rangle\end{aligned}$$

The first two constraints arise from the lambda expressions that are passed as arguments to f. The next constraints, $\beta_3 \sqsubseteq \beta_2$ and $\beta_5 \sqsubseteq \beta_4$ come from the well-formedness condition on function types. That is, the functions result must be as dynamic as the function itself. Lastly, $\beta_4 \sqsubseteq \beta_6$ and $\beta_4 \sqsubseteq \beta_7$ propagate the results of the function calls to the components of the pair.

The increased precision becomes visible in the third example. We can assign the following type to bar_3:

$$\begin{aligned}&\forall \beta_1\ \beta_2\ \beta_3\ \beta_4 :: \star.(\beta_3 \sqsubseteq \beta_4, \beta_2 \sqsubseteq \beta_4, \mathbf{D} \sqsubseteq \beta_1)\\ &\quad \Rightarrow (\text{int}\langle\beta_1\rangle \to \text{int}\langle\beta_2\rangle)\langle\beta_3\rangle \to \text{int}\langle\beta_4\rangle\end{aligned}$$

For determining the type of $foo\ bar_3$, we instantiate the latter accordingly resulting in the following constraints in particular: $\mathbf{D} \sqsubseteq \beta_1$, $\beta_1 \sqsubseteq \beta_2$, $\beta_2 \sqsubseteq \beta_4$, $\beta_4 \sqsubseteq \beta_6$ and $\beta_4 \sqsubseteq \beta_7$. The resulting type therefore is $\text{int}\langle\mathbf{D}\rangle \times \text{int}\langle\mathbf{D}\rangle$.

Such functions with more than two levels of arrows can arise in actual programs, e.g. as the result of replacing type classes in Haskell with explicit dictionary passing. The above example can be seen as a heavily restricted instance of this, due to the lack of user-defined data types and polymorphism in the underlying type system.

6. Related Work

In preparation for this thesis we also looked at *region analysis*, in particular the system by Tofte and Talpin used in the MLKit compiler [24], and the automated amortized *resource analysis* by Jost et al.[12].

6.1. Region Analysis

Region-based memory management strikes a middle-ground between garbage collection and manual memory management in terms of memory safety and determinism. The idea is to have (usually) lexically scoped regions in which memory can be allocated throughout their lifetime, but it is only freed at the end of the region's scope.

While it would be possible for a programmer to manually specify regions, that approach is only marginally less unsafe and time-consuming than managing memory entirely manually.

Therefore, region analysis is used for automatically determining the places in the source program where memory allocation and deallocation should take place. In order to prevent use-after-free errors, it must be ensured that a region will always outlive any uses of references to the data inside.

Although region analysis initially seemed like a promising candidate, several problems appeared upon closer investigation. It is uncertain whether the approach to higher-ranked polyvariance taken in this thesis and the work it is based on is actually applicable to region analysis.

There is a major difference between the flow and exception analyses for which the higher-ranked system had already been formulated, and region analysis. The former two compute the annotations of function effects and return values solely in terms of the arguments. They do not impose any restrictions on the annotations of the arguments of a function. On the contrary, in region analysis the annotations of the arguments are constrained by the function implementation.

Example 6.1. The following function definitions serve as examples for the problems we were facing. The actual type signature for h is intentionally left open, because it turned out that there is no clear or straightforward way for assigning one based on the higher-ranked frameworks we investigated. The region annotations and effects of function types have been omitted in order to increase readability, as they are not essential for demonstrating the problem.

$$f :: \forall \rho_1\ \rho_2 . \mathrm{int}\langle \rho_1 \rangle \to \mathrm{int}\langle \rho_2 \rangle \to \mathrm{int}\langle \rho_2 \rangle \to \mathrm{int}\langle \rho_2 \rangle$$
$$f\ c\ x\ y = \mathbf{if}\ c > 0\ \mathbf{then}\ x\ \mathbf{else}\ y$$

$g :: \forall \rho_1\ \rho_2\ \rho_3\ \rho_4 . \mathrm{int}\langle\rho_1\rangle \to \mathrm{int}\langle\rho_2\rangle \to \mathrm{int}\langle\rho_3\rangle \to \mathrm{int}\langle\rho_4\rangle$
$g\ c\ x\ y = \mathbf{if}\ c > 0\ \mathbf{then}\ x + y\ \mathbf{else}\ x - y$
$h :: ?$
$h\ k\ x = k\ 1\ x\ 2$

Because f returns either x or y unchanged, the return value of f and the last two arguments must live in the same region. On the other hand, function g does not suffer from the same problem, because it only reads its arguments and produces a fresh result that is stored in a region chosen by the caller.

Of course, in this example integers would likely be stored on the stack, but the problem persists for larger types. One could also imagine implicitly copying the values in the body of f, but similarly, this hides program behavior from the programmer at best, and leads to performance problems with larger values at worst.

The last problem is assigning a proper type signature to h that would allow it to be applied to f as well as g. We need a way of expressing certain constraints involving the annotations of arguments. But then, the types would lose their flexibility, which is a crucial invariant of the type reconstruction algorithm in its current form.

We concluded that the techniques by Holdermans and Hage and Koot cannot be readily applied to region analysis due to the necessity of imposing additional constraints on the annotations of arguments. Of course, it should be noted that this does not rule out the existence of other approaches that allow higher-ranked polyvariance to be used in region analysis.

6.2. Resource Analysis

Resource analysis also looked like a good candidate for extending it to a higher-ranked analysis. However, we refrained from further investigating this topic, not for technical reasons, but because there is already an analysis by Hoffmann, Das, and Weng[8] delivering results akin to what we would expect from higher-ranked polyvariance. It has been implemented in Resource Aware ML[1]. Notably, it is more precise than a previous implementation of the analysis in the functional programming language Hume[2] with higher-order functions, but only rank-1 polyvariance following the 2010 paper by Jost et al. [12].

However, their type system differs in quite a few points from the general approach taken in this thesis. Therefore, it might still be interesting to study the differences and the respective advantages and disadvantages of either method.

[1] RAML can be interactively tried at `http://raml.co/interface.html` (last visited 07-06-2017).
[2] Hume can be interactively tried at `http://www-fp.cs.st-andrews.ac.uk/embounded/software/cost/cost.cgi` (last visited 07-06-2017).

7. Conclusion and Future Work

We presented a type and effect system for dependency analyses employing higher-ranked polyvariance together with polymorphic recursion. This increased the precision compared to rank-1 polyvariance. It can be seen as an extension of the dependency core calculus by Abadi et al. [1] and is therefore general enough to cover various analyses such as binding-time analysis, security analysis, exception analysis and more.

The system has been formulated for a lazy functional language with general recursion. Moreover, there are data types for sums, products and a unit value. It is modeled after the language used to represent the dependency core calculus. We distinguish the *source* language and the *target* language. The source language is used in the formulation of the underlying type system. The target language is used for the type and effect system and provides explicit terms for abstracting and instantiating effects. This is similar to System-F, where abstraction over type variables and their subsequent instantiation is likewise reflected on the term level.

We adopted the use of effect operators which have been introduced by Holdermans and Hage in order to make the analysis of a function generic in its arguments. They have been formalized for exception analysis by Koot in the $\lambda^{\cup}$-calculus, a simply typed lambda calculus enriched with an algebraic set structure consisting of singleton sets and unions. We generalized this concept to the $\lambda^{\sqcup}$-calculus that works with an arbitrary lattice instead.

A main result of the first part of this thesis is the *noninterference* theorem with respect to a call-by-name operational semantics for the target language. Noninterference guarantees that the evaluation to WHNF of a term with a certain effect is not influenced by a variable it depends on, if the effect of said variable is not encompassed by the term's effect. For that reason, it is an important result for the correctness of the type and effect system and for subsequent usage of the analysis results.

As a preparation for the type reconstruction algorithm, we showed that deciding semantic equality of $\lambda^{\sqcup}$-terms is decidable using the fact that the underlying lattice is finite. We then present canonical forms for $\lambda^{\sqcup}$-terms using reduction and a fixed ordering of terms. These canonical forms are based on Koot's work and were supposed to be used in conjunction with syntactic equality (up to α-equivalence) in order to efficiently decide semantic equality. However, it turned out that in our system as well as in the prior work, there are certain terms with different canonical forms that are still semantically equivalent.

We then proceeded by introducing certain kinds of annotated types, namely *pattern* types and *conservative* types. They are essential for the modularity of the analysis and have been previously used by Koot and Holdermans and Hage (calling them *fully parametric* and *fully flexible* respectively). Conservative types are, in a certain sense,

the most general types possible corresponding to a given underlying type. Pattern types can be instantiated to any conservative type of the same shape using pattern unification, a restricted form of higher-order unification. Any function parameters in a conservative type are required to be pattern types, thereby making the least assumptions about the arguments. It is possible to automatically compute the pattern type corresponding to a certain underlying type. This type completion is used in the inference algorithm for assigning the argument types of lambda expressions.

We continued by developing a type reconstruction algorithm for our type and effect system based on the prior work by Koot. It works directly on the syntactic structure of the input terms and does not use constraint solving. This is in contrast to the earlier approach by Holdermans and Hage. Polymorphic recursion is handled through Mycroft fixpoint interation.

Furthermore, we proved the correctness of the reconstruction algorithm. Notably, its results are typeable in the type and effect system, and it will always terminate with a correct answer for any well-typed source term. As a corollary, our type and effect system is a conservative extension of the underlying type system, i.e. every well-typed source term is analyzable. Moreover, the types computed by the reconstruction algorithm are always the least ones, under some restrictions.

Additionally, we realized a prototype implementation of the type reconstruction algorithm for the simple functional language, but extended with integers and booleans in order to allow for more meaningful examples. Note that due to the inefficient method that is used for deciding semantic equality of annotations, the performance of the implementation is rather poor.

An evaluation, where we compared our higher-ranked system to analyses employing rank-1 polyvariance, showed that we can indeed expect precision gains from a higher-ranked analysis. However, due to the simplicity of the language, we could not investigate the impact on real world programs.

Finally, there are several directions in which the topic of this thesis could be developed, including the aforementioned shortcomings.

First of all, the language we have used as a base for our analysis is far away from a fully-fledged functional language such as Haskell. Notably, it is missing support for recursive data types, type polymorphism and user defined data types. A particular challenge with the latter two features is likely to be the handling of annotations hidden inside a type variable or type constructor.

Furthermore, deciding semantic equality of dependency terms is slow due to using a brute force approach that does not scale well to large lattices and programs. One starting point could be the investigation of stronger guarantees than just monotonicity about functions expressible in the $\lambda^{\sqcup}$-calculus. These, in turn, could be used for defining more extensive reduction rules. Alternatively, there might be ways of improving the performance of the current brute-force approach, e.g. by pruning the search space in some way.

Another approach to improving the performance of inference for recursive bindings might be the usage of *accelerated Kleene-Mycroft iteration*, as described by Dussart, Henglein, and Mossin. In order to reduce the cost of fixpoint iteration in the presence

of nested recursive bindings, they initialize the fixpoint iteration with the result of the most recent execution instead of bottom [4].

Orthogonal to the previous issues is the addition of minimal typing derivations [2]. While the most general analyses provided through conservative types are necessary for modularity, they can hinder certain program transformations based on the analysis results because they are *too* general. Holdermans and Hage show this is the case for dead code analysis [9]. Thus, it would be desirable to find the least general types possible for bindings not accessible from other modules.

Lastly, it would certainly be useful to formalize the criteria that an analysis has to fulfill in order for the higher-ranked approach taken in this thesis to be applicable. For example, a characterization of the properties of region analysis that make it unsuitable for this method. In this context, it might also be worthwhile to thoroughly compare this approach with the type system used for resource analysis by Hoffmann, Das, and Weng. While the latter seems to produce similar results, it achieves those through different means.

Bibliography

[1] Martín Abadi et al. “A core calculus of dependency”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’99*. Association for Computing Machinery (ACM), 1999.

[2] Nikolaj Skallerud Bjørner. “Minimal Typing Derivations”. In: *In ACM SIGPLAN Workshop on ML and its Applications*. 1994, pp. 120–126.

[3] Gilles Dowek. “Handbook of Automated Reasoning”. In: ed. by Alan Robinson and Andrei Voronkov. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., 2001. Chap. Higher-order Unification and Matching, pp. 1009–1062.

[4] Dirk Dussart, Fritz Henglein, and Christian Mossin. “Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time”. In: *Static Analysis*. Springer Nature, 1995, pp. 118–135.

[5] Nevin Heintze and Jon G. Riecke. “The SLam calculus”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’98*. Association for Computing Machinery (ACM), 1998.

[6] Fritz Henglein. “Type inference with polymorphic recursion”. In: *ACM Transactions on Programming Languages and Systems* 15.2 (Apr. 1993), pp. 253–289.

[7] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (Dec. 1969), p. 29.

[8] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. “Towards Automatic Resource Bound Analysis for OCaml”. In: *CoRR* abs/1611.00692 (2016).

[9] Stefan Holdermans and Jurriaan Hage. “On the rôle of minimal typing derivations in type-driven program transformation”. In: *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications - LDTA ’10*. Association for Computing Machinery (ACM), 2010.

[10] Stefan Holdermans and Jurriaan Hage. “Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP ’10*. Association for Computing Machinery (ACM), 2010.

[11] Brian T. Howard. “Inductive, coinductive, and pointed types”. In: *Proceedings of the first ACM SIGPLAN international conference on Functional programming - ICFP ’96*. Association for Computing Machinery (ACM), 1996.

[12] Steffen Jost et al. “Static determination of quantitative resource usage for higher-order programs”. In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’10*. Association for Computing Machinery (ACM), 2010.

[13] A.J. Kfoury and J. Tiuryn. “Type reconstruction in finite rank fragments of the second-order λ-calculus”. In: *Information and Computation* 98.2 (June 1992), pp. 228–257.

[14] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. “The Undecidability of the Semi-unification Problem”. In: *Information and Computation* 102.1 (Jan. 1993), pp. 83–101.

[15] Ruud Koot. “Higher-ranked Exception Types”. 2015.

[16] John McLean. “Security Models”. In: *Encyclopedia of Software Engineering*. Ed. by John J. Marciniak. Wiley Press, 1994.

[17] Dale Miller. “A logic programming language with lambda-abstraction, function variables, and simple unification”. In: *Extensions of Logic Programming*. Springer Nature, 1991, pp. 253–281.

[18] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (Dec. 1978), pp. 348–375.

[19] Alan Mycroft. “Polymorphic type schemes and recursive definitions”. In: *Lecture Notes in Computer Science*. Springer Nature, 1984, pp. 217–228.

[20] Flemming Nielson and Hanne Riis Nielson. “Type and Effect Systems”. In: *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*. London, UK, UK: Springer-Verlag, 1999, pp. 114–136.

[21] Hanne R. Nielson and Flemming Nielson. “Automatic binding time analysis for a typed λ-calculus”. In: *Science of Computer Programming* 10.2 (1988), pp. 139–176.

[22] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Jan. 1987.

[23] Frank Tip. *A Survey of Program Slicing Techniques*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1994.

[24] Mads Tofte and Jean-Pierre Talpin. “Region-Based Memory Management”. In: *Information and Computation* 132.2 (Feb. 1997), pp. 109–176.

[25] Keith Wansbrough and Simon Peyton Jones. “Once upon a polymorphic type”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’99*. Association for Computing Machinery (ACM), 1999.

[26] Guangyu Zhang. “Binding-Time Analysis: Subtyping versus Subeffecting”. MA thesis. Utrecht University, 2008.

A. Proofs

Proof of Lemma 2.11. Proof by induction on the derivation tree of $\Sigma \vdash_s \xi : \kappa$.

[S-Var] By definition of the rule, we have $\xi = \beta$ and $\Sigma(\beta) = \kappa$.

1. Let $\rho \sqsubseteq \rho_1$ be arbitrary. Then $[\![\beta]\!]_\rho = \rho(\beta) \sqsubseteq \rho_1(\beta) = [\![\beta]\!]_{\rho_1}$ by assumption.
2. Since ρ is compatible with Σ, and $\beta \in \mathrm{dom}(\Sigma)$, $[\![\beta]\!]_\rho$ is well-defined.
3. By the same argument $[\![\beta]\!]_\rho \in V_{\Sigma(\beta)}$.

[S-Abs] By definition of the rule, we have $\xi = \lambda\beta :: \kappa_1.\xi'$ and $\kappa = \kappa_1 \Rightarrow \kappa_2$. The premise is $\Sigma, \beta :: \kappa_1 \vdash_s \xi' : \kappa_2$.

1. Let $\rho \sqsubseteq \rho_1$ be arbitrary, then

$$\begin{aligned} [\![\lambda\beta :: \kappa_1.\xi']\!]_\rho =& \lambda v \in V_{\kappa_2}.\ [\![\xi']\!]_{\rho[\beta\mapsto v]} \\ \sqsubseteq& \lambda v \in V_{\kappa_2}.\ [\![\xi']\!]_{\rho_1[\beta\mapsto v]} = [\![\lambda\beta :: \kappa_1.\xi']\!]_{\rho_1} \end{aligned}$$

2. By induction, $[\![\xi']\!]_{\rho'}$ is well-defined for all ρ' compatible with $\Sigma, \beta :: \kappa_1$. Therefore, $[\![\lambda\beta :: \kappa_1.\xi']\!]_\rho$ is well-defined as well.
3. By induction, $[\![\xi']\!]_{\rho'} \in V_{\kappa_2}$ for all ρ' compatible with $\Sigma, \beta : \kappa_1$. Furthermore, since $[\![\xi']\!]_{_}$ is monotone, so is the function $\lambda v \in V_{\kappa_1}.\ [\![\xi']\!]_{\rho[\beta\mapsto v]}$. In order to see why, let $x, y \in V_{\kappa_2}$ be arbitrary such that $x \sqsubseteq y$,

$$\begin{aligned} (\lambda v \in V_{\kappa_1}.\ [\![\xi']\!]_{\rho[\beta\mapsto v]})(x) &= [\![\xi']\!]_{\rho[\beta\mapsto x]} \\ &\sqsubseteq [\![\xi']\!]_{\rho[\beta\mapsto y]} = (\lambda v \in V_{\kappa_1}.\ [\![\xi']\!]_{\rho[\beta\mapsto v]})(y). \end{aligned}$$

   Consequently, $[\![\lambda\beta :: \kappa_1.\xi']\!]_\rho \in V_{\kappa_1\Rightarrow\kappa_2}$.

[S-App] By definition of the rule, we have $\xi = \xi_1\ \xi_2$. The premises are $\Sigma \vdash_s \xi_1 : \kappa_1 \Rightarrow \kappa$ and $\Sigma \vdash_s \xi_2 : \kappa_1$.

1. Let $\rho \sqsubseteq \rho_1$ be arbitrary. By induction, $[\![\xi_1]\!]_\rho$ is a monotone function. Hence,

$$[\![\xi_1\ \xi_2]\!]_\rho = [\![\xi_1]\!]_\rho\,([\![\xi_2]\!]_\rho) \sqsubseteq [\![\xi_1]\!]_\rho\,([\![\xi_2]\!]_{\rho_1}) \sqsubseteq [\![\xi_1]\!]_{\rho_1}\,([\![\xi_2]\!]_{\rho_1}) = [\![\xi_1\ \xi_2]\!]_{\rho_1}.$$

   The first and last step hold by definition of the denotational semantics, the first inequality is due to the monotone function, the second inequality due to the pointwise extension of inequality.
2. By induction $[\![\xi_1]\!]_\rho \in V_{\kappa_1\Rightarrow\kappa}$ and $[\![\xi_2]\!]_\rho \in V_{\kappa_1}$, therefore the function application is well-defined.

3. Since by the previous argument $[\![\xi_1]\!]_\rho$ is a function from V_{κ_1} to V_κ, we know $[\![\xi_1\ \xi_2]\!]_\rho \in V_\kappa$.

[S-Join] By definition of the rule, we have $\xi = \xi_1 \sqcup \xi_2$. The premises are $\Sigma \vdash_{\mathrm{s}} \xi_1 : \kappa$ and $\Sigma \vdash_{\mathrm{s}} \xi_2 : \kappa$.

1. Let $\rho \sqsubseteq \rho_1$ be arbitrary. Then

$$\begin{aligned} [\![\xi_1 \sqcup \xi_2]\!]_\rho &= [\![\xi_1]\!]_\rho \sqcup [\![\xi_2]\!]_\rho \\ &\sqsubseteq [\![\xi_1]\!]_{\rho_1} \sqcup [\![\xi_2]\!]_{\rho_1} = [\![\xi_1 \sqcup \xi_2]\!]_{\rho_1} . \end{aligned}$$

2. By induction, $[\![\xi_1]\!]_\rho \in V_\kappa$ and $[\![\xi_2]\!]_\rho \in V_\kappa$, therefore the join operation is well-defined.
3. By the previous argument, we have $[\![\xi_1]\!]_\rho \sqcup [\![\xi_2]\!]_\rho \in V_\kappa$.

[S-Lat] By definition of the rule, we have $\xi = \ell$ and $\kappa = \star$.

1. Let $\rho \sqsubseteq \rho_1$ be arbitrary, then

$$[\![\ell]\!]_\rho = \ell \sqsubseteq \ell = [\![\ell]\!]_{\rho_1} .$$

2. This case is trivially well-defined.
3. By definition, $\ell \in V_\star$.

□

Proof of Lemma 2.19. In order to prove $\Sigma' \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$, we need to show that $[\![\xi_1]\!]_\rho \sqsubseteq [\![\xi_2]\!]_\rho$ holds for all environments ρ compatible with Σ'.

First, we prove that the assignments of the variables that are not free in a certain term have no effect on its denotation.

Let ρ' be an arbitrary environment compatible with Σ'. We define an environment ρ with domain $\mathrm{dom}(\Sigma)$ by

$$\rho(\beta) = \begin{cases} \rho'(\beta) & \text{if } \beta \in \mathrm{fav}(\xi_1) \cup \mathrm{fav}(\xi_2) \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ denotes the least element of the corresponding lattice $V_{\Sigma(\beta)}$. Clearly, $\rho \restriction_{\mathrm{fav}(\xi_1)} = \rho' \restriction_{\mathrm{fav}(\xi_1)}$ and $\rho \restriction_{\mathrm{fav}(\xi_2)} = \rho' \restriction_{\mathrm{fav}(\xi_2)}$ by definition.
Claim: ρ is compatible with Σ.
Proof: Let $\beta \in \mathrm{dom}(\Sigma)$ be an arbitrary variable. If $\beta \in \mathrm{fav}(\xi_1) \cup \mathrm{fav}(\xi_2)$, then $\rho(\beta) = \rho'(\beta) \in V_{\Sigma'(\beta)}$ by lemma 2.11. Since $\Sigma(\beta) = \Sigma'(\beta)$, $\rho(\beta) \in V_{\Sigma(\beta)}$.

Otherwise, if $\beta \notin \mathrm{fav}(\xi_1) \cup \mathrm{fav}(\xi_2)$, then we have $\rho(\beta) = \bot \in V_{\Sigma(\beta)}$ by definition. ■

Using lemma 2.18, we can derive

$$\begin{aligned} [\![\xi_1]\!]_{\rho'} &= [\![\xi_1]\!]_{\rho' \restriction_{\mathrm{fav}(\xi_1)}} = [\![\xi_1]\!]_{\rho \restriction_{\mathrm{fav}(\xi_1)}} = [\![\xi_1]\!]_\rho \\ \sqsubseteq [\![\xi_2]\!]_\rho &= [\![\xi_2]\!]_{\rho \restriction_{\mathrm{fav}(\xi_2)}} = [\![\xi_2]\!]_{\rho' \restriction_{\mathrm{fav}(\xi_2)}} = [\![\xi_2]\!]_{\rho'} \end{aligned}$$

where the inequality follows from $\Sigma \vdash_{\mathrm{sub}} \xi_1 \sqsubseteq \xi_2$ and the above claim, stating that ρ is compatible with Σ. □

Proof of Lemma 2.34. By induction on the derivation of $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$.

[T-Var] We have $\widehat{\Gamma}(x) = \widehat{\tau} \,\&\, \xi$. By definition $\lfloor\widehat{\Gamma}\rfloor(x) = \lfloor\widehat{\tau}\rfloor$ and $\lfloor x \rfloor = x$. Hence, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} x : \lfloor\widehat{\tau}\rfloor$ holds by [U-Var].

[T-Unit] Since $\lfloor\widehat{\mathrm{unit}}\rfloor = \mathrm{unit}$ and $\lfloor()\rfloor = ()$, we have $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} () : \mathrm{unit}$ by [U-Unit].

[T-Sub] There are $\widehat{\tau}'$ and ξ' such that $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t} : \widehat{\tau}' \& \xi'$, $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}' \leqslant \widehat{\tau}$ and $\Sigma \vdash_{\mathrm{sub}} \xi' \sqsubseteq \xi$. By induction, we have $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \lfloor\widehat{t}\rfloor : \lfloor\widehat{\tau}'\rfloor$ and by lemmas 2.27 and 2.29, we have $\lfloor\widehat{\tau}'\rfloor = \lfloor\widehat{\tau}\rfloor$.

[T-AnnAbs] We have $\widehat{t} = \Lambda\beta :: \kappa.\widehat{t'}$ and $\widehat{\tau} = \forall\beta :: \kappa.\widehat{\tau}'$. Moreover, $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t'} : \widehat{\tau}' \& \xi$ holds. By induction, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \lfloor\widehat{t'}\rfloor : \lfloor\widehat{\tau}'\rfloor$ holds. Furthermore, $\lfloor\widehat{\tau}\rfloor = \lfloor\forall\beta :: \kappa.\widehat{\tau}'\rfloor = \lfloor\widehat{\tau}'\rfloor$ and $\lfloor\widehat{t}\rfloor = \lfloor\Lambda\beta :: \kappa.\widehat{t'}\rfloor = \lfloor\widehat{t'}\rfloor$ hold by definition.

[T-AnnApp] We have $\widehat{t} = \widehat{t'}\ \langle\xi'\rangle$, $\widehat{\tau} = [\xi' \,/\, \beta]\widehat{\tau}'$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t'} : \forall\beta :: \kappa.\widehat{\tau}' \,\&\, \xi$. Since the substitution only affects annotations, we have $\lfloor[\xi' \,/\, \beta]\widehat{\tau}'\rfloor = \lfloor\widehat{\tau}'\rfloor$. By definition, $\lfloor\forall\beta :: \kappa.\widehat{\tau}'\rfloor = \lfloor\widehat{\tau}'\rfloor$ and $\lfloor\widehat{t'}\ \langle\xi'\rangle\rfloor = \lfloor\widehat{t'}\rfloor$. By induction, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \lfloor\widehat{t'}\rfloor : \lfloor\widehat{\tau}'\rfloor$.

The remaining cases follow analogously. □

Proof of Theorem 2.43. By induction on annotated terms t.

$t = x$ By lemma 2.38, $\{x\} = \mathrm{ftv}(x) \subseteq \emptyset$, contradiction. Thus, this case cannot happen.

$t = \lambda x : \widehat{\tau}' \,\&\, \xi'.t_1$ Clearly, $t \in \mathbf{Nf}$.

$t = t_1\ t_2$ We apply the induction hypothesis to t_1 and distinguish two cases.

If $t_1 \in \mathbf{Nf}$, then either $t_1 \in \mathbf{Nf}'$ or $t_1 = \mathrm{ann}_\ell(v')$. Suppose the former holds, then $t_1 = \lambda x : \widehat{\tau}' \,\&\, \xi'.t_1'$ as this is the only possibility such that t is well-typed. But then, we can apply [E-Abs] and get $(\lambda x : \widehat{\tau}' \,\&\, \xi'.t_1')\ t_2 \to [t_2 \,/\, x]t_1'$. In the latter case, we can apply [E-LiftApp], and get $(\mathrm{ann}_\ell(v'))\ t_2 \to \mathrm{ann}_\ell(v'\ t_2)$.

When the above is not the case, we get a reduction $t_1 \to t_1'$ and we can apply [E-Context], resulting in

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$t = (t_1, t_2)$ Clearly, $t \in \mathbf{Nf}$.

$t = \mathbf{proj}_i(t')$ We apply the induction hypothesis to t' and distinguish two cases.

If $t' \in \mathbf{Nf}$, then either $t' \in \mathbf{Nf}'$ or $t' = \mathrm{ann}_\ell(v')$. Suppose the former holds, then $t' = (t_1, t_2)$ as this is the only possibility such that t is well-typed. But then, we can apply [E-Proj] and get $\mathrm{proj}_i(t_1, t_2) \to t_i$. In the latter case, we can apply [E-LiftProj] and get $\mathrm{proj}_i(\mathrm{ann}_\ell(v')) \to \mathrm{ann}_\ell(\mathrm{proj}_i(v'))$.

When the above is not the case, we get a reduction $t' \to t''$ and we can apply [E-Context], resulting in

$$\frac{t' \to t''}{\mathrm{proj}_i(t') \to \mathrm{proj}_i(t'')}$$

$t = \textsf{\textbf{inl}}_\tau(t')$ Clearly, $t \in \mathbf{Nf}$.

$t = \textsf{\textbf{inr}}_\tau(t')$ Clearly, $t \in \mathbf{Nf}$.

$t = \textbf{case}\ t_1\ \textbf{of}\ \{\textsf{\textbf{inl}}(x) \to t_2; \textsf{\textbf{inr}}(y) \to t_3\}$ We apply the induction hypothesis to t_1 and distinguish two cases.

If $t_1 \in \mathbf{Nf}$, then either $t_1 \in \mathbf{Nf'}$ or $t_1 = \text{ann}_\ell(v')$. In the former case, we have either $t_1 = \text{inl}_{\tau_2}(t_1')$ or $t_1 = \text{inr}_{\tau_1}(t_1')$ as these are the only possibilities for t to be well-typed. But then, we either have

$$\textbf{case}\ \text{inl}_{\tau_2}(t_1')\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} \to [t_1' \ / \ x]t_2$$

by [E-CaseInl] or

$$\textbf{case}\ \text{inr}_{\tau_1}(t_1')\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} \to [t_1' \ / \ y]t_3$$

by [E-CaseInr]. In the latter case, $t_1 = \text{ann}_\ell(v')$, we have

$$\begin{aligned}&(\textbf{case}\ (\text{ann}_\ell(v'))\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\})\\ \to&\text{ann}_\ell(\ \textbf{case}\ v'\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\})\end{aligned}$$

by [E-LiftCase].

If the above does not hold, we get a reduction $t_1 \to t_1'$ and we have

$$\frac{t_1 \to t_1'}{\textbf{case}\ t_1\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} \to\ \textbf{case}\ t_1'\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\}}$$

by [E-Context].

$t = \mu x : \hat{\tau}' \ \&\ \xi'.t'$ We can apply [E-Fix] and get $\mu x : \hat{\tau}' \ \&\ \xi'.t' \to [\mu x : \hat{\tau}' \ \&\ \xi'.t' \ / \ x]t'$.

$t = \textsf{\textbf{seq}}\ t_1\ t_2$ We apply the induction hypothesis to t_1 and distinguish two cases.

If $t_1 \in \mathbf{Nf}$, then either $t_1 \in \mathbf{Nf'}$ or $t_1 = \text{ann}_\ell(v')$. In the former case, we have $\text{seq}\ t_1\ t_2 \to t_2$ by [E-Seq], in the latter case $\text{seq}\ (\text{ann}_\ell(v'))\ t_2 \to \text{ann}_\ell(\text{seq}\ v'\ t_2)$ by [E-LiftSeq].

If the above does not hold, then there is a reduction $t_1 \to t_1'$ and we have

$$\frac{t_1 \to t_1'}{\text{seq}\ t_1\ t_2 \to \text{seq}\ t_1'\ t_2}$$

by [E-Context].

$t = \textsf{\textbf{ann}}_\ell(t')$ We apply the induction hypothesis to t' and distinguish two cases.

If $t' \in \mathbf{Nf}$, then either $t' \in \mathbf{Nf'}$ or $t' = \text{ann}_{\ell'}(v')$. In the former case, $t \in \mathbf{Nf}$. In the latter case, we have $v' \in \mathbf{Nf'}$ and $\text{ann}_\ell(\text{ann}_{\ell'}(v')) \to \text{ann}_{\ell \sqcup \ell'}(v')$ by [E-AnnJoin].

If the above does not hold, then there is a reduction $t' \to t''$ and we have

$$\frac{t' \to t''}{\text{ann}_\ell(t') \to \text{ann}_\ell(t'')}$$

by [E-Context].

$t = \Lambda\beta :: \kappa.t'$ Clearly, $t \in \mathbf{Nf}$.

$t = t'\ \langle\xi\rangle$ We apply the induction hypothesis to t' and distinguish two cases.

If $t' \in \mathbf{Nf}$, then either $t' \in \mathbf{Nf}'$ or $t' = \mathrm{ann}_\ell(v')$. Suppose the former holds, then $t' = \Lambda\beta :: \kappa.t''$ as this is the only possibility such that t is well-typed. But then, we can apply [E-AnnAbs] and get $(\Lambda\beta :: \kappa.t'')\ \langle\xi\rangle \to [\xi\ /\ \beta]t''$. In the latter case, we can apply [E-LiftAnnApp] and get $(\mathrm{ann}_\ell(v'))\ \langle\xi\rangle \to \mathrm{ann}_\ell(v'\ \langle\xi\rangle)$.

When the above is not the case, we get a reduction $t' \to t''$, and we can apply [E-Context], resulting in

$$\frac{t' \to t''}{t'\ \langle\xi\rangle \to t''\ \langle\xi\rangle}$$

□

Proof of Lemma 2.44. By induction on $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$.

[T-Var] If $t = x$, we have $\widehat{\tau} = \widehat{\tau}'$ and $\xi = \xi'$ and therefore $\emptyset \mid \emptyset \vdash_{\mathrm{te}} [t'\ /\ x]x : \widehat{\tau} \,\&\, \xi$. By extending the context (see lemma 2.39), we arrive at $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t : \widehat{\tau} \,\&\, \xi$. If $t \neq x$, then $[t'\ /\ x]t = t$. As t is a different variable than x, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi$ by removing the binding for x from the context in the assumption (allowed by lemma 2.39).

[T-Unit] We have $[t'\ /\ x]t = [t'\ /\ x]() = ()$. Since t does not contain a free occurrence of x, the result can be inferred by applying lemma 2.39 to the assumption.

[T-Abs] We have $t = \lambda x_1 : \widehat{\tau}_1 \& \xi_1.t_2$ for some t_2 and $\Sigma \mid \widehat{\Gamma}, x{:}\widehat{\tau}' \& \xi', x_1{:}\widehat{\tau}_1 \& \xi_1 \vdash_{\mathrm{te}} t_2{:}\widehat{\tau}_2 \& \xi_2$ such that $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ and $\xi = \xi_2$.

If $x_1 = x$, then $[t'\ /\ x]\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_2 = \lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_2$. Moreover, we can replace the context $(\widehat{\Gamma}, x{:}\widehat{\tau}' \& \xi', x_1{:}\widehat{\tau}_1 \& \xi_1)$ in the premise with $(\widehat{\Gamma}, x_1{:}\widehat{\tau}_1 \& \xi_1)$ because the older binding of x is shadowed by the newer one. Then we get $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t : \widehat{\tau} \,\&\, \xi$ by [T-Abs].

If $x_1 \neq x$, then we can swap the bindings in the context of the premise and get $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$ by lemma 2.39. We can then apply the induction hypothesis in order to get $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\mathrm{te}} [t'\ /\ x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. Finally, we apply [T-Abs], arriving at $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.[t'\ /\ x]t_2 : \widehat{\tau} \,\&\, \xi$. As $x_1 \neq x$, $[t'\ /\ x]\lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.t_2 = \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.[t'\ /\ x]t_2$, hence the proof is complete.

[T-App] We have $t = t_1\ t_2$ and some type and effect pair $\widehat{\tau}_2 \,\&\, \xi_2$ such that $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \,\&\, \xi$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$ hold.

By induction, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \,\&\, \xi$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. Hence, we can also derive $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x](t_1\ t_2) : \widehat{\tau} \,\&\, \xi$.

[T-Pair] We have $t = (t_1, t_2)$, $\xi = \bot$ and $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle$ for some $\widehat{\tau}_1$, ξ_1, $\widehat{\tau}_2$ and ξ_2 such that $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$ hold.

By induction, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. Hence, $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t'\ /\ x](t_1, t_2) : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi$ also holds.

[T-Proj] We have $t = \text{proj}_i(t_1)$ and some $\widehat{\tau}_1, \widehat{\tau}_2, \xi_1, \xi_2$ such that $\widehat{\tau} = \widehat{\tau}_i$, $\xi = \xi_i$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_i$.

By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_i$. Using [T-Proj], we can infer $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]\text{proj}_i(t_1) : \widehat{\tau} \,\&\, \xi$.

[T-Inl] We have $t = \text{inl}_\tau(t_1)$, $\xi = \bot$ and $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\ \xi_2$ such that $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ holds and $\lfloor\widehat{\tau}_2\rfloor = \tau$.

By induction, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ and by [T-Inl] we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]\text{inl}_\tau(t_1) : \widehat{\tau} \,\&\, \xi$.

[T-Inr] Analogous to the previous case.

[T-Case] We have $t = \textbf{case}\ t_1\ \textbf{of}\ \{\text{inl}(x_1) \to t_2; \text{inl}(x_2) \to t_3\}$ and $\widehat{\tau}_1$, $\widehat{\tau}_2$, ξ_1 and ξ_2 such that $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi$, $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi', x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} t_2 : \widehat{\tau} \,\&\, \xi$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi', x_2 : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\text{te}} t_3 : \widehat{\tau} \,\&\, \xi$ hold. By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi$.

If $x = x_1 = x_2$, we have $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} t_2 : \widehat{\tau} \,\&\, \xi$ and $\Sigma \mid \widehat{\Gamma}, x_2 : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\text{te}} t_3 : \widehat{\tau} \,\&\, \xi$ by lemma 2.39. Moreover, $[t' \,/\, x]\ \textbf{case}\ t_1\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(x) \to t_3\} = \textbf{case}\ ([t' \,/\, x]t_1)\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}_x() \to t_3\}$ because x is bound in the subterms. Hence, we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$ by [T-Case].

If $x = x_1 \neq x_2$, we again have $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} t_2 : \widehat{\tau} \,\&\, \xi$, and we have $\Sigma \mid \widehat{\Gamma}, x_2 : \widehat{\tau}_2 \,\&\, \xi_2, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_3 : \widehat{\tau} \,\&\, \xi$ by lemma 2.39. By induction, we get $\Sigma \mid \widehat{\Gamma}, x_2 : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\text{te}} [t' \,/\, x]t_3 : \widehat{\tau} \,\&\, \xi$. As $[t' \,/\, x]\ \textbf{case}\ t_1\ \textbf{of}\ \{\text{inl}(x_1) \to t_2; \text{inr}(x_2) \to t_3\} = \textbf{case}\ ([t' \,/\, x]t_1)\ \textbf{of}\ \{\text{inl}(x_1) \to t_2; \text{inr}(x_2) \to [t' \,/\, x]t_3\}$ in this case, we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$ by [T-Case].

The case where $x = x_2 \neq x_1$ can be handled analogously. Similarly, if $x \neq x_1$ and $x \neq x_2$, we apply the induction hypothesis to both branches of the case expression.

[T-Ann] We have $t = \text{ann}_\ell(t_1)$, $\Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi$. By induction, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$ holds. Then, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]\text{ann}_\ell(t_1) : \widehat{\tau} \,\&\, \xi$ can be inferred by [T-Ann].

[T-Fix] We have $t = \mu x_1 : \widehat{\tau} \,\&\, \xi.t_1$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi', x_1 : \widehat{\tau} \,\&\, \xi \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi$.

If $x = x_1$, then $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau} \,\&\, \xi \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi$ because the newer binding in the context shadows the old one. Since in this case, $[t' \,/\, x]\mu x_1 : \widehat{\tau} \,\&\, \xi.t_1 = \mu x_1 : \widehat{\tau} \,\&\, \xi.t_1$, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$ simply by applying [T-Fix].

If $x \neq x_1$, then we can use lemma 2.39 to infer $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau} \,\&\, \xi, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi$. By induction, we have $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau} \,\&\, \xi \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau} \,\&\, \xi$. As $[t' \,/\, x]\mu x_1 : \widehat{\tau} \,\&\, \xi.t_1 = \mu x_1 : \widehat{\tau} \,\&\, \xi.[t' \,/\, x]t_1$, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$ by [T-Fix].

[T-Seq] We have $t = \text{seq}\ t_1\ t_2$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi$ for some $\widehat{\tau}_1$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_2 : \widehat{\tau} \,\&\, \xi$. By induction and [T-Seq], we can infer $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]\text{seq}\ t_1\ t_2 : \widehat{\tau} \,\&\, \xi$.

[T-SUB] We have $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t : \widehat{\tau}_1 \,\&\, \xi_1$ for some $\widehat{\tau}_1 \,\&\, \xi_1$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi$ hold. By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau}_1 \,\&\, \xi_1$ and by [T-SUB] we get $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t : \widehat{\tau} \,\&\, \xi$.

[T-ANNABS] We have $t = \Lambda\beta :: \kappa.t_1$ and $\widehat{\tau} = \forall\beta :: \kappa.\widehat{\tau}_1$ such that $\Sigma, \beta :: \kappa \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi$ holds.

By induction, we get $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi$ and by [T-ANNABS], $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \Lambda\beta :: \kappa.[t' \,/\, x]t_1 : \widehat{\tau} \,\&\, \xi$. Since $[t' \,/\, x]\Lambda\beta :: \kappa.t_1 = \Lambda\beta :: \kappa.[t' \,/\, x]t_1$, this completes the proof.

[T-ANNAPP] We have $t = t_1 \ \langle\xi_1\rangle$ and $\widehat{\tau} = [\xi_1 \,/\, \beta]\widehat{\tau}_1$ such that $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \,\&\, \xi$ holds.

By induction, we get $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \,\&\, \xi$. Applying [T-ANNAPP] results in $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 \ \langle\xi_1\rangle : [\xi_1 \,/\, \beta]\widehat{\tau}_1 \,\&\, \xi$. Since $[t' \,/\, x](t_1 \ \langle\xi_1\rangle) = [t' \,/\, x]t_1 \ \langle\xi\rangle$, this completes the proof.

□

Proof of Lemma 2.45. By induction on $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau} \,\&\, \xi$.

[T-VAR] We have $\widehat{t} = x$ and $\widehat{\Gamma}(x) = \widehat{\tau} \,\&\, \xi$. Then, $([\xi' \,/\, \beta]\widehat{\Gamma})(x) = [\xi' \,/\, \beta]\widehat{\tau} \,\&\, [\xi' \,/\, \beta]\xi$. Since $[\xi' \,/\, \beta]x = x$, we have $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\text{te}} x : [\xi' \,/\, \beta]\widehat{\tau} \,\&\, [\xi' \,/\, \beta]\xi$ by [T-VAR].

[T-UNIT] We have $\widehat{t} = ()$, $\widehat{\tau} = \widehat{\text{unit}}$ and $\xi = \bot$. Since $[\xi' \,/\, \beta]\widehat{\text{unit}} = \widehat{\text{unit}}$ and $[\xi' \,/\, \beta]\bot = \bot$, the result follows trivially.

[T-ABS] We have $\widehat{t} = \lambda x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t_1}$ and $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ such that $\Sigma, \beta :: \kappa \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_2 \,\&\, \xi_2$. By induction, $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma}, x : [\xi' \,/\, \beta]\widehat{\tau}_1 \,\&\, [\xi' \,/\, \beta]\xi_1 \vdash_{\text{te}} [\xi' \,/\, \beta]\widehat{t_1} : [\xi' \,/\, \beta]\widehat{\tau}_2 \,\&\, [\xi' \,/\, \beta]\xi_2$. From this, we can derive $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' \,/\, \beta]\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t_1} : [\xi' \,/\, \beta](\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle) \,\&\, [\xi' \,/\, \beta]\xi_2$ by applying [T-ABS] and moving the substitutions outwards.

[T-INL] We have $\widehat{t} = \text{inl}_{\lfloor\widehat{\tau}_2\rfloor}(\widehat{t'})$, $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ and $\xi = \bot$ for some $\widehat{t'}$, $\widehat{\tau}_1 \,\&\, \xi_1$, $\widehat{\tau}_2 \,\&\, \xi_2$ such that $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}_1 \,\&\, \xi_1$ holds.

By induction, $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' \,/\, \beta]\widehat{t'} : [\xi' \,/\, \beta]\widehat{\tau}_1 \,\&\, [\xi' \,/\, \beta]\xi_1$ holds. Using [T-INL] and moving the substitutions outwards, $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' \,/\, \beta]\text{inl}_{\lfloor\widehat{\tau}_2\rfloor}(\widehat{t'}) : [\xi' \,/\, \beta](\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle) \,\&\, \bot$ can be derived.

[T-ANN] We have $\widehat{t} = \text{ann}_\ell(\widehat{t'})$ such that $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau} \,\&\, \xi$ and $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \ell \sqsubseteq \xi$.

Applying lemma 2.17 using $[\xi'/\beta]$ for both substitutions results in $\Sigma \vdash_{\text{sub}} [\xi'/\beta]\ell \sqsubseteq [\xi' \,/\, \beta]\xi$. But $[\xi' \,/\, \beta]\ell = \ell$, hence we can omit the substitution of the left term.

By induction, we get $\Sigma \mid [\xi' \,/\, \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' \,/\, \beta]\widehat{t'} : [\xi' \,/\, \beta]\widehat{\tau} \,\&\, [\xi' \,/\, \beta]\xi$. We can derive the desired result by [T-ANN].

[T-Sub] We have $\widehat{\tau}_1$ and ξ_1 such that $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t} : \widehat{\tau}_1 \,\&\, \xi_1$, $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}$ and $\Sigma, \beta :: \kappa \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi$ hold.

By induction, $\Sigma \mid [\xi' / \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' / \beta]\widehat{t} : [\xi' / \beta]\widehat{\tau}_1 \,\&\, [\xi' / \beta]\xi_1$. Applying lemma 2.17 with $\theta_1 = \theta_2 = [\xi' / \beta]$ leads to $\Sigma \vdash_{\text{sub}} [\xi' / \beta]\xi_1 \sqsubseteq [\xi' / \beta]\xi$. By lemma 2.32, $\Sigma \vdash_{\text{sub}} [\xi' / \beta]\widehat{\tau}_1 \leqslant [\xi' / \beta]\widehat{\tau}$. Hence, $\Sigma \mid [\xi' / \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' / \beta]\widehat{t} : [\xi' / \beta]\widehat{\tau} \,\&\, [\xi' / \beta]\xi$ also holds.

[T-AnnAbs] We have $\widehat{t} = \Lambda\beta_1 :: \kappa_1.\widehat{t'}$ for some $\widehat{t'}$ and $\widehat{\tau} = \forall\beta_1 :: \kappa_1.\widehat{\tau}'$ for some $\widehat{\tau}'$ such that $\Sigma, \beta :: \kappa, \beta_1 :: \kappa_1 \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi$ and $\beta \notin \text{fav}(\widehat{\Gamma}) \cup \text{fav}(\xi)$.

We assume $\beta_1 \neq \beta$, as the substitution would not apply to $\widehat{t'}$ otherwise and the result follows trivially. By lemma 2.40, we then also have $\Sigma, \beta :: \kappa, \beta_1 :: \kappa_1 \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \,\&\, \xi$.

Applying the induction hypothesis results in $\Sigma, \beta_1 :: \kappa_1 \mid [\xi' / \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' / \beta]\widehat{t'} : [\xi' / \beta]\widehat{\tau}' \,\&\, [\xi' / \beta]\xi$.

By lemma 2.5, $\text{fav}(\xi') \subseteq \text{dom}(\Sigma)$. Therefore, the condition $\beta \notin \text{fav}([\xi' / \beta]\widehat{\Gamma}) \cup \text{fav}([\xi' / \beta]\xi)$ is still fulfilled (as no occurrences of β could have been introduced). This allows us to derive $\Sigma \mid [\xi'/\beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi'/\beta]\widehat{t} : [\xi'/\beta]\widehat{\tau} \,\&\, [\xi'/\beta]\xi$ by [T-AnnAbs].

[T-AnnApp] We have $\widehat{t} = \widehat{t_1}\ \langle\xi_1\rangle$, $\widehat{\tau} = [\xi_1 / \beta_1]\widehat{\tau}'$, $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \forall\beta_1 :: \kappa_1.\widehat{\tau}' \,\&\, \xi$ and $\Sigma, \beta :: \kappa \vdash_{\text{s}} \xi_1 : \kappa_1$. By lemma 2.7, $\Sigma \vdash_{\text{s}} [\xi' / \beta]\xi_1 : \kappa_1$.

Without loss of generality, we assume $\beta_1 \neq \beta$. Then $[\xi' / \beta]\forall\beta_1 :: \kappa_1.\widehat{\tau}' = \forall\beta_1 :: \kappa_1.[\xi' / \beta]\widehat{\tau}'$. By induction, $\Sigma \mid [\xi' / \beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi' / \beta]\widehat{t_1} : \forall\beta_1 :: \kappa_1.[\xi' / \beta]\widehat{\tau}' \,\&\, [\xi' / \beta]\xi$.

Hence, we can derive $\Sigma \mid [\xi'/\beta]\widehat{\Gamma} \vdash_{\text{te}} [\xi'/\beta]\widehat{t_1}\ \langle[\xi'/\beta]\xi_1\rangle : [\xi_1/\beta_1][\xi'/\beta']\widehat{\tau}' \,\&\, [\xi'/\beta]\xi$ by [T-AnnApp]. As $\beta_1 \neq \beta$, $[\xi_1 / \beta_1][\xi' / \beta']\widehat{\tau}' = [\xi' / \beta][\xi_1 / \beta_1]\widehat{\tau}' = [\xi' / \beta]\widehat{\tau}$, this is what we needed to show.

The remaining cases can be proven similarly. □

Proof of Theorem 2.46. By induction on $t \to t'$.

[E-Abs] We have $t = (\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_1)\ t_2$ and $t' = [t_2 / x]t_1$. By applying lemma 2.35 and discarding the cases of non-matching rules, we can assume that the derivation of $\emptyset \mid \emptyset \vdash_{\text{te}} (\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_1)\ t_2 : \widehat{\tau} \,\&\, \xi$ has the following form.

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_1 : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_2 \qquad \emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}_1 \,\&\, \xi_1}{\emptyset \mid \emptyset \vdash_{\text{te}} (\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_1)\ t_2 : \widehat{\tau}_2 \,\&\, \xi_2}\ [\text{T-App}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} (\lambda x : \widehat{\tau}_1 \,\&\, \xi_1.t_1)\ t_2 : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

Similarly, the derivation for the lambda expression can be assumed to have the

following form.

$$\dfrac{\dfrac{\emptyset \mid x : \widehat{\tau}_1' \mathbin{\&} \xi_1' \vdash_{\text{te}} t_1 : \widehat{\tau}_2' \mathbin{\&} \xi_2'}{\emptyset \mid \emptyset \vdash_{\text{te}} \lambda x : \widehat{\tau}_1' \mathbin{\&} \xi_1'.t_1 : \widehat{\tau}_1'\langle \xi_1' \rangle \to \widehat{\tau}_2'\langle \xi_2' \rangle \mathbin{\&} \xi_2'}\,[\text{T-Abs}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_1'\langle \xi_1' \rangle \to \widehat{\tau}_2'\langle \xi_2' \rangle \leqslant \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \quad \emptyset \vdash_{\text{sub}} \xi_2' \sqsubseteq \xi_2}{\emptyset \mid \emptyset \vdash_{\text{te}} \lambda x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 : \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_2}\,[\text{T-Sub}]$$

By lemma 2.30, we also must have $\emptyset \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1'$, $\emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_1'$, $\emptyset \vdash_{\text{sub}} \widehat{\tau}_2' \leqslant \widehat{\tau}_2$ and $\emptyset \vdash_{\text{sub}} \xi_2' \sqsubseteq \xi_2$. Hence, we can derive $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}_1' \mathbin{\&} \xi_1'$ using [T-Sub].

By lemma 2.44, we get $\emptyset \mid \emptyset \vdash_{\text{te}} [t_2 \,/\, x]t_1 : \widehat{\tau}_2' \mathbin{\&} \xi_2'$. Since we know $\emptyset \vdash_{\text{sub}} \widehat{\tau}_2' \leqslant \widehat{\tau}_2$ and $\emptyset \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}$ (and the same for the corresponding effect), we can derive $\emptyset \mid \emptyset \vdash_{\text{te}} [t_2 \,/\, x]t_1 : \widehat{\tau} \mathbin{\&} \xi$.

[E-AnnAbs] We have $t = (\Lambda\beta :: \kappa.t_1)\ \langle \xi' \rangle$ and $t' = [\xi' \,/\, \beta]t_1$. By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \Lambda\beta :: \kappa.t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \mathbin{\&} \xi_1 \qquad \emptyset \vdash_{\text{s}} \xi' : \kappa}{\emptyset \mid \emptyset \vdash_{\text{te}} (\Lambda\beta :: \kappa.t_1)\ \langle \xi' \rangle : [\xi' \,/\, \beta]\widehat{\tau}_1 \mathbin{\&} \xi_1}\,[\text{T-AnnApp}] \quad \emptyset \vdash_{\text{sub}} [\xi' \,/\, \beta]\widehat{\tau}_1 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} (\Lambda\beta :: \kappa.t_1)\ \langle \xi' \rangle : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

And by the same lemma,

$$\dfrac{\dfrac{\beta :: \kappa \mid \emptyset \vdash_{\text{te}} t_1 : \widehat{\tau}_2 \mathbin{\&} \xi_2 \qquad \beta \notin \text{fav}(\emptyset) \cup \text{fav}(\xi_2)}{\emptyset \mid \emptyset \vdash_{\text{te}} \Lambda\beta :: \kappa.t_1 : \forall\beta :: \kappa.\widehat{\tau}_2 \mathbin{\&} \xi_2}\,[\text{T-AnnAbs}] \quad \emptyset \vdash_{\text{sub}} \forall\beta :: \kappa.\widehat{\tau}_2 \leqslant \forall\beta :: \kappa.\widehat{\tau}_1 \quad \emptyset \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_1}{\emptyset \mid \emptyset \vdash_{\text{te}} \Lambda\beta :: \kappa.t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \mathbin{\&} \xi_1}\,[\text{T-Sub}]$$

By lemma 2.30, $\beta :: \kappa \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_1$. Since $\beta \notin \text{fav}(\xi_2)$ (and also $\beta \notin \text{fav}(\xi_1)$), we have $\beta :: \kappa \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_1$ by lemma 2.31. We can then derive $\beta :: \kappa \mid \emptyset \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \mathbin{\&} \xi_1$ by [T-Sub].

By lemma 2.45, we then have $\emptyset \mid \emptyset \vdash_{\text{te}} [\xi' / \beta]t_1 : [\xi' / \beta]\widehat{\tau}_1 \mathbin{\&} [\xi' / \beta]\xi_1$. Applying [T-Sub] and noting that β is not a free variable of ξ_1 results in $\emptyset \mid \emptyset \vdash_{\text{te}} [\xi' / \beta]t_1 : \widehat{\tau} \mathbin{\&} \xi$.

[E-Fix] We have $t = \mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1$ and $t' = [\mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 \,/\, x]t_1$. By lemma 2.35, we have

$$\dfrac{\dfrac{\emptyset \mid x : \widehat{\tau}_1 \mathbin{\&} \xi_1 \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \mathbin{\&} \xi_1}{\emptyset \mid \emptyset \vdash_{\text{te}} \mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 : \widehat{\tau}_1 \mathbin{\&} \xi_1}\,[\text{T-Fix}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

Applying lemma 2.44 results in $\emptyset \mid \emptyset \vdash_{\text{te}} [\mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 \,/\, x]t_1 : \widehat{\tau}_1 \mathbin{\&} \xi_1$. Applying subtyping allows us to infer $\emptyset \mid \emptyset \vdash_{\text{te}} [\mu x : \widehat{\tau}_1 \mathbin{\&} \xi_1.t_1 \,/\, x]t_1 : \widehat{\tau} \mathbin{\&} \xi$.

[E-Proj] We have $t = \mathrm{proj}_i(t_1, t_2)$ and $t' = t_i$. By applying lemma 2.35 twice, we get

$$\cfrac{\cfrac{\cfrac{\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1' \,\&\, \xi_1' \qquad \emptyset \mid \emptyset \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_2' \,\&\, \xi_2'}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} (t_1, t_2) : \widehat{\tau}_1'\langle\xi_1'\rangle \times \widehat{\tau}_2'\langle\xi_2'\rangle \,\&\, \bot}\,[\text{T-Pair}] \quad \begin{array}{c}\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_1'\langle\xi_1'\rangle \times \widehat{\tau}_2'\langle\xi_2'\rangle \leqslant \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \\ \emptyset \vdash_{\mathrm{sub}} \bot \sqsubseteq \xi_i\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} (t_1, t_2) : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_i}\,[\text{T-Sub}]}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{proj}_i(t_1, t_2) : \widehat{\tau}_i \,\&\, \xi_i}\,[\text{T-Proj}] \quad \begin{array}{c}\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \\ \emptyset \vdash_{\mathrm{sub}} \xi_i \sqsubseteq \xi\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{proj}_i(t_1, t_2) : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By lemma 2.30, we have $\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_i' \leqslant \widehat{\tau}_i$ and $\emptyset \vdash_{\mathrm{sub}} \xi_i' \sqsubseteq \xi_i$. Hence, we can also derive $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_i : \widehat{\tau} \,\&\, \xi$.

[E-CaseInl] We have $t = \textbf{case}\ \mathrm{inl}_\tau(t_1)\ \textbf{of}\ \{\mathrm{inl}(x) \to t_2; \mathrm{inr}(y) \to t_3\}$ and $t' = [t_1 \,/\, x]t_2$. By lemma 2.35, we have

$$\cfrac{\cfrac{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{inl}_\tau(t_1) : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi' \qquad \begin{array}{c}\emptyset \mid x : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\mathrm{te}} t_2 : \widehat{\tau}' \,\&\, \xi' \\ \emptyset \mid y : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\mathrm{te}} t_3 : \widehat{\tau}' \,\&\, \xi'\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} t : \widehat{\tau}' \,\&\, \xi'}\,[\text{T-Case}] \quad \begin{array}{c}\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \\ \emptyset \vdash_{\mathrm{sub}} \xi' \sqsubseteq \xi\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} t : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By the same lemma,

$$\cfrac{\cfrac{\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1' \,\&\, \xi_1'}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{inl}_\tau(t_1) : \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_2'\langle\xi_2'\rangle \,\&\, \bot}\,[\text{T-Inl}] \quad \begin{array}{c}\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_1'\langle\xi_1'\rangle + \widehat{\tau}_2'\langle\xi_2'\rangle \leqslant \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \\ \emptyset \vdash_{\mathrm{sub}} \bot \sqsubseteq \xi\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{inl}_\tau(t_1) : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi}\,[\text{T-Sub}]$$

By lemma 2.30, $\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_1$ and $\emptyset \vdash_{\mathrm{sub}} \xi_1' \sqsubseteq \xi_1$. Then, $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ holds by [T-Sub].

By lemma 2.44, we get $\emptyset \mid \emptyset \vdash_{\mathrm{te}} [t_1 \,/\, x]t_2 : \widehat{\tau}' \,\&\, \xi'$. Using [T-Sub], we can derive $\emptyset \mid \emptyset \vdash_{\mathrm{te}} [t_1 \,/\, x]t_2 : \widehat{\tau} \,\&\, \xi$.

[E-CaseInr] Analogous to the previous case.

[E-Seq] We have $t = \mathrm{seq}\ v'\ t'$ for any t'. By lemma 2.35, we have

$$\cfrac{\cfrac{\emptyset \mid \emptyset \vdash_{\mathrm{te}} v' : \widehat{\tau}_1 \,\&\, \xi' \qquad \emptyset \mid \emptyset \vdash_{\mathrm{te}} t' : \widehat{\tau}_2 \,\&\, \xi'}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{seq}\ v'\ t' : \widehat{\tau}_2 \,\&\, \xi'}\,[\text{T-Seq}] \quad \begin{array}{c}\emptyset \vdash_{\mathrm{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \\ \emptyset \vdash_{\mathrm{sub}} \xi' \sqsubseteq \xi\end{array}}{\emptyset \mid \emptyset \vdash_{\mathrm{te}} \mathrm{seq}\ v'\ t' : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

But then we can also derive $\emptyset \mid \emptyset \vdash_{\mathrm{te}} t' : \widehat{\tau} \,\&\, \xi$ by [T-Sub].

[E-Context] We prove this case exemplarily for $C = \text{proj}_i(\square)$, the other cases can be handled similarly. We have $t = \text{proj}_i(t_1)$ and $t' = \text{proj}_i(t_1')$ with $t_1 \to t_1'$. By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} t_1 : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_i}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(t_1) : \widehat{\tau}_i \mathbin{\&} \xi_i}\,[\text{T-Proj}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(t_1) : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

By induction, $\emptyset \mid \emptyset \vdash_{\text{te}} t_1' : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_i$ holds. Then

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} t_1' : \widehat{\tau}_1\langle \xi_1 \rangle \times \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_i}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(t_1') : \widehat{\tau}_i \mathbin{\&} \xi_i}\,[\text{T-Proj}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(t_1') : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

is also a valid derivation, completing the proof. The general idea for the remaining cases is to simply replace t_1 with t_1' in the derivation tree, as both share the same type and effect by induction.

[E-LiftApp] We have $t = \text{ann}_\ell(v')\ t_2$ and $t' = \text{ann}_\ell(v'\ t_2)$. By lemma 2.35, we have

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_2 \qquad \emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}_1 \mathbin{\&} \xi_1}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v')\ t_2 : \widehat{\tau}_2 \mathbin{\&} \xi_2}\,[\text{T-App}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v')\ t_2 : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

By lemma 2.37, we have $\emptyset \mid \emptyset \vdash_{\text{te}} v' : \widehat{\tau}_1\langle \xi_1 \rangle \to \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \xi_2$ and $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi_2$. Then we can also derive $\emptyset \mid \emptyset \vdash_{\text{te}} v'\ t_2 : \widehat{\tau} \mathbin{\&} \xi$ analogously to the above tree, and we have $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$ by transitivity. Therefore, $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v'\ t_2) : \widehat{\tau} \mathbin{\&} \xi$ holds by [T-Ann].

[E-LiftAnnApp] We have $t = \text{ann}_\ell(v')\ \langle \xi' \rangle$ and $t' = \text{ann}_\ell(v'\ \langle \xi' \rangle)$. By lemma 2.35, we have

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \forall \beta :: \kappa.\widehat{\tau}_1 \mathbin{\&} \xi_1 \qquad \emptyset \vdash_{\text{s}} \xi' : \kappa}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v')\ \langle \xi' \rangle : [\xi' \,/\, \beta]\widehat{\tau}_1 \mathbin{\&} \xi_1}\,[\text{T-AnnApp}] \quad \emptyset \vdash_{\text{sub}} [\xi' \,/\, \beta]\widehat{\tau}_1 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v')\ \langle \xi' \rangle : \widehat{\tau} \mathbin{\&} \xi}\,[\text{T-Sub}]$$

By lemma 2.37, we have $\emptyset \mid \emptyset \vdash_{\text{te}} v' : \forall \beta :: \kappa.\widehat{\tau}_1 \mathbin{\&} \xi_1$ and $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi_1$. Then we can also derive $\emptyset \mid \emptyset \vdash_{\text{te}} v'\ \langle \xi' \rangle : \widehat{\tau} \mathbin{\&} \xi$ analogously to the above tree, and we have $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$ by transitivity. Therefore, $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v'\ \langle \xi' \rangle) : \widehat{\tau} \mathbin{\&} \xi$ holds by [T-Ann].

[E-LiftProj] We have $t = \text{proj}_i(\text{ann}_\ell(v'))$ and $t' = \text{ann}_\ell(\text{proj}_i(v'))$. By lemma 2.35,

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi_i}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(\text{ann}_\ell(v')) : \widehat{\tau}_i \;\&\; \xi_i}\,[\text{T-Proj}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(\text{ann}_\ell(v')) : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

Similar to the preceding cases, we can derive $\emptyset \mid \emptyset \vdash_{\text{te}} \text{proj}_i(v'){:}\widehat{\tau}\&\xi$ and $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$ from the results of lemma 2.37. Therefore, $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(\text{proj}_i(v')) : \widehat{\tau} \;\&\; \xi$ must also hold.

[E-LiftCase] We have $t = \textbf{case}\ \text{ann}_\ell(v')\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\}$ and $t' = \text{ann}_\ell(\ \textbf{case}\ v'\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\})$. By lemma 2.35,

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi' \quad \emptyset \mid x : \widehat{\tau}_1 \;\&\; \xi_1 \vdash_{\text{te}} t_2 : \widehat{\tau}' \;\&\; \xi' \quad \emptyset \mid y : \widehat{\tau}_2 \;\&\; \xi_2 \vdash_{\text{te}} t_3 : \widehat{\tau}' \;\&\; \xi'}{\emptyset \mid \emptyset \vdash_{\text{te}} \textbf{case}\ \text{ann}_\ell(v')\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} : \widehat{\tau}' \;\&\; \xi'}\,[\text{T-Case}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi' \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \textbf{case}\ \text{ann}_\ell(v')\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

From the results of lemma 2.37 we can derive $\emptyset \mid \emptyset \vdash_{\text{te}} \textbf{case}\ v'\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\} : \widehat{\tau} \;\&\; \xi$ and $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$. Then, $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(\ \textbf{case}\ v'\ \textbf{of}\ \{\text{inl}(x) \to t_2; \text{inr}(y) \to t_3\}) : \widehat{\tau} \;\&\; \xi$ must also hold.

[E-LiftSeq] We have $t = \text{seq}\ (\text{ann}_\ell(v'))\ t_2$ and $t' = \text{ann}_\ell(\text{seq}\ v'\ t_2)$. By lemma 2.35,

$$\dfrac{\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(v') : \widehat{\tau}_1 \;\&\; \xi' \quad \emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \;\&\; \xi'}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{seq}\ (\text{ann}_\ell(v'))\ t_2 : \widehat{\tau}_2 \;\&\; \xi'}\,[\text{T-Seq}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi' \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{seq}\ (\text{ann}_\ell(v'))\ t_2 : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

Again, we infer $\emptyset \mid \emptyset \vdash_{\text{te}} \text{seq}\ v'\ t_2 : \widehat{\tau} \;\&\; \xi$ and $\emptyset \vdash_{\text{sub}} \ell \sqsubseteq \xi$ from the results of lemma 2.37. Then, $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_\ell(\text{seq}\ v'\ t_2) : \widehat{\tau} \;\&\; \xi$ must also hold.

[E-JoinAnn] We have $t = \text{ann}_{\ell_1}(\text{ann}_{\ell_2}(v'))$ and $t' = \text{ann}_{\ell_1 \sqcup \ell_2}(v')$. Applying lemma 2.37 twice results in $\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_{\ell_2}(v') : \widehat{\tau} \;\&\; \xi$ and $\emptyset \mid \emptyset \vdash_{\text{te}} v' : \widehat{\tau} \;\&\; \xi$ with $\Sigma \vdash_{\text{sub}} \ell_1 \sqsubseteq \xi$ and $\Sigma \vdash_{\text{sub}} \ell_2 \sqsubseteq \xi$. Hence, we also have $\Sigma \vdash_{\text{sub}} \ell_1 \sqcup \ell_2 \sqsubseteq \xi$ and therefore

$$\dfrac{\emptyset \mid \emptyset \vdash_{\text{te}} v' : \widehat{\tau} \;\&\; \xi \quad \emptyset \vdash_{\text{sub}} \ell_1 \sqcup \ell_2 \sqsubseteq \xi}{\emptyset \mid \emptyset \vdash_{\text{te}} \text{ann}_{\ell_1 \sqcup \ell_2}(v') : \widehat{\tau} \;\&\; \xi}\,[\text{T-Ann}]$$

□

Proof of Lemma 2.53. By induction on t.

$t = x_1$ If $x_1 = x$, then $\widehat{\tau}' = \widehat{\tau}$, $\xi' = \xi$ and clearly $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t : \widehat{\tau} \,\&\, \xi$. Otherwise, $[t' \,/\, x]x_1 = x_1$ and we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} x_1 : \widehat{\tau} \,\&\, \xi$ by assumption. Extending the context allows us to conclude $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} x_1 : \widehat{\tau} \,\&\, \xi$ by lemma 2.39.

$t = ()$ Since $[t' \,/\, x]() = ()$, we have $\widehat{\tau} = \widehat{\text{unit}}$ and therefore $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} () : \widehat{\text{unit}} \,\&\, \xi$ by [T-Unit] and [T-Sub].

$t = \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.t_2$ If $x = x_1$, then $[t' \,/\, x]t = t$ and the result follows by lemma 2.39. If $x \neq x_1$, then we have $[t' \,/\, x]t = \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.[t' \,/\, x]t_2$. By lemma 2.35, there is a derivation of the form

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} [t' \,/\, x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.[t' \,/\, x]t_2 : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot}\,[\text{T-Abs}] \quad \Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.[t' \,/\, x]t_2 : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By induction, we therefore have $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. We can swap the last two bindings in the context and infer $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.t_2 : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot$ by [T-Abs], due to $x \neq x_1$. Lastly, applying rule [T-Sub] results in $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \lambda x_1 : \widehat{\tau}_1 \,\&\, \xi_1.t_2 : \widehat{\tau} \,\&\, \xi$.

$t = t_1\ t_2$ We have $[t' \,/\, x](t_1\ \ t_2) = [t' \,/\, x]t_1\ [t' \,/\, x]t_2$, and by lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}_1\langle\xi_1\rangle \,\&\, \xi_1 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1\ [t' \,/\, x]t_2 : \widehat{\tau}_1 \,\&\, \xi_1}\,[\text{T-App}] \quad \Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1\ [t' \,/\, x]t_2 : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

for any t' with $\emptyset \mid \emptyset \vdash_{\text{te}} t' : \widehat{\tau}' \,\&\, \xi'$.

By induction, we get $\Sigma \mid \widehat{\Gamma}, x{:}\widehat{\tau}'\&\xi' \vdash_{\text{te}} t_1{:}\widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}_1\langle\xi_1\rangle\&\xi_1$ and $\Sigma \mid \widehat{\Gamma}, x{:}\widehat{\tau}'\&\xi' \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1\ t_2 : \widehat{\tau} \,\&\, \xi$ by [T-App] and [T-Sub].

$t = (t_1, t_2)$ We have $[t' \,/\, x](t_1, t_2) = ([t' \,/\, x]t_1, [t' \,/\, x]t_2)$. By lemma 2.35, we get a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} ([t' \,/\, x]t_1, [t' \,/\, x]t_2) : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot}\,[\text{T-Pair}] \quad \Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \bot \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} ([t' \,/\, x]t_1, [t' \,/\, x]t_2) : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By induction, we get $\Sigma \mid \widehat{\Gamma}, x{:}\widehat{\tau}'\&\xi' \vdash_{\text{te}} t_1{:}\widehat{\tau}_1\&\xi_1$ and $\Sigma \mid \widehat{\Gamma}, x{:}\widehat{\tau}'\&\xi' \vdash_{\text{te}} t_2{:}\widehat{\tau}_2\&\xi_2$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} (t_1, t_2) : \widehat{\tau} \,\&\, \xi$ by [T-Pair] and [T-Sub].

$t = \textbf{proj}_i(t_1)$ We have $[t' \,/\, x]\mathrm{proj}_i(t_1) = \mathrm{proj}_i([t' \,/\, x]t_1)$. By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi_i}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \mathrm{proj}_i([t' \,/\, x]t_1) : \widehat{\tau}_i \;\&\; \xi_i}\,[\text{T-Proj}] \quad \widehat{\tau}_i \vdash_{\mathrm{sub}} \widehat{\tau} \leqslant \cdot \quad \xi_i \vdash_{\mathrm{sub}} \xi \sqsubseteq \cdot}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \mathrm{proj}_i([t' \,/\, x]t_1) : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

By induction, we have $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi_i$. Using [T-Proj] and [T-Sub], we can infer $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t : \widehat{\tau} \;\&\; \xi$.

$t = \textbf{inl}_{\tau_2}(t_1)$ We have $[t'/x]\mathrm{inl}_\tau(t_1) = \mathrm{inl}_\tau([t'/x]t_1)$. By lemma 2.35, we get a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \;\&\; \xi_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \mathrm{inl}_\tau([t' \,/\, x]t_1) : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \bot}\,[\text{T-Inl}] \quad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau} \quad \Sigma \vdash_{\mathrm{sub}} \bot \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \mathrm{inl}_\tau([t' \,/\, x]t_1) : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

By induction, we get $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1 \;\&\; \xi_1$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} \mathrm{inl}_\tau(t_1) : \widehat{\tau} \;\&\; \xi$ by [T-Inl] and [T-Sub].

$t = \textbf{inr}_{\tau_1}(t_2)$ Analogous to the previous case.

$t = \textbf{case}\ t_1\ \textbf{of}\ \{\textbf{inl}(x_1) \to t_2; \textbf{inr}(x_2) \to t_3\}$ If $x \neq x_1$ and $x \neq x_2$, then we have $[t'/x]t = \textbf{case}\ [t' \,/\, x]t_1\ \textbf{of}\ \{\mathrm{inl}(x_1) \to [t' \,/\, x]t_2; \mathrm{inr}(x_2) \to [t' \,/\, x]t_3\}$. By lemma 2.35, there is a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi_3 \quad \Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \;\&\; \xi_1 \vdash_{\mathrm{te}} [t' \,/\, x]t_2 : \widehat{\tau}_3 \;\&\; \xi_3 \quad \Sigma \mid \widehat{\Gamma}, x_2 : \widehat{\tau}_2 \;\&\; \xi_2 \vdash_{\mathrm{te}} [t' \,/\, x]t_3 : \widehat{\tau}_3 \;\&\; \xi_3}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]\ \textbf{case}\ t_1\ \textbf{of}\ \{\mathrm{inl}(x_1) \to t_2; \mathrm{inr}(x_2) \to t_3\} : \widehat{\tau}_3 \;\&\; \xi_3}\,[\text{T-Case}] \quad \Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\mathrm{sub}} \xi_3 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} [t' \,/\, x]\ \textbf{case}\ t_1\ \textbf{of}\ \{\mathrm{inl}(x_1) \to t_2; \mathrm{inr}(x_2) \to t_3\} : \widehat{\tau} \;\&\; \xi}\,[\text{T-Sub}]$$

By induction, we get $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t_1 : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \xi_3$, $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_2 \;\&\; \xi_2, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_3 \;\&\; \xi_3$ and $\Sigma \mid \widehat{\Gamma}, x_2 : \widehat{\tau}_2 \;\&\; \xi_2, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} t_3 : \widehat{\tau}_3 \;\&\; \xi_3$. Using lemma 2.39, we can reorder the contexts of the latter two typing statements, resulting in $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi', x_1 : \widehat{\tau}_2 \;\&\; \xi_2 \vdash_{\mathrm{te}} t_2 : \widehat{\tau}_3 \;\&\; \xi_3$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi', x_2 : \widehat{\tau}_2 \;\&\; \xi_2 \vdash_{\mathrm{te}} t_3 : \widehat{\tau}_3 \;\&\; \xi_3$.

Now we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \;\&\; \xi' \vdash_{\mathrm{te}} \textbf{case}\ t_1\ \textbf{of}\ \{\mathrm{inl}(x_1) \to t_2; \mathrm{inr}(x_2) \to t_3\} : \widehat{\tau} \;\&\; \xi$ using [T-Case] and [T-Sub].

If $x = x_1$ or $x = x_2$, the substitution does not apply to the corresponding branches. We can then handle the corresponding cases using lemma 2.39 directly without resorting to induction, similar to what we did in the case for lambda abstractions.

$t = \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . t_1$ If $x = x_1$, then $[t' \,/\, x]t = t$ and the result follows by lemma 2.39. If $x \neq x_1$, then we have $[t' \,/\, x]t = \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . [t' \,/\, x]t_1$. By lemma 2.35, there is a derivation of the form

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1}\,[\text{T-Fix}] \quad \Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . [t' \,/\, x]t_1 : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By induction, we therefore have $\Sigma \mid \widehat{\Gamma}, x_1 : \widehat{\tau}_1 \,\&\, \xi_1, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1$. We can swap the last two bindings in the context and infer $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ by [T-Fix], for $x \neq x_1$. Lastly, applying rule [T-Sub] results in $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \mu x_1 : \widehat{\tau}_1 \,\&\, \xi_1 . t_1 : \widehat{\tau} \,\&\, \xi$.

$t = \textbf{seq}\ t_1\ t_2$ We have $[t' \,/\, x]\text{seq}\ t_1\ t_2 = \text{seq}\ ([t' \,/\, x]t_1)\ ([t' \,/\, x]t_2)$. By lemma 2.35, we get a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_2 : \widehat{\tau}_2 \,\&\, \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{seq}\ ([t' \,/\, x]t_1)\ ([t' \,/\, x]t_2) : \widehat{\tau}_2 \,\&\, \xi_2}\,[\text{T-Seq}] \quad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{seq}\ ([t' \,/\, x]t_1)\ ([t' \,/\, x]t_2) : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

By induction, we get $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau}_1 \,\&\, \xi_1$ and $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_2 : \widehat{\tau}_2 \,\&\, \xi_2$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{seq}\ t_1\ t_2 : \widehat{\tau} \,\&\, \xi$ by [T-Seq] and [T-Sub].

$t = \textbf{ann}_\ell(t_1)$ We have $[t' \,/\, x]\text{ann}_\ell(t_1) = \text{ann}_\ell([t' \,/\, x]t_1)$. By lemma 2.37, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau} \,\&\, \xi$ and $\Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi$.

By induction, we get $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \widehat{\tau} \,\&\, \xi$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{ann}_\ell(t_1) : \widehat{\tau} \,\&\, \xi$ by [T-Ann].

$t = \Lambda\beta :: \kappa . t_1$ We have $[t' \,/\, x]\Lambda\beta :: \kappa . t_1 = \Lambda\beta :: \kappa . [t' \,/\, x]t_1$. By lemma 2.35, we get a derivation

$$\dfrac{\dfrac{\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \widehat{\tau}_1 \,\&\, \xi_1 \qquad \beta \notin \text{fav}(\widehat{\Gamma}) \cup \text{fav}(\xi)}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \Lambda\beta :: \kappa . [t' \,/\, x]t_1 : \forall\beta :: \kappa . \widehat{\tau}_1 \,\&\, \xi_1}\,[\text{T-AnnApp}] \quad \Sigma \vdash_{\text{sub}} \forall\beta :: \kappa . \widehat{\tau}_1 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \Lambda\beta :: \kappa . [t' \,/\, x]t_1 : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

We assume that $\beta \notin \text{dom}(\Sigma)$, because we could simply rename the annotation variable otherwise. We can then extend the typing judgment for t' with this variable, obtaining $\Sigma, \beta :: \kappa \mid \widehat{\Gamma} \vdash_{\text{te}} t' : \widehat{\tau}' \,\&\, \xi'$ by lemma 2.40.

Applying the induction hypothesis results in $\Sigma, \beta{::}\kappa \mid \widehat{\Gamma}, x{:}\widehat{\tau}'\&\xi' \vdash_{\text{te}} t_1{:}\forall\beta{::}\kappa.\widehat{\tau}_1\&\xi_1$. The side condition is still fulfilled, because $\beta \notin \text{fav}(\widehat{\tau}') \cup \text{fav}(\xi')$, as the typing judgment $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t' : \widehat{\tau}' \,\&\, \xi'$ does not refer to β.

Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \Lambda\beta :: \kappa.t_1 : \widehat{\tau} \,\&\, \xi$ by [T-AnnAbs] and [T-Sub].

$t = t_1\ \langle\xi_1\rangle$ We have $[t' \,/\, x]t_1\ \langle\xi_1\rangle = [t' \,/\, x]t_1\ \langle\xi_2\rangle$. The ξ_2 subterm is unaffected because the substitution only applies to term variables. By lemma 2.35, we get a derivation

$$\dfrac{\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \,\&\, \xi_1 \qquad \Sigma \vdash_{\text{s}} \xi_2 : \kappa}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1\ \langle\xi_2\rangle : [\xi_2 \,/\, \beta]\widehat{\tau}_1 \,\&\, \xi_1}\ [\text{T-AnnApp}] \quad \Sigma \vdash_{\text{sub}} [\xi_2 \,/\, \beta]\widehat{\tau}_1 \leqslant \widehat{\tau} \quad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} [t' \,/\, x]t_1\ \langle\xi_2\rangle : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

By induction, we get $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1 : \forall\beta :: \kappa.\widehat{\tau}_1 \,\&\, \xi_1$. Thus, we can derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_1\ \langle\xi_2\rangle : \widehat{\tau} \,\&\, \xi$ by [T-AnnApp] and [T-Sub].

□

Proof of Lemma 2.54. By induction on $[t_1 \,/\, x]t \to t'$. The general idea in all of the cases is to use the assumption $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$ to rule out certain cases for the term t so that the same reduction rule applies regardless of the substituted subterm.

[E-Abs] We have $[t_1 \,/\, x]t = (\lambda y : \widehat{\tau}_1 \,\&\, \xi_1.t_3)\ t_4$. Without loss of generality we assume $x \neq y$. Due to the assumption $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$, we can rule out the cases where $t = x$ and $t = x\ t_4'$. Therefore, there are terms t_3' and t_3' such that $t = (\lambda y : \widehat{\tau}_1 \,\&\, \xi_1.t_3')\ t_4'$ and $[t_1 \,/\, x]t = (\lambda y : \widehat{\tau}_1 \,\&\, \xi_1.[t_1 \,/\, x]t_3')\ [t_1 \,/\, x]t_4'$.

But then we have for arbitrary t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$ that $(\lambda y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3')\ [t_2 \,/\, x]t_4' \to [[t_2 \,/\, x]t_4' \,/\, y][t_2 \,/\, x]t_3'$ holds. Because y does not occur free in t_2, we can reorder the substitutions and arrive at $(\lambda y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3')\ [t_2 \,/\, x]t_4' \to [t_2 \,/\, x][t_4' \,/\, y]t_3'$. By defining $t'' = [t_4' \,/\, y]t_3'$, we reach the desired conclusion $[t_2 \,/\, x]t \to [t_2 \,/\, x]t''$.

[E-AnnAbs] We have $[t_1 \,/\, x]t = (\Lambda\beta :: \kappa.t_3)\ \langle\xi_1\rangle$. Again, we can rule out $t = x$ and $t = x\ \langle\xi_1\rangle$. Hence, there is a t_3' such that $t = (\Lambda\beta :: \kappa.t_3')\ \langle\xi_1\rangle$.

But then $[t_2 \,/\, x]t \to [\xi_1 \,/\, \beta][t_2 \,/\, x]t_3'$ is a valid derivation by [E-AnnAbs] for all t_2 such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$. Since β cannot occur free in t_2, $[\xi_1 \,/\, \beta][t_2 \,/\, x]t_3' = [t_2 \,/\, x][\xi_1 \,/\, \beta]t_3'$. Therefore, we choose $t'' = [\xi_1 \,/\, \beta]t_3'$.

[E-Fix] We have $[t_1 \,/\, x]t = \mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3$. Again, we know $t \neq x$.

If $x = y$, then $t = \mu x : \widehat{\tau}_1 \,\&\, \xi_1.t_3$ and x does not occur free in t. But then we always have $\mu x : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \to [\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \,/\, x]t_3$ by [T-Fix]. Moreover, x also does not occur free in $[\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \,/\, x]t_3$, implying $[t_2 \,/\, x][\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \,/\, x]t_3 =$

$[\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \,/\, x]t_3$ for all t_2. Hence, choosing $t'' = [\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3 \,/\, x]t_3$ leads to the desired result.

If $x \neq y$, then $t = \mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3'$ for some t_3'. Let t_2 be arbitrary such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$. Then $[t_2 \,/\, x]t = \mu y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3'$ and we have $\mu y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3' \rightarrow [\mu y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3' \,/\, y][t_2 \,/\, x]t_3'$ by [T-Fix]. Since y does not occur free in t_2, we can reorder the substitutions as follows: $[\mu y : \widehat{\tau}_1 \,\&\, \xi_1.[t_2 \,/\, x]t_3' \,/\, y][t_2 \,/\, x]t_3' = [t_2 \,/\, x][\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3' \,/\, y]t_3'$. Thus, choosing $t'' = [\mu y : \widehat{\tau}_1 \,\&\, \xi_1.t_3' \,/\, y]t_3'$ leads to the desired result.

[E-Proj] We have $[t_1 \,/\, x]t = \text{proj}_i(t_3, t_4)$, hence t must be of the form $t = \text{proj}_i(e_1, e_2)$ for $\emptyset \vdash_{\text{sub}} \xi' \sqsubseteq \xi$ to hold. Then indeed for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$, we have $[t_2 \,/\, x]\text{proj}_i(t_3, t_4) = \text{proj}_i([t_2 \,/\, x]e_1, [t_2 \,/\, x]e_2)$ and therefore $[t_2 \,/\, x]t \rightarrow [t_2 \,/\, x]e_i$ by [E-Proj]. We choose $t'' = e_i$.

[E-CaseInl] We have $[t_1 \,/\, x]t = \textbf{case } \text{inl}_\tau(e_1) \textbf{ of } \{\text{inl}(x_1) \rightarrow e_2; \text{inr}(x_2) \rightarrow e_3\}$. As we can rule out the cases where $t = x$ and $t = \textbf{case } x \textbf{ of } \{...\}$, we must have $t = \textbf{case } \text{inl}_\tau(e_1') \textbf{ of } \{\text{inl}(x_1) \rightarrow e_2'; \text{inr}(x_2) \rightarrow e_3'\}$ for some e_1', e_2' and e_3'. We assume $x \neq x_1$ and $x \neq x_2$ as we have seen previously how to handle constructs where name shadowing occurs.

Let t_2 be any target term such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$ holds. Then, $[t_2 \,/\, x] \textbf{ case } \text{inl}_\tau(e_1') \textbf{ of } \{\text{inl}(x_1) \rightarrow e_2'; \text{inr}(x_2) \rightarrow e_3'\} = \textbf{case } \text{inl}_\tau([t_2/x]e_1') \textbf{ of } \{\text{inl}(x_1) \rightarrow [t_2 \,/\, x]e_2'; \text{inr}(x_2) \rightarrow [t_2 \,/\, x]e_3'\}$ and therefore $\textbf{case } \text{inl}_\tau([t_2 \,/\, x]e_1') \textbf{ of } \{\text{inl}(x_1) \rightarrow [t_2 \,/\, x]e_2'; \text{inr}(x_2) \rightarrow [t_2 \,/\, x]e_3'\} \rightarrow [[t_2 \,/\, x]e_1' \,/\, x_1][t_2 \,/\, x]e_2'$. We choose $t'' = [e_1' \,/\, x_1]e_2'$. We can reorder the substitutions because x_1 does not occur free in t_2, thus $[t_2/x]t'' = [[t_2/x]e_1'/x_1][t_2/x]e_2'$. Hence, we get the desired $[t_2/x]t \rightarrow [t_2/x]t''$.

[E-CaseInr] Analogous to previous case.

[E-Context] We show this case exemplarily for $C = \text{proj}_i(\square)$, but it can be proven in a similar way for other evaluation contexts. Thus, we have $[t_1 \,/\, x]t = \text{proj}_i(t_3)$ and $t' = \text{proj}_i(t_3')$ such that $t_3 \rightarrow t_3'$.

Due to the assumption $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$, we can rule out the cases where $t = x$ and $t = \text{proj}_i(x)$. This is because the typing rule [T-Proj] ensures that the effect of the argument of the projection is the same as the effect of the projection itself. Therefore, $t = \text{proj}_i(t_4)$ for some term $t_4 \neq x$, but possibly containing x. This also implies $[t_1 \,/\, x]t = \text{proj}_i([t_1 \,/\, x]t_4)$.

Then we also have $t_3 = [t_1 \,/\, x]t_4$ and thus $[t_1 \,/\, x]t_4 \rightarrow t_3'$. By lemma 2.35, there is a derivation for $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(t_4) : \widehat{\tau} \,\&\, \xi$ of the following form

$$\dfrac{\dfrac{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_4 : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_i}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(t_4) : \widehat{\tau}_i \,\&\, \xi_i}\,[\text{T-Proj}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(t_4) : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

As $\emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi$ holds, we also have $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi_i$.

Therefore, all prerequisites for the induction hypothesis are met, and we get a term t_4' such that for all terms t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \mathbin{\&} \xi'$ we have $[t_2 \,/\, x]t_4 \to [t_2 \,/\, x]t_4'$. Moreover, $\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} t_4' : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \xi_i$.

But then we also have for all such t_2

$$\frac{[t_2 \,/\, x]t_4 \to [t_2 \,/\, x]t_4'}{\text{proj}_i([t_2 \,/\, x]t_4) \to \text{proj}_i([t_2 \,/\, x]t_4')}\;[\text{E-Context}]$$

and hence by setting $t'' = \text{proj}_i(t_4')$ the desired $[t_2 \,/\, x]t \to [t_2 \,/\, x]t''$.

[E-LiftApp] We have $[t_1 \,/\, x]t = \text{ann}_\ell(v_1)\ e_1$ for some $v_1 \in \mathbf{Nf'}$. As $t = x$ or $t = x\ e_1$ would violate the assumptions, $t = \text{ann}_\ell(v_2)\ e_2$ for some v_2, e_2 and $[t_1 \,/\, x]v_2 = v_1$, $[t_1 \,/\, x]e_2 = e_1$.

Using lemma 2.35, there is a derivation for $\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ e_2 : \widehat{\tau} \mathbin{\&} \xi$ of the following form.

$$\frac{\dfrac{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2) : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \xi_2 \qquad \emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} e_2 : \widehat{\tau}_1 \mathbin{\&} \xi_1}{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ e_2 : \widehat{\tau}_2 \mathbin{\&} \xi_2}\;[\text{T-App}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ e_2 : \widehat{\tau} \mathbin{\&} \xi}\;[\text{T-Sub}]$$

Since $[t_1 \,/\, x]v_2 \in \mathbf{Nf'}$, $[t_1 \,/\, x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$.

Let t_2 be an arbitrary target term such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2{:}\widehat{\tau}'\&\xi'$ holds. By lemma 2.52, $[t_2 \,/\, x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$. Then, $[t_2 \,/\, x]v_2 \in \mathbf{Nf'}$ by definition. This means we can derive $[t_2 \,/\, x]\text{ann}_\ell(v_2)\ e_2 \to [t_2 \,/\, x]\text{ann}_\ell(v_2\ e_2)$ by [E-LiftApp]. Thus, choosing $t'' = \text{ann}_\ell(v_2\ e_2)$ leads to the desired conclusion.

[E-LiftAnnApp] We have $[t_1 \,/\, x]t = \text{ann}_\ell(v_1)\ \langle\xi_2\rangle$ for some $v_1 \in \mathbf{Nf'}$. As $t = x$ or $t = x\ \langle\xi_2\rangle$ would violate the assumptions, $t = \text{ann}_\ell(v_2)\ \langle\xi_2'\rangle$ for some v_2, ξ_2' and $[t_1 \,/\, x]v_2 = v_1$, $[t_1 \,/\, x]\xi_2' = \xi_2$.

Using lemma 2.35, there is a derivation for $\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ \langle\xi_2'\rangle : \widehat{\tau} \mathbin{\&} \xi$ of the following form.

$$\frac{\dfrac{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2) : \forall\beta\ \kappa.\widehat{\tau}_1 \mathbin{\&} \xi_1 \qquad \emptyset \vdash_{\text{s}} \xi_2 : \kappa}{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ \langle\xi_2\rangle : [\xi_2 \,/\, \beta]\widehat{\tau}_1 \mathbin{\&} \xi_1}\;[\text{T-AnnApp}] \quad \emptyset \vdash_{\text{sub}} [\xi_2 \,/\, \beta]\widehat{\tau}_1 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2)\ \langle\xi_2\rangle : \widehat{\tau} \mathbin{\&} \xi}\;[\text{T-Sub}]$$

Since $[t_1 \,/\, x]v_2 \in \mathbf{Nf'}$, $[t_1 \,/\, x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$.

Let t_2 be an arbitrary target term such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2{:}\widehat{\tau}'\&\xi'$ holds. By lemma 2.52, $[t_2 \mathbin{/} x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$. Then, $[t_2 \mathbin{/} x]v_2 \in \mathbf{Nf'}$ by definition. This means we can derive $[t_2 \mathbin{/} x]\text{ann}_\ell(v_2)\ \langle\xi_2\rangle \to [t_2 \mathbin{/} x]\text{ann}_\ell(v_2\ \langle\xi_2\rangle)$ by [E-LiftAnnApp]. Thus, choosing $t'' = \text{ann}_\ell(v_2\ \langle\xi_2\rangle)$ leads to the desired conclusion.

[E-LiftProj] We have $[t_1 \mathbin{/} x]t = \text{proj}_i(\text{ann}_\ell(v_1))$ for some $v_1 \in \mathbf{Nf'}$. As $t = x$ or $t = \text{proj}_i(x)$ would violate the assumptions, $t = \text{proj}_i(\text{ann}_\ell(v_2))$ for some v_2 and $[t_1 \mathbin{/} x]v_2 = v_1$.

Using lemma 2.35, there is a derivation for $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(\text{ann}_\ell(v_2)) : \widehat{\tau} \,\&\, \xi$ of the following form.

$$\dfrac{\dfrac{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2) : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_i}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(\text{ann}_\ell(v_2)) : \widehat{\tau}_i \,\&\, \xi_i}\,[\text{T-Proj}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_i \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{proj}_i(\text{ann}_\ell(v_2)) : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

Since $[t_1 \mathbin{/} x]v_2 \in \mathbf{Nf'}$, $[t_1 \mathbin{/} x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$.

Let t_2 be an arbitrary target term such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2{:}\widehat{\tau}'\&\xi'$ holds. By lemma 2.52, $[t_2 \mathbin{/} x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$. Then, $[t_2 \mathbin{/} x]v_2 \in \mathbf{Nf'}$ by definition. This means we can derive $[t_2 \mathbin{/} x]\text{proj}_i(\text{ann}_\ell(v_2)) \to [t_2 \mathbin{/} x]\text{ann}_\ell(\text{proj}_i(v_2))$ by [E-LiftProj]. Thus, choosing $t'' = \text{ann}_\ell(\text{proj}_i(v_2))$ leads to the desired conclusion.

[E-LiftCase] We have $[t_1 \mathbin{/} x]t = \textbf{case}\ \text{ann}_\ell(v_1)\ \textbf{of}\ \{\text{inl}(x_1) \to e_1; \text{inr}(x_2) \to e_2\}$ for some $v_1 \in \mathbf{Nf'}$. As $t = x$ or $t = \textbf{case}\ x\ \textbf{of}\ \{...\}$ would violate the assumptions, $t = \textbf{case}\ \text{ann}_\ell(v_2)\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\}$ for some v_2, e_1', e_2' and $[t_1/x]v_2 = v_1$, $[t_1 \mathbin{/} x]e_1' = e_1$, $[t_1 \mathbin{/} x]e_2' = e_2$.

Using lemma 2.35, there is a derivation of the following form.

$$\dfrac{\dfrac{\begin{array}{c}\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2) : \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \xi_3 \\ \emptyset \mid x : \widehat{\tau}' \,\&\, \xi', x_1 : \widehat{\tau}_1 \,\&\, \xi_1 \vdash_{\text{te}} e_1' : \widehat{\tau}_3 \,\&\, \xi_3 \\ \emptyset \mid x : \widehat{\tau}' \,\&\, \xi', x_2 : \widehat{\tau}_2 \,\&\, \xi_2 \vdash_{\text{te}} e_2' : \widehat{\tau}_3 \,\&\, \xi_3\end{array}}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \textbf{case}\ \text{ann}_\ell(v_2)\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\} : \widehat{\tau}_3 \,\&\, \xi_3}\,[\text{T-Case}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_3 \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \textbf{case}\ \text{ann}_\ell(v_2)\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\} : \widehat{\tau} \,\&\, \xi}\,[\text{T-Sub}]$$

Since $[t_1 \mathbin{/} x]v_2 \in \mathbf{Nf'}$, $[t_1 \mathbin{/} x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$.

Let t_2 be an arbitrary target term such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \,\&\, \xi'$ holds. By lemma 2.52, $[t_2 \mathbin{/} x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$. Then, $[t_2 \mathbin{/} x]v_2 \in \mathbf{Nf'}$ by definition. This means we can derive $[t_2 \mathbin{/} x]\ \textbf{case}\ (\text{ann}_\ell(v_2))\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\} \to [t_2 \mathbin{/} x]\text{ann}_\ell(\ \textbf{case}\ v_2\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\})$ by [E-LiftCase]. Thus, choosing $t'' = \text{ann}_\ell(\ \textbf{case}\ v_2\ \textbf{of}\ \{\text{inl}(x_1) \to e_1'; \text{inr}(x_2) \to e_2'\})$ leads to the desired conclusion.

[E-LiftSeq] We have $[t_1 / x]t = \text{seq}\ (\text{ann}_\ell(v_1))\ t_3$ for some v' and t_3. Again, we can rule out the cases $t = x$ and $t = \text{seq}\ x\ t_3$ due to $\emptyset \vdash_{\text{sub}} \xi' \not\sqsubseteq \xi$. Then, $t = \text{seq}\ (\text{ann}_\ell(v_2))\ t_4$ such that $[t_1 / x]v_2 = v_1$ and $[t_1 / x]t_4 = t_3$.

Using lemma 2.35, there is a derivation for $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{seq}\ (\text{ann}_\ell(v_2))\ t_4 : \widehat{\tau} \,\&\, \xi$ of the following form.

$$\dfrac{\dfrac{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{ann}_\ell(v_2) : \widehat{\tau}_1' \,\&\, \xi_1 \qquad \emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t_4 : \widehat{\tau}_1 \,\&\, \xi_1}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{seq}\ (\text{ann}_\ell(v_2))\ t_4 : \widehat{\tau}_1 \,\&\, \xi_1}\ [\text{T-Seq}] \quad \emptyset \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau} \quad \emptyset \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi}{\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} \text{seq}\ (\text{ann}_\ell(v_2))\ t_4 : \widehat{\tau} \,\&\, \xi}\ [\text{T-Sub}]$$

We note that $[t_1 / x]\text{ann}_\ell(v_2) = \text{ann}_\ell(v_1) \in \mathbf{Nf}$. By lemma 2.52, for all terms t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$, $[t_2 / x]\text{ann}_\ell(v_2) \in \mathbf{Nf}$ and $[t_1 / x]\text{ann}_\ell(v_2) \simeq [t_2 / x]\text{ann}_\ell(v_2)$.

Therefore, the following derivation is valid for all such t_2.

$$\dfrac{[t_2 / x]v_2 \in \mathbf{Nf}'}{\text{seq}\ (\text{ann}_\ell([t_2 / x]v_2))\ ([t_2 / x]t_4) \to \text{ann}_\ell(\text{seq}\ ([t_2 / x]v_2)\ ([t_2 / x]t_4))}\ [\text{E-LiftSeq}]$$

By setting $t'' = \text{ann}_\ell(\text{seq}\ v_2\ t_4)$ we get the desired result.

[E-JoinAnn] We have $[t_1 / x]t = \text{ann}_{\ell_1}(\text{ann}_{\ell_2}(v_1))$ with $v_1 \in \mathbf{Nf}'$. The assumptions let us rule out the cases $t = x$ and $t = \text{ann}_{\ell_1}(x)$. Hence, $t = \text{ann}_{\ell_1}(\text{ann}_{\ell_2}(v_2))$ for some v_2 such that $[t_1 / x]v_2 = v_1$. By lemma 2.37, we have $\emptyset \mid x : \widehat{\tau}' \& \xi' \vdash_{\text{te}} \text{ann}_{\ell_2}(v_2) : \widehat{\tau} \& \xi$.

Note that $\text{ann}_{\ell_2}(v_1) \in \mathbf{Nf}$. We can apply lemma 2.52, therefore $[t_2 / x]\text{ann}_{\ell_2}(v_2) \in \mathbf{Nf}$ for all t_2 with $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$. This allows us to derive $[t_2 / x]\text{ann}_{\ell_1}(\text{ann}_{\ell_2}(v_2)) \to [t_2 / x]\text{ann}_{\ell_1 \sqcup \ell_2}(v_2)$ by [E-JoinAnn] for all such t_2. Choosing $t'' = \text{ann}_{\ell_1 \sqcup \ell_2}(v_2)$ results in the desired statement.

By theorem 2.46, we have $\emptyset \mid \emptyset \vdash_{\text{te}} [t_2 / x]t'' : \widehat{\tau} \& \xi$ for all t_2 such that $\emptyset \mid \emptyset \vdash_{\text{te}} t_2 : \widehat{\tau}' \& \xi'$. By lemma 2.53, $\emptyset \mid x : \widehat{\tau}' \,\&\, \xi' \vdash_{\text{te}} t'' : \widehat{\tau} \,\&\, \xi$. □

Proof of Lemma 3.14. By induction on the derivation of $\Sigma \vdash_p \widehat{\tau} \,\&\, \xi \triangleright \Sigma'$.

[P-Unit] In the premise of the [P-Unit] rule we have $\beta \notin \overline{\alpha_i}$. Since $\text{dom}(\Sigma) = \overline{\alpha_i}$ and $\text{dom}(\Sigma') = \{\beta\}$ in this case, we have $\text{dom}(\Sigma) \cap \text{dom}(\Sigma') = \emptyset$.

Furthermore, we have $\text{fav}(\widehat{\text{unit}}) = \emptyset$ and $\text{fav}(\beta\ \overline{\alpha_i}) = \{\beta\} \cup \alpha_i = \text{dom}(\Sigma') \cup \text{dom}(\Sigma)$.

Lastly, we have $\Sigma' = \beta :: \overline{\Sigma(\alpha_i) \Rightarrow} \star$. By repeated application of [S-App], we can derive $\Sigma, \Sigma' \vdash_{\text{s}} \beta\ \overline{\alpha_i} : \star$. By [W-Unit] we have $\Sigma, \Sigma' \vdash_{\text{wft}} \widehat{\text{unit}}$.

[P-Sum] In this case, $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$. The premises of the [P-Sum] rule are $\Sigma \vdash_p \widehat{\tau}_1 \,\&\, \xi_1 \triangleright \Sigma_1'$, $\Sigma \vdash_p \widehat{\tau}_2 \,\&\, \xi_2 \triangleright \Sigma_2'$ and $\beta \notin \Sigma, \Sigma_1', \Sigma_2'$ with $\Sigma_1' = \overline{\beta_j :: \kappa_{\beta_j}}$ and $\Sigma_2' = \overline{\gamma_j :: \kappa_{\gamma_k}}$.

We can then apply the induction hypothesis in order to get

1. $\text{dom}(\Sigma) \cap \text{dom}(\Sigma_1') = \emptyset$ and $\text{dom}(\Sigma) \cap \text{dom}(\Sigma_2') = \emptyset$ and

2. $\mathrm{fav}(\widehat{\tau}_1) \cup \mathrm{fav}(\xi_1) = \mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'_1)$ and $\mathrm{fav}(\widehat{\tau}_2) \cup \mathrm{fav}(\xi_2) = \mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'_2)$, and
3. $\Sigma, \Sigma'_1 \vdash_{\mathrm{s}} \xi_1 : \star$ and $\Sigma, \Sigma'_1 \vdash_{\mathrm{wft}} \widehat{\tau}_1$ and $\Sigma, \Sigma'_2 \vdash_{\mathrm{s}} \xi_2 : \star$ and $\Sigma, \Sigma'_2 \vdash_{\mathrm{wft}} \widehat{\tau}_2$.

Since we have $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma'_1, \Sigma'_2$ and $\beta \notin \Sigma, \Sigma'_1, \Sigma'_2$ it follows that

$$\mathrm{dom}(\Sigma) \cap \mathrm{dom}(\Sigma') = \mathrm{dom}(\Sigma) \cap (\{\beta\} \cup \mathrm{dom}(\Sigma'_1) \cup \mathrm{dom}(\Sigma'_2)) = \emptyset$$

and

$$\begin{aligned} &\mathrm{fav}(\widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle) \cup \mathrm{fav}(\beta\ \overline{\alpha_i}) \\ =&\mathrm{fav}(\widehat{\tau}_1) \cup \mathrm{fav}(\xi_1) \cup \mathrm{fav}(\widehat{\tau}_2) \cup \mathrm{fav}(\xi_2) \cup \mathrm{fav}(\beta\ \overline{\alpha_i}) \\ =&\mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'_1) \cup \mathrm{dom}(\Sigma'_2) \cup \{\beta\} = \mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'). \end{aligned}$$

Similar to the previous case, we can derive $\Sigma, \Sigma' \vdash_{\mathrm{s}} \beta\ \overline{\alpha_i} : \star$. By application of [W-SUM] and result (3) of the induction hypothesis, we get $\Sigma, \Sigma'_1, \Sigma'_2 \vdash_{\mathrm{wft}} \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$. The latter derivation is possible because all parts of the combined environment have disjoint domains and therefore do not shadow existing bindings (see lemma 2.22).

[P-PROD] Analogous to the [P-SUM] case.

[P-ARR] In this case, $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$. The premises of the [P-ARR] rule are $\emptyset \vdash_p \widehat{\tau}_1 \& \xi_1 \triangleright \Sigma'_1$, $\Sigma, \Sigma'_1 \vdash_p \widehat{\tau}_1\ \&\ \xi_2 \triangleright \Sigma'_2$ and $\beta \notin \Sigma, \Sigma'_1, \Sigma'_2$ with $\Sigma'_1 = \overline{\beta_j :: \kappa_{\beta_j}}$ and $\Sigma'_2 = \overline{\gamma_j :: \kappa_{\gamma_k}}$.

We can then apply the induction hypothesis in order to get

1. $\emptyset \cap \mathrm{dom}(\Sigma'_1) = \emptyset$ and $\mathrm{dom}(\Sigma, \Sigma'_1) \cap \mathrm{dom}(\Sigma'_2) = \emptyset$,
2. $\mathrm{fav}(\widehat{\tau}_1) \cup \mathrm{fav}(\xi_1) = \emptyset \cup \mathrm{dom}(\Sigma'_1)$ and $\mathrm{fav}(\widehat{\tau}_2) \cup \mathrm{fav}(\xi_2) = \mathrm{dom}(\Sigma, \Sigma'_1) \cup \mathrm{dom}(\Sigma'_2)$, and
3. $\Sigma'_1 \vdash_{\mathrm{s}} \xi_1 : \star$ and $\Sigma'_1 \vdash_{\mathrm{wft}} \widehat{\tau}_1$ and $\Sigma, \Sigma'_1, \Sigma'_2 \vdash_{\mathrm{s}} \xi_2 : \star$ and $\Sigma, \Sigma'_1, \Sigma'_2 \vdash_{\mathrm{wft}} \widehat{\tau}_2$.

Since we have $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma'_2$ and $\beta \notin \Sigma, \Sigma'_1, \Sigma'_2$ it follows that

$$\mathrm{dom}(\Sigma) \cap \mathrm{dom}(\Sigma') = \mathrm{dom}(\Sigma) \cap (\{\beta\} \cup \mathrm{dom}(\Sigma'_2)) = \emptyset$$

and

$$\begin{aligned} &\mathrm{fav}(\forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle) \cup \mathrm{fav}(\beta\ \overline{\alpha_i}) \\ =&((\mathrm{fav}(\widehat{\tau}_1) \cup \mathrm{fav}(\xi_1) \cup \mathrm{fav}(\widehat{\tau}_2) \cup \mathrm{fav}(\xi_2)) \setminus \overline{\beta_j}) \cup \mathrm{fav}(\beta\ \overline{\alpha_i}) \\ =&((\mathrm{dom}(\Sigma'_1) \cup \mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'_2)) \setminus \mathrm{dom}(\Sigma'_1)) \cup \{\beta\} \cup \mathrm{dom}(\Sigma) \\ =&\mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'_2) \cup \{\beta\} = \mathrm{dom}(\Sigma) \cup \mathrm{dom}(\Sigma'). \end{aligned}$$

Similar to the previous case, we can derive $\Sigma, \Sigma' \vdash_{\mathrm{s}} \beta\ \overline{\alpha_i} : \star$.

We can show $\Sigma, \Sigma' \vdash_{\mathrm{wft}} \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ by repeatedly applying [W-FORALL] and then showing that $\Sigma, \Sigma', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\mathrm{wft}} \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ holds.

We apply rule [W-ARR] whose premises are given by

1. $\Sigma, \Sigma', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\mathrm{s}} \xi_1 : \star$ follows by context extension from $\Sigma'_1 \vdash_{\mathrm{s}} \xi_1 : \star$,
2. analogously, $\Sigma, \Sigma', \overline{\beta_j :: \beta_{\kappa_j}} \vdash_{\mathrm{wft}} \widehat{\tau}_1$ follows from $\Sigma'_1 \vdash_{\mathrm{wft}} \widehat{\tau}_1$
3. $\Sigma, \Sigma', \overline{\beta_j :: \beta_{\kappa_j}} \vdash_{\mathrm{s}} \xi_2 : \star$ follows from $\Sigma, \Sigma'_1, \Sigma'_2 \vdash_{\mathrm{s}} \xi_2 : \star$ by exploiting that the domains of Σ'_1 and Σ'_2 are disjoint and that β does not occur free in ξ_2 (which are both known by induction), and
4. analogously $\Sigma, \Sigma', \overline{\beta_j :: \beta_{\kappa_j}} \vdash_{\mathrm{wft}} \widehat{\tau}_1$ follows from $\Sigma, \Sigma'_1, \Sigma'_2 \vdash_{\mathrm{wft}} \widehat{\tau}_1$.

□

Proof of Lemma 3.15. Proof by induction on the derivation tree of $\Sigma \vdash_p \widehat{\tau} \,\&\, \xi \rhd \Sigma'$.

[P-Unit] We have $\widehat{\tau} = \widehat{\mathrm{unit}}$ and from the assumption $\lfloor \widehat{\tau} \rfloor = \lfloor \widehat{\tau}' \rfloor$ we can infer that the only rule that possibly applies to $\Sigma \vdash_p \widehat{\tau}' \& \xi' \rhd \Sigma'$ is also [P-Unit]. Hence, $\widehat{\tau}' = \widehat{\mathrm{unit}}$ as well.

We have $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star$ and $\Sigma'' = \beta' :: \overline{\kappa_{\alpha_i} \Rightarrow} \star$. Therefore, Σ' and Σ'' are equal up to renaming.

From the definition of [P-Unit] we know $\xi = \beta \ \overline{\alpha_i}$ and $\xi' = \beta' \ \overline{\alpha_i}$. Since by lemma 3.14, Σ' and Σ as well as Σ'' and Σ are disjoint, we have $[\Sigma' \,/\, \Sigma'']\xi' = [\beta \,/\, \beta']\xi' = \xi$ and therefore in particular $\xi \equiv_\alpha [\Sigma' \,/\, \Sigma'']\xi'$.

[P-Sum] We have $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ for some $\widehat{\tau}_1$, ξ_1, $\widehat{\tau}_2$ and ξ_2. Furthermore, we can infer from $\lfloor \widehat{\tau} \rfloor = \lfloor \widehat{\tau}' \rfloor$ that the the only rule that possibly applies to $\Sigma \vdash_p \widehat{\tau}' \& \xi' \rhd \Sigma''$ is [P-Sum]. Therefore, $\widehat{\tau}' = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}'_2\langle\xi'_2\rangle$ for some $\widehat{\tau}'_1$, ξ'_1, $\widehat{\tau}'_2$ and ξ'_2.

The premises of $\Sigma \vdash_p \widehat{\tau} \,\&\, \xi \rhd \Sigma'$ are $\Sigma \vdash_p \widehat{\tau}_1 \,\&\, \xi_1 \rhd \Sigma'_1$ and $\Sigma \vdash_p \widehat{\tau}_2 \,\&\, \xi_2 \rhd \Sigma'_2$, and the premises of $\Sigma \vdash_p \widehat{\tau}' \,\&\, \xi' \rhd \Sigma'$ are $\Sigma \vdash_p \widehat{\tau}'_1 \,\&\, \xi'_1 \rhd \Sigma''_1$ and $\Sigma \vdash_p \widehat{\tau}'_2 \,\&\, \xi'_2 \rhd \Sigma''_2$.

By induction, we know that Σ'_1 and Σ''_1 as well as Σ'_2 and Σ''_2 are equal up to renaming. Moreover, $\widehat{\tau}_1 \equiv_\alpha [\Sigma'_1 \,/\, \Sigma''_1]\widehat{\tau}'_1$, $\widehat{\tau}_2 \equiv_\alpha [\Sigma'_2 \,/\, \Sigma''_2]\widehat{\tau}'_2$, $\xi_1 \equiv_\alpha [\Sigma'_1 \,/\, \Sigma''_1]\xi'_1$ and $\xi_2 \equiv_\alpha [\Sigma'_2 \,/\, \Sigma''_2]\xi'_2$.

Suppose that $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$. Since $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma'_1, \Sigma'_2$ and $\Sigma'' = \beta' :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma''_1, \Sigma''_2$, they are equal up to renaming.

Hence, we must also have $\widehat{\tau} \equiv_\alpha [\Sigma' \,/\, \Sigma'']\widehat{\tau}'$. Analogously to the previous case, we have $\xi = [\Sigma' \,/\, \Sigma'']\xi'$.

[P-Prod] Analogous to the previous case.

[P-Arr] We have $\widehat{\tau}_1 = \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ for some $\widehat{\tau}_1$, ξ_1, $\widehat{\tau}_2$ and ξ_2. Furthermore, we can infer from $\lfloor \widehat{\tau} \rfloor = \lfloor \widehat{\tau}' \rfloor$ that the only rule that applies to $\Sigma \vdash_p \widehat{\tau}' \,\&\, \xi' \rhd \Sigma''$ is [P-Arr]. Thus, $\widehat{\tau}' = \forall \overline{\beta'_{j'} :: \kappa_{\beta'_{j'}}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}'_2\langle\xi'_2\rangle$.

The premises of $\Sigma \vdash_p \widehat{\tau} \,\&\, \xi \rhd \Sigma'$ are $\emptyset \vdash_p \widehat{\tau}_1 \,\&\, \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}}$ and $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_p \widehat{\tau}_2 \,\&\, \xi_2 \rhd \Sigma'_2$, and the premises of $\Sigma \vdash_p \widehat{\tau}' \,\&\, \xi' \rhd \Sigma'$ are $\emptyset \vdash_p \widehat{\tau}'_1 \,\&\, \xi'_1 \rhd \overline{\beta'_{j'} :: \kappa'_{\beta'_{j'}}}$ and $\Sigma, \overline{\beta'_{j'} :: \kappa'_{\beta'_{j'}}} \vdash_p \widehat{\tau}'_2 \,\&\, \xi'_2 \rhd \Sigma''_2$.

By induction, we know that $\overline{\beta_j :: \kappa_{\beta_j}}$ is equal to $\overline{\beta'_{j'} :: \kappa'_{\beta'_{j'}}}$ up to renaming. Therefore, the environments have the same size and $\kappa_{\beta_j} = \kappa'_{\beta'_j}$ for all j. Moreover, $\widehat{\tau}_1 \equiv_\alpha [\overline{\beta_j} \,/\, \overline{\beta'_j}]\widehat{\tau}'_1$ and $\xi_1 \equiv_\alpha [\overline{\beta_j} \,/\, \overline{\beta'_j}]\xi'_1$.

Hence, we can perform renaming in the second premise and get $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_p [\overline{\beta_j} \,/\, \overline{\beta'_j}]\widehat{\tau}'_2 \,\&\, [\overline{\beta_j} \,/\, \overline{\beta'_j}]\xi'_2 \rhd \Sigma''_2$.

Now we can apply induction here as well, hence Σ'_2 and Σ''_2 are equal up to renaming and $\widehat{\tau}_2 \equiv_\alpha [\Sigma'_2 \,/\, \Sigma''_2, \overline{\beta_j} \,/\, \overline{\beta'_j}]\widehat{\tau}'_2$ and $\xi_2 \equiv_\alpha [\Sigma'_2 \,/\, \Sigma''_2, \overline{\beta_j} \,/\, \overline{\beta'_j}]\xi'_2$.

Suppose that $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$. Since $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma'_2$ and $\Sigma'' = \beta' :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \Sigma''_2$, they are equal up to renaming.

Since the $\overline{\beta_j}$ are bound in $\widehat{\tau}$ and $\widehat{\tau}'$, we have indeed $\widehat{\tau} \equiv_\alpha [\Sigma' \,/\, \Sigma'']\widehat{\tau}'$ and similar to the previous cases, $\xi = [\Sigma' \,/\, \Sigma'']\xi'$.

□

Proof of Lemma 3.26. By induction on the derivation tree of $\Sigma \vdash_c \tau : \widehat{\tau} \,\&\, \xi \rhd \Sigma'$.

[C-Unit] By definition of the rule [C-Unit], $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \text{unit}$, $\widehat{\tau} = \widehat{\text{unit}}$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star$.

By definition $\theta\xi = (\lambda \overline{x_i :: \kappa_{\alpha_i}}.\bot_\star)\ \overline{\alpha_i}$. We have $\Sigma \vdash_{\text{sub}} (\lambda \overline{x_i :: \kappa_{\alpha_i}}.\bot_\star)\ \overline{\alpha_i} \sqsubseteq \bot_\star$. Hence, we can apply lemma 3.23 and get $\Sigma'' \vdash_{\text{sub}} \theta\xi \sqsubseteq \bot$.

Since $\widehat{\text{unit}}$ is the only conservative type $\widehat{\tau}'$ such that $\lfloor \widehat{\tau}' \rfloor = \text{unit}$, we also have $\Sigma'' \vdash_{\text{sub}} \theta\widehat{\tau} \leqslant \widehat{\tau}'$ by [Sub-Refl].

[C-Sum] By definition of the rule [C-Sum], $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \tau_1 + \tau_2$, $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \overline{\beta_j :: \kappa_{\beta_j}}, \overline{\gamma_k :: \kappa_{\gamma_k}}$.

The premises are $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_1 : \widehat{\tau}_1 \& \xi_1 \rhd \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_c \tau_2 : \widehat{\tau}_2 \& \xi_2 \rhd \overline{\gamma_k :: \kappa_{\gamma_k}}$.

By definition of $\lfloor \cdot \rfloor$ and the fact that $\widehat{\tau}'$ is conservative, we know that $\widehat{\tau}' = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}'_2\langle\xi'_2\rangle$ for some $\widehat{\tau}'_1, \widehat{\tau}'_2, \xi'_1, \xi'_2$.

Note that the substitutions occurring in the induction hypothesis are simply restrictions of the substitution θ.

By induction, we have $\Sigma'' \vdash_{\text{sub}} \theta\widehat{\tau}_1 \leqslant \widehat{\tau}'_1$ and $\Sigma'' \vdash_{\text{sub}} \theta\xi_1 \sqsubseteq \bot$ as well as $\Sigma'' \vdash_{\text{sub}} \theta\widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma'' \vdash_{\text{sub}} \theta\xi_2 \sqsubseteq \bot$.

From the transitivity of the subsumption relation on dependency terms we can deduce $\Sigma'' \vdash_{\text{sub}} \theta\xi_1 \sqsubseteq \xi'_1$ and $\Sigma'' \vdash_{\text{sub}} \theta\xi_2 \sqsubseteq \xi'_2$, as $\bot$ is the least element of this order.

Subtyping rule [S-Sum] lets us conclude $\Sigma'' \vdash_{\text{sub}} \theta\widehat{\tau} \leqslant \widehat{\tau}'$.

Analogously to the first case, we can also prove $\Sigma'' \vdash_{\text{sub}} \theta\xi \sqsubseteq \bot$.

[C-Prod] Analogous to [C-Sum].

[C-Arr] In this case we have $\Sigma = \overline{\alpha_i :: \kappa_{\alpha_i}}$, $\tau = \tau_1 \to \tau_2$, $\widehat{\tau} = \forall \overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$, $\xi = \beta\ \overline{\alpha_i}$ and $\Sigma' = \beta :: \overline{\kappa_{\alpha_i} \Rrightarrow} \star, \overline{\gamma_k :: \kappa_{\gamma_k}}$.

The premises are $\emptyset \vdash_c \tau_1 : \widehat{\tau}_1 \& \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_c \tau_2 : \widehat{\tau}_2 \& \xi_2 \triangleright \overline{\gamma_k :: \kappa_{\gamma_k}}$.

By definition of $\lfloor\cdot\rfloor$ and the fact that $\widehat{\tau}'$ is conservative, we know that $\widehat{\tau}' = \forall \overline{\beta'_j :: \kappa_{\beta'_j}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}'_2\langle\xi'_2\rangle$ for some $\widehat{\tau}'_1, \widehat{\tau}'_2, \xi'_1, \xi'_2$. Since the number and sorts of the variables that are quantified over are determined by the definition of conservative types and the shape of $\widehat{\tau}'_1$, we can assume without loss of generality that $\beta_j = \beta'_j$ for all j.

Due to $\widehat{\tau}'$ being conservative, we also know $\emptyset \vdash_p \widehat{\tau}'_1 \ \&\ \xi'_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$. By lemma 3.20 applied to the first premise we have $\emptyset \vdash_p \widehat{\tau}_1 \ \&\ \xi_1 \triangleright \overline{\beta_j :: \kappa_{\beta_j}}$. Since pattern types are unique up to alpha-equivalence (see lemma 3.15), we can again assume without loss of generality that $\widehat{\tau}'_1 = \widehat{\tau}_1$ and $\xi'_1 = \xi_1$. Then we have $\overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}_1$ by [Sub-Refl] and $\overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi_1$ from the reflexivity of the subsumption relation. We can extend the contexts with arbitrary variables not occurring free in $\widehat{\tau}_1$ and ξ_1 (using lemmas 2.19 and 2.31), therefore $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}_1$ and $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi_1$ also hold.

By induction, we have $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \theta\widehat{\tau}_2 \leqslant \widehat{\tau}'_2$ and $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \theta\xi_2 \sqsubseteq \bot$.

From the transitivity of the subsumption relation on dependency terms we can deduce $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \theta\xi_2 \sqsubseteq \xi'_2$, as $\bot$ is the least element of this order.

Subtyping rule [S-Arr] lets us conclude $\Sigma'', \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \to \theta\widehat{\tau}_2\langle\theta\xi_2\rangle \leqslant \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}'_2\langle\xi'_2\rangle$. The substitution θ only applies to the return type, because all of the free variables of $\widehat{\tau}_1$ and ξ_1 are in bound in $\widehat{\tau}$ (see also lemma 3.14).

Repeated application of [Sub-Forall] gives us the desired $\Sigma'' \vdash_{\text{sub}} \theta\widehat{\tau} \leqslant \widehat{\tau}'$.

Analogously to the previous cases, we can also prove $\Sigma'' \vdash_{\text{sub}} \xi \sqsubseteq \bot$.

□

Proof of Lemma 3.34. By induction on the derivation of $\Sigma \vdash_p \widehat{\tau}' \ \&\ \xi' \triangleright \overline{\beta_i :: \kappa_{\beta_i}}$.

[P-Unit] We have $\widehat{\tau}' = \widehat{\text{unit}}$ and $\overline{\beta_j :: \kappa_{\beta_j}} = \beta :: \overline{\kappa_{\alpha_i} \Rrightarrow} \star$. Since $\widehat{\tau}$ is conservative and $\lfloor\widehat{\tau}\rfloor = \lfloor\widehat{\tau}'\rfloor$, we also have $\widehat{\tau} = \widehat{\text{unit}}$. Therefore, $\theta = \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \widehat{\tau}'; \widehat{\tau}) = [\,]$. Clearly, $\theta\widehat{\tau}' = \theta\widehat{\text{unit}} = \widehat{\text{unit}} = \widehat{\tau}$ and $\emptyset = \text{dom}(\theta) = \{\beta\} \setminus \{\beta\}$.

[P-Sum] We have $\widehat{\tau}' = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}'_2\langle\xi'_2\rangle$ for some $\widehat{\tau}'_1$, ξ'_1, $\widehat{\tau}'_2$ and ξ'_2. The premises of the rule are $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}'_1 \ \&\ \xi'_1 \triangleright \overline{\gamma_k :: \kappa_{\gamma_k}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_p \widehat{\tau}'_2 \ \&\ \xi'_2 \triangleright \overline{\gamma'_{k'} :: \kappa_{\gamma'_{k'}}}$ with $\overline{\beta_j :: \kappa_{\beta_j}} = \beta :: \overline{\kappa_{\alpha_i} \Rrightarrow} \star, \overline{\gamma_k :: \kappa_{\gamma_k}}, \overline{\gamma'_{k'} :: \kappa_{\gamma'_{k'}}}$. By the definition of the pattern rules, $\xi'_1 = \beta'\ \overline{\alpha_i}$ and $\xi'_2 = \beta''\ \overline{\alpha_i}$ for some $\beta' :: \overline{\kappa_{\alpha_i} \Rrightarrow} \star$ and $\beta'' :: \overline{\kappa_{\alpha_i} \Rrightarrow} \star$.

Because $\lfloor\widehat{\tau}\rfloor = \lfloor\widehat{\tau}'\rfloor$ and $\widehat{\tau}$ is conservative, $\widehat{\tau} = \widehat{\tau}_1\langle\xi_1\rangle + \widehat{\tau}_2\langle\xi_2\rangle$ for some $\widehat{\tau}_1$, ξ_1, $\widehat{\tau}_2$ and ξ_2. By definition, both $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative as well. From $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\text{wft}} \widehat{\tau}$ we can infer $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\text{wft}} \widehat{\tau}_1$ and $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\text{wft}} \widehat{\tau}_2$ as they are the premises of [W-Sum] which is the only rule that applies.

By induction, we get $\theta' = \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \hat{\tau}'_1; \hat{\tau}_1)$ and $\theta'' = \mathcal{M}(\alpha_i :: \kappa_{\alpha_i}; \hat{\tau}'_2; \hat{\tau}_2)$. It follows

$$\theta = \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \hat{\tau}'; \hat{\tau}) = [\beta' \mapsto \lambda\overline{\alpha_i :: \kappa_{\alpha_i}}.\xi_1, \beta'' \mapsto \lambda\overline{\alpha_i :: \kappa_{\alpha_i}}.\xi_2] \circ \theta' \circ \theta''.$$

We have $\mathrm{dom}(\theta) = \{\beta', \beta''\} \cup \mathrm{dom}(\theta') \cup \mathrm{dom}(\theta'') = \{\beta', \beta''\} \cup (\overline{\gamma_k} \setminus \{\beta'\}) \cup (\overline{\gamma'_{k'}} \setminus \{\beta''\}) = \overline{\beta_j} \setminus \{\beta\}$. Moreover, $\theta(\hat{\tau}'_1\langle\xi'_1\rangle + \hat{\tau}'_2\langle\xi'_2\rangle) = \theta\hat{\tau}'_1\langle\theta\xi'_1\rangle + \theta\hat{\tau}'_2\langle\theta\xi'_2\rangle = \theta'\hat{\tau}'_1\langle\xi_1\rangle + \theta''\hat{\tau}'_2\langle\xi_2\rangle = \hat{\tau}_1\langle\xi_1\rangle + \hat{\tau}_2\langle\xi_2\rangle = \hat{\tau}$ where the second step is justified by lemma 3.14.

Lastly, we need to show $\Sigma \vdash_{\mathrm{s}} \theta\beta' : \overline{\kappa_{\alpha_i} \Rightarrow} \star$ and $\Sigma \vdash_{\mathrm{s}} \theta\beta'' : \overline{\kappa_{\alpha_i} \Rightarrow} \star$. These statements can be derived by repeated application of [S-Abs].

$$\frac{\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\mathrm{s}} \xi_1 : \star}{\Sigma \vdash_{\mathrm{s}} \lambda\overline{\alpha_i :: \kappa_{\alpha_i}}.\xi_1 : \overline{\kappa_{\alpha_i} \Rightarrow} \star}\ \overline{[\text{S-Abs}]}_i$$

The assumption follows as a premise of $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\mathrm{wft}} \hat{\tau}$. This holds analogously for $\theta\beta''$ and the remaining judgments follow by induction. Hence, $\Sigma \vdash_{\mathrm{s}} \theta\beta_j : \kappa_{\beta_j}$ for all $\beta_j \in \mathrm{dom}(\theta)$.

[P-Prod] Analogous to previous case.

[P-Arr] We have $\hat{\tau}' = \forall\overline{\beta'_k :: \kappa_{\beta'_k}}.\hat{\tau}'_1\langle\xi'_1\rangle \to \hat{\tau}'_2\langle\xi'_2\rangle$ for some $\hat{\tau}'_1$, ξ'_1, $\hat{\tau}'_2$ and ξ'_2. The premises of the rule are $\emptyset \vdash_p \hat{\tau}'_1 \,\&\, \xi'_1 \triangleright \overline{\beta'_k :: \kappa_{\beta'_k}}$ and $\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}} \vdash_p \hat{\tau}'_2 \,\&\, \xi'_2 \triangleright \overline{\gamma_l :: \kappa_{\gamma_l}}$ where $\overline{\beta_j :: \kappa_{\beta_k}} = \beta :: \overline{\kappa_{\alpha_i} \Rightarrow} \star, \overline{\gamma_l :: \kappa_{\gamma_l}}$.

Since we have $\lfloor\hat{\tau}\rfloor = \lfloor\hat{\tau}'\rfloor$ and $\hat{\tau}$ is conservative, $\hat{\tau} = \forall\overline{\beta''_l :: \kappa_{\beta''_l}}.\hat{\tau}_1\langle\xi_1\rangle \to \hat{\tau}_2\langle\xi_2\rangle$ for some $\hat{\tau}_1$, $\hat{\tau}_2$, ξ_1 and ξ_2 with $\emptyset \vdash_p \hat{\tau}_1 \,\&\, \xi_1 \triangleright \overline{\beta''_l :: \kappa_{\beta''_l}}$. Because pattern types are unique up to renaming (see lemma 3.15), we can assume without loss of generality that $\overline{\beta''_l :: \kappa_{\beta''_l}} = \overline{\beta'_k :: \kappa_{\beta'_k}}$ and therefore $\hat{\tau}_1 = \hat{\tau}'_1$ and $\xi_1 = \xi'_1$. By the definition of the pattern rules, $\xi_2 = \gamma'\ \overline{\alpha_i}\ \overline{\beta'_k}$ for some $\gamma' :: \overline{\kappa_{\alpha_i} \Rightarrow}\ \overline{\kappa_{\beta'_k} \Rightarrow} \star$. By assumption, $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}} \vdash_{\mathrm{wft}} \hat{\tau}$. But then we must also have $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}} \vdash_{\mathrm{wft}} \hat{\tau}_2$.

We can now apply the induction hypothesis. We get a substitution

$$\theta' = \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}; \hat{\tau}'_2; \hat{\tau}_2)$$

such that $\theta'\hat{\tau}'_2 = \hat{\tau}_2$ and $\Sigma \vdash_{\mathrm{s}} \theta'\gamma_l : \kappa_{\gamma_l}$ for all $\gamma_l \in \mathrm{dom}(\theta') = \overline{\gamma_l} \setminus \{\gamma'\}$. Unfolding the definition of $\mathcal{M}$ yields

$$\begin{aligned}
\theta =& \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \hat{\tau}'; \hat{\tau}) \\
=& \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}; \forall\overline{\beta'_k :: \kappa_{\beta'_k}}.\hat{\tau}'_1\langle\xi'_1\rangle \to \hat{\tau}'_2\langle\xi'_2\rangle; \forall\overline{\beta'_k :: \kappa_{\beta'_k}}.\hat{\tau}'_1\langle\xi'_1\rangle \to \hat{\tau}_2\langle\xi_2\rangle) \\
&\vdots \\
=& \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}; \hat{\tau}'_1\langle\xi'_1\rangle \to \hat{\tau}'_2\langle\xi'_2\rangle; \hat{\tau}'_1\langle\xi'_1\rangle \to \hat{\tau}_2\langle\xi_2\rangle) \\
=& [\gamma' \mapsto \lambda\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}.\xi_2] \circ \mathcal{M}(\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}; \hat{\tau}'_2; \hat{\tau}_2) \\
=& [\gamma' \mapsto \lambda\overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}.\xi_2] \circ \theta'.
\end{aligned}$$

Clearly, $\text{dom}(\theta) = \overline{\gamma_l} = \overline{\beta_j} \setminus \{\beta\}$. From the well-formedness of $\widehat{\tau}$ we can infer $\Sigma, \overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}} \vdash_{\text{s}} \xi_2 : \star$. Therefore, $\Sigma \vdash_{\text{s}} \lambda \overline{\alpha_i :: \kappa_{\alpha_i}}, \overline{\beta'_k :: \kappa_{\beta'_k}}.\xi_2 : \overline{\kappa_{\alpha_i} \Rightarrow} \overline{\kappa_{\beta'_k} \Rightarrow} \star$ by repeated application of [S-ABS]. Also, $\theta\widehat{\tau}' = \forall \overline{\beta'_k :: \kappa_{\beta'_k}}.\theta\widehat{\tau}'_1\langle\theta\xi'_1\rangle \to \theta\widehat{\tau}'_2\langle\theta\xi'_2\rangle = \forall \overline{\beta'_k :: \kappa_{\beta'_k}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to \theta\widehat{\tau}'_2\langle\theta\xi'_2\rangle = \forall \overline{\beta'_k :: \kappa_{\beta'_k}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle = \widehat{\tau}$. The second step is justified by the fact that the free variables of $\widehat{\tau}'_1$ and ξ'_1 are disjoint from $\text{dom}(\theta)$ (see lemma 3.14). The third step follows from the induction hypothesis.

□

Proof of Lemma 3.35. By induction on τ (i.e. the underlying structure of both $\widehat{\tau}_1$ and $\widehat{\tau}_2$).

$\tau = \textbf{unit}$ Then $\widehat{\tau}_1 = \widehat{\tau}_2 = \widehat{\text{unit}}$ because $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are also conservative. By definition, $\widehat{\text{unit}} \sqcup \widehat{\text{unit}} = \widehat{\text{unit}}$. Clearly, $\lfloor\widehat{\text{unit}}\rfloor = \text{unit}$ by definition, $\Sigma \vdash_{\text{sub}} \widehat{\text{unit}} \leqslant \widehat{\text{unit}}$ by [SUB-REFL], $\Sigma \vdash_{\text{wft}} \widehat{\text{unit}}$ by [W-UNIT] and $\widehat{\text{unit}}$ conservative by definition.

$\tau = \tau' + \tau''$ Then $\widehat{\tau}_1 = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}''_1\langle\xi''_1\rangle$ and $\widehat{\tau}_2 = \widehat{\tau}'_2\langle\xi'_2\rangle + \widehat{\tau}''_2\langle\xi''_2\rangle$ such that $\lfloor\widehat{\tau}'_1\rfloor = \tau' = \lfloor\widehat{\tau}'_2\rfloor$ and $\lfloor\widehat{\tau}''_1\rfloor = \tau'' = \lfloor\widehat{\tau}''_2\rfloor$, because $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative. Moreover, we know $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$, and the only rule applying here is [W-SUM]. Therefore, $\Sigma \vdash_{\text{wft}} \widehat{\tau}'_1$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}''_1$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}''_2$.

By induction, we get $\lfloor\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2\rfloor = \tau'$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_1 \sqcup \widehat{\tau}'_2$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_2 \leqslant \widehat{\tau}'_1 \sqcup \widehat{\tau}'_2$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}'_1 \sqcup \widehat{\tau}'_2$ and $\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2$ conservative. Analogously, $\lfloor\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2\rfloor = \tau''$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$, $\Sigma \vdash_{\text{sub}} \widehat{\tau}''_2 \leqslant \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$ and $\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$ conservative.

We have $\widehat{\tau}_1 \sqcup \widehat{\tau}_2 = (\widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}''_1\langle\xi''_1\rangle) \sqcup (\widehat{\tau}'_2\langle\xi'_2\rangle + \widehat{\tau}''_2\langle\xi''_2\rangle) = (\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$. Clearly, $\lfloor(\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle\rfloor = \lfloor\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2\rfloor + \lfloor\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2\rfloor = \tau' + \tau'' = \tau$.

We can derive

$$\frac{\begin{array}{cc} \Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_1 \sqcup \widehat{\tau}'_2 & \Sigma \vdash_{\text{sub}} \widehat{\tau}''_2 \leqslant \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2 \\ \Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi'_1 \sqcup \xi'_2 & \Sigma \vdash_{\text{sub}} \xi''_1 \sqsubseteq \xi''_1 \sqcup \xi''_2 \end{array}}{\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}''_1\langle\xi''_1\rangle \leqslant (\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle} \text{[SUB-SUM]}$$

and $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_2\langle\xi'_2\rangle + \widehat{\tau}''_2\langle\xi''_2\rangle \leqslant (\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$ follows analogously. The resulting type $(\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$ is conservative, as its parts are conservative by induction.

Lastly,

$$\frac{\begin{array}{cc} \Sigma \vdash_{\text{wft}} \widehat{\tau}'_1 \sqcup \widehat{\tau}'_2 & \Sigma \vdash_{\text{s}} \xi'_1 \sqcup \xi'_2 : \star \\ \Sigma \vdash_{\text{wft}} \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2 & \Sigma \vdash_{\text{s}} \xi''_1 \sqcup \xi''_2 : \star \end{array}}{\Sigma \vdash_{\text{wft}} (\widehat{\tau}'_1 \sqcup \widehat{\tau}'_2)\langle\xi'_1 \sqcup \xi'_2\rangle + (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle}$$

where the well-sortedness of the annotations follows from the well-formedness of $\widehat{\tau}_1$ and $\widehat{\tau}_2$ and [S-JOIN].

$\tau = \tau' \times \tau''$ Analogous to the previous case.

$\tau = \tau' \to \tau''$ Then $\widehat{\tau}_1 = \forall \overline{\beta_i :: \kappa_{\beta_i}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}''_1\langle\xi''_1\rangle$ and $\widehat{\tau}_2 = \forall \overline{\beta'_j :: \kappa_{\beta'_j}}.\widehat{\tau}'_2\langle\xi'_2\rangle \to \widehat{\tau}''_2\langle\xi''_2\rangle$ since both are conservative. Since $\lfloor\widehat{\tau}_1\rfloor = \tau = \lfloor\widehat{\tau}_2\rfloor$, we also know $\lfloor\widehat{\tau}'_1\rfloor = \tau' = \lfloor\widehat{\tau}'_2\rfloor$ and $\lfloor\widehat{\tau}''_1\rfloor = \tau'' = \lfloor\widehat{\tau}''_2\rfloor$.

By definition of conservativeness, $\emptyset \vdash_p \widehat{\tau}'_1 \& \xi'_1 \triangleright \overline{\beta_i :: \kappa_{\beta_i}}$ and $\emptyset \vdash_p \widehat{\tau}'_2 \& \xi'_2 \triangleright \overline{\beta'_j :: \kappa_{\beta'_j}}$. By lemma 3.15, patterns are unique up to renaming. Hence, we can assume without loss of generality that $\overline{\beta_i :: \kappa_{\beta_i}} = \overline{\beta'_j :: \kappa_{\beta'_j}}$, and therefore $\widehat{\tau}'_1 = \widehat{\tau}'_2$ and $\xi'_1 = \xi'_2$.

Then, $\widehat{\tau}_1 \sqcup \widehat{\tau}_2 = \forall \overline{\beta_i :: \kappa_{\beta_i}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$.

From $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$ we can infer $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{wft}} \widehat{\tau}''_1$ and $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{wft}} \widehat{\tau}''_2$.

The induction hypothesis yields $\lfloor\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2\rfloor = \tau''$, $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$, $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{sub}} \widehat{\tau}''_2 \leqslant \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$, $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{wft}} \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$ and the conservativeness of $\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$.

Clearly, $\lfloor\widehat{\tau}_1 \sqcup \widehat{\tau}_2\rfloor = \lfloor\forall \overline{\beta_i :: \kappa_{\beta_i}}.\widehat{\tau}'_1\langle\xi'_1\rangle \to (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle\rfloor = \lfloor\widehat{\tau}'_1\rfloor \to \lfloor\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2\rfloor = \tau' \to \tau''$.

Using [SUB-ARR], we can derive $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{sub}} \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}''_1\langle\xi''_1\rangle \leqslant \widehat{\tau}'_1\langle\xi'_1\rangle \to (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$ and $\Sigma, \overline{\beta_i :: \kappa_{\beta_i}} \vdash_{\text{sub}} \widehat{\tau}'_1\langle\xi'_1\rangle \to \widehat{\tau}''_2\langle\xi''_2\rangle \leqslant \widehat{\tau}'_1\langle\xi'_1\rangle \to (\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2)\langle\xi''_1 \sqcup \xi''_2\rangle$. The contravariant position can be treated invariantly, because we could assume $\widehat{\tau}'_1 = \widehat{\tau}'_2$.

Repeated application of [SUB-FORALL] results in the desired $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$.

As $\widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$ is conservative and $\emptyset \vdash_p \widehat{\tau}'_1 \ \&\ \xi'_1 \triangleright \overline{\beta_i :: \kappa_{\beta_i}}$, $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ is also conservative.

Similarly, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1 \sqcup \widehat{\tau}_2$ follows from $\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}'_1$ and $\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}''_1 \sqcup \widehat{\tau}''_2$.

□

Proof of Theorem 3.36. By induction on t. All of the following cases assume an implicit application of the foregoing lemmas 3.32 and 3.33, as we only show that the types and effects before canonicalization are derivable in the annotated type system.

$t = x$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; x) = x : \widehat{\Gamma}(x)$. Suppose $\widehat{\Gamma}(x) = \widehat{\tau} \ \&\ \xi$. By assumption, $\widehat{\Gamma}$ is well-formed under Σ, therefore $\widehat{\tau}$ is conservative, $\Sigma \vdash_{\text{wft}} \widehat{\tau}$ and $\Sigma \vdash_{\text{s}} \xi : \star$.

$t = ()$ $\mathcal{R}(\widehat{\Gamma}; \Sigma; ()) = () : \widehat{\text{unit}} : \bot$. Trivially, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} : \widehat{\text{unit}} \ \&\ \bot$ by [T-UNIT], $\Sigma \vdash_{\text{wft}} \widehat{\text{unit}}$ by [W-UNIT], $\Sigma \vdash_{\text{s}} \bot : \star$ by [S-LAT] and $\widehat{\text{unit}}$ is conservative by definition.

$t = \textbf{ann}_\ell(t')$ In this case $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{ann}_\ell(t')) = \text{ann}_\ell(\widehat{t'}) : \widehat{\tau}' \ \&\ [\![\xi' \sqcup \ell]\!]_\Sigma$ where $\widehat{t'} : \widehat{\tau}' \ \&\ \xi' = \mathcal{R}(\widehat{\Gamma}; \Sigma; t')$.

By induction, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \ \&\ \xi'$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}'$, $\Sigma \vdash_{\text{s}} \xi' : \star$ and $\widehat{\tau}'$ is conservative. We can then derive

$$\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}' \ \&\ \xi' \qquad \dfrac{}{\Sigma \vdash_{\text{sub}} \widehat{\tau}' \leqslant \widehat{\tau}'}\text{[SUB-REFL]} \qquad \Sigma \vdash_{\text{sub}} \xi' \sqsubseteq \xi' \sqcup \ell}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau}' \ \&\ \xi' \sqcup \ell}\text{[T-SUB]}$$

and subsequently

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} t : \widehat{\tau} \mathbin{\&} \xi' \sqcup \ell \qquad \Sigma \vdash_{\text{sub}} \ell \sqsubseteq \xi' \sqcup \ell}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \operatorname{ann}_{\ell}(\widehat{t'}) : \widehat{\tau} \mathbin{\&} \xi' \sqcup \ell} \text{ [T-ANN]}$$

Moreover, $\Sigma \vdash_{\text{s}} \xi' \sqcup \ell : \star$ by [S-JOIN].

$t = \textbf{seq}\ t_1\ t_2$ In this case $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{seq}\ t_1\ t_2) = \text{seq}\ \widehat{t_1}\ \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} [\![\xi_1 \sqcup \xi_2]\!]_{\Sigma}$ where $\widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_1)$ and $\widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_2)$.

We can derive

$$\frac{\dfrac{\text{by induction}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1} \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1 \qquad \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_1 \sqcup \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1 \sqcup \xi_2} \text{ [T-SUB]}$$

and

$$\frac{\dfrac{\text{by induction}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2} \qquad \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_2 \qquad \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_1 \sqcup \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_1 \sqcup \xi_2} \text{ [T-SUB]}$$

concluding with

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1 \sqcup \xi_2 \qquad \Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_1 \sqcup \xi_2}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{seq}\ \widehat{t_1}\ \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_1 \sqcup \xi_2} \text{ [T-SEQ]}$$

Moreover, the conservativeness of $\widehat{\tau}_2$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$ follow by induction and $\Sigma \vdash_{\text{s}} \xi_1 \sqcup \xi_2 : \star$ follows by induction and rule [S-JOIN].

$t = (t_1, t_2)$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1, t_2) = (\widehat{t_1}, \widehat{t_2}) : \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \bot$ where $\widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_1)$ and $\widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_2)$.

We can derive

$$\frac{\dfrac{\text{by induction}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1} \qquad \dfrac{\text{by induction}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} (\widehat{t_1}, \widehat{t_2}) : \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle \mathbin{\&} \bot} \text{ [T-PAIR]}$$

By induction, we also have $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$, $\Sigma \vdash_{\text{s}} \xi_1 : \star$ and $\Sigma \vdash_{\text{s}} \xi_2 : \star$. This allows us to conclude $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle$. By [S-LAT], $\Sigma \vdash_{\text{s}} \bot : \star$. Lastly, since $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative by induction, so is $\widehat{\tau}_1\langle \xi_1 \rangle + \widehat{\tau}_2\langle \xi_2 \rangle$.

$t = \textbf{inl}_{\tau_2}(t_1)$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{inl}_{\tau_2}(t_1)) = \text{inl}_{\tau_2}(\widehat{t_1}) : \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \mathbin{\&} \bot$ where $\widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_1)$.

By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \mathbin{\&} \xi_1$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$, $\Sigma \vdash_{\text{s}} \xi_1 : \star$ and $\widehat{\tau}_1$ is conservative. By lemma 3.25, $\bot_{\tau_2}$ is conservative, $\Sigma \vdash_{\text{wft}} \bot_{\tau_2}$ and $\lfloor \bot_{\tau_2} \rfloor = \tau_2$. We can therefore apply [T-INL] and get $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{inl}_{\tau_2}(\widehat{t_1}) : \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle \mathbin{\&} \bot$. By [W-SUM], $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle$ and by [S-LAT], $\Sigma \vdash_{\text{s}} \bot : \star$. Lastly, as both $\widehat{\tau}_1$ and $\bot_{\tau_2}$ are conservative, so is $\widehat{\tau}_1\langle \xi_1 \rangle + \bot_{\tau_2}\langle \bot \rangle$.

$t = \textbf{inl}_{\tau_1}(t_2)$ Analogous to the previous case.

$t = \textbf{proj}_i(t')$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{proj}_i(t')) = \text{proj}_i(\widehat{t'}) : \widehat{\tau}_i \mathbin{\&} \llbracket \xi \sqcup \xi_i \rrbracket_\Sigma$ where $\widehat{t'} : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \xi = \mathcal{R}(\widehat{\Gamma}; \Sigma; t')$ and $i = 1$ or $i = 2$.

By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \xi$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle$, $\Sigma \vdash_{\text{s}} \xi : \star$ and $\widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle$ is conservative. This implies that $\Sigma \vdash_{\text{wft}} \widehat{\tau}_i$ and $\Sigma \vdash_{\text{s}} \xi_i : \star$ hold and that $\widehat{\tau}_i$ is conservative.

We can derive the following subtyping judgment

$$\frac{\begin{array}{cc}\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_1 & \Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_2 \\ \Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_1 \sqcup \xi & \Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_2 \sqcup \xi\end{array}}{\Sigma \vdash_{\text{sub}} \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau}_1\langle\xi_1 \sqcup \xi\rangle \times \widehat{\tau}_2\langle\xi_2 \sqcup \xi\rangle}\ [\textsc{Sub-Prod}]$$

Together with $\Sigma \vdash_{\text{sub}} \xi \sqsubseteq \xi_i \sqcup \xi$ we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}_1\langle\xi_1 \sqcup \xi\rangle \times \widehat{\tau}_2\langle\xi_2 \sqcup \xi\rangle \mathbin{\&} \xi_i \sqcup \xi$ using [T-Sub]. This has the right form to apply [T-Proj], leaving us with $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \text{proj}_i(\widehat{t'}) : \widehat{\tau}_i \mathbin{\&} \xi_i \sqcup \xi$.

$t = \textbf{case}\ t'\ \textbf{of}\ \{\textbf{inl}(x) \to t_1; \textbf{inr}(y) \to t_2\}$ In this case, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \textbf{case}\ \widehat{t'}\ \textbf{of}\ \{\text{inl}(x) \to \widehat{t_1}; \text{inr}(y) \to \widehat{t_2}\} : \llbracket \widehat{\tau}_1 \sqcup \widehat{\tau}_2 \rrbracket_\Sigma \mathbin{\&} \llbracket \xi_1 \sqcup \xi_2 \sqcup \xi_2 \rrbracket_\Sigma$ where $\widehat{t_1} : \widehat{\tau}\langle\xi\rangle + \widehat{\tau}'\langle\xi'\rangle \mathbin{\&} \xi_1 = \mathcal{R}(\widehat{\Gamma}; \Sigma; t_1)$, $\widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2 = \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau} \mathbin{\&} \xi; \Sigma; t_2)$ and $\widehat{t_3} : \widehat{\tau}_3 \mathbin{\&} \xi_3 = \mathcal{R}(\widehat{\Gamma}, y : \widehat{\tau}' \mathbin{\&} \xi'; \Sigma; t_3)$.

By induction, $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}\langle\xi\rangle + \widehat{\tau}'\langle\xi'\rangle \mathbin{\&} \xi_1$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}\langle\xi\rangle + \widehat{\tau}'\langle\xi'\rangle$, $\Sigma \vdash_{\text{s}} \xi_1 : \star$ and $\widehat{\tau}\langle\xi\rangle + \widehat{\tau}'\langle\xi'\rangle$ is conservative. Hence, both $\widehat{\Gamma}, x : \widehat{\tau} \mathbin{\&} \xi$ and $\widehat{\Gamma}, x : \widehat{\tau}' \mathbin{\&} \xi'$ are well-formed under Σ and we can apply the induction hypothesis to the remaining recursive calls. This results in $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau} \mathbin{\&} \xi \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \mathbin{\&} \xi_2$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$, $\Sigma \vdash_{\text{s}} \xi_2 : \star$ and $\widehat{\tau}_2$ conservative, and similarly for the third call.

Using [T-Sub] and lemma 3.35, we can derive $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}\langle\xi\rangle + \widehat{\tau}'\langle\xi'\rangle \mathbin{\&} \xi_1 \sqcup \xi_2 \sqcup \xi_3$, $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau} \mathbin{\&} \xi \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \mathbin{\&} \xi_1 \sqcup \xi_2 \sqcup \xi_3$ and $\Sigma \mid \widehat{\Gamma}, y : \widehat{\tau}' \mathbin{\&} \xi' \vdash_{\text{te}} \widehat{t_3} : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \mathbin{\&} \xi_1 \sqcup \xi_2 \sqcup \xi_3$. This makes it possible to apply [T-Case], leaving us with the desired

$$\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \textbf{case}\ \widehat{t_1}\ \textbf{of}\ \{\text{inl}(x) \to \widehat{t_2}; \text{inr}(y) \to \widehat{t_3}\} : \widehat{\tau}_2 \sqcup \widehat{\tau}_2 \mathbin{\&} \xi_1 \sqcup \xi_2 \sqcup \xi_3$$

Also by lemma 3.35, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2 \sqcup \widehat{\tau}_3$ and $\widehat{\tau}_2 \sqcup \widehat{\tau}_3$ conservative. By [S-Join], $\Sigma \vdash_{\text{s}} \xi_1 \sqcup \xi_2 \sqcup \xi_3 : \star$.

$t = \lambda x : \tau_1.t'$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \lambda x : \tau_1.t') = \Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \mathbin{\&} \beta.\widehat{t'} : \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \bot$ where $\widehat{\tau}_1 \mathbin{\&} \beta \rhd \overline{\beta_i :: \kappa_i} = \mathcal{C}([]; \tau_1)$ and $\widehat{t'} : \widehat{\tau}_2 \mathbin{\&} \xi_2 = \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}_1 \mathbin{\&} \beta; \Sigma, \overline{\beta_i :: \kappa_i}; t')$.

By lemma 3.20 we get $\emptyset \vdash_p \widehat{\tau}_1 \mathbin{\&} \beta \rhd \overline{\beta_i :: \kappa_i}$. Applying lemma 3.14 to the latter judgment gives us $\overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}_1$ and $\overline{\beta_i :: \kappa_i} \vdash_{\text{s}} \beta : \star$. We can now extend the context in the previous two judgments with Σ since the $\overline{\beta_i}$ are fresh. For the same reason, $\widehat{\Gamma}$ is well-formed under $\Sigma, \overline{\beta_i :: \kappa_i}$. By lemma 3.18, $\widehat{\tau}_1$ is conservative. We can conclude that the extended environment $\widehat{\Gamma}, x : \widehat{\tau}_1 \mathbin{\&} \beta$ is well-formed under $\Sigma, \overline{\beta_i :: \kappa_i}$ as well.

We can derive

$$\dfrac{\dfrac{\dfrac{\text{by induction}}{\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \beta \vdash_{\text{te}} \widehat{t'} : \widehat{\tau}_2 \,\&\, \xi_2}}{\Sigma, \overline{\beta_i :: \kappa_i} \mid \widehat{\Gamma} \vdash_{\text{te}} \lambda x : \widehat{\tau}_1 \,\&\, \beta.\widehat{t'} : \widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot}\,[\text{T-Abs}]}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \,\&\, \beta.\widehat{t'} : \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot}\,\overline{[\text{T-AnnAbs}]}_i$$

where $\overline{[\text{T-AnnAbs}]}_i$ denotes an application of rule [T-AnnAbs] for every i. We have omitted the side-conditions of [T-AnnAbs] in the tree above. However, since the $\overline{\beta_i}$ are fresh, we have $\beta_i \notin \text{fav}(\widehat{\Gamma}) \cup \text{fav}(\bot) = \text{fav}(\widehat{\Gamma})$ for all i.

Similarly, we can derive

$$\dfrac{\dfrac{\dfrac{\text{by lemma 3.14}}{\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}_1} \quad \dfrac{\text{by induction}}{\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}_2} \\ \Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{s}} \beta : \star \quad \Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{s}} \xi_2 : \star}{\Sigma, \overline{\beta_i :: \kappa_i} \vdash_{\text{wft}} \widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle}\,[\text{W-Arr}]}{\Sigma \vdash_{\text{wft}} \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle}\,\overline{[\text{W-Forall}]}_i$$

$\Sigma \vdash_{\text{s}} \bot : \star$ follows trivially from [S-Lat]. By induction, $\widehat{\tau}_2$ is conservative, and since we established earlier that $\emptyset \vdash_p \widehat{\tau}_1 \,\&\, \beta \rhd \overline{\beta_i :: \kappa_i}$ holds, the resulting type $\forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ is conservative.

$t = t_1\ t_2$ In this case $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1\ \ t_2) = \widehat{t_1}\ \ \langle\overline{\theta\ \beta_i}\rangle\ \ \widehat{t_2} : [\![\theta\ \widehat{\tau}']\!]_\Sigma \,\&\, [\![\xi_1 \sqcup \theta\ \xi']\!]_\Sigma$ where $\widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1 = \mathcal{R}(\Gamma; \Sigma; t_1)$, $\widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2 = \mathcal{R}(\Gamma; \Sigma; t_2)$, $\widehat{\tau}_2'\langle\beta\rangle \to \widehat{\tau}'\langle\xi'\rangle \rhd \overline{\beta_i} = \mathcal{I}(\widehat{\tau}_1)$ and $\theta = [\beta \mapsto \xi_2] \circ \mathcal{M}([\,]; \widehat{\tau}_2'; \widehat{\tau}_2)$.

By induction, we have $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$, $\Sigma \vdash_{\text{wft}} \widehat{\tau}_1$ and $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$, $\Sigma \vdash_{\text{s}} \xi_1 : \star$ and $\Sigma \vdash_{\text{s}} \xi_2 : \star$ and both $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative.

Since $\mathcal{I}(\widehat{\tau}_1) = \widehat{\tau}_2'\langle\beta\rangle \to \widehat{\tau}'\langle\xi'\rangle \rhd \overline{\beta_i}$, we know that $\widehat{\tau}_1 = \forall\overline{\beta_i' :: \kappa_{\beta_i'}}.\widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_2'\rangle$ for some $\overline{\beta_i'}$. Additionally, $\widehat{\tau}_1$ is conservative. This implies $\emptyset \vdash_p \widehat{\tau}_1' \,\&\, \xi_1' \rhd \overline{\beta_i' :: \kappa_{\beta_i'}}$. By the definitions of the pattern rules we can infer that $\xi_1' = \beta'$ for some $\beta' \in \overline{\beta_i'}$.

Furthermore, due to the way $\mathcal{I}$ is defined, $[\overline{\beta_i/\beta_i'}](\widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_2'\rangle) = \widehat{\tau}_2'\langle\beta\rangle \to \widehat{\tau}'\langle\xi'\rangle$.

Since by induction $\Sigma \vdash_{\text{wft}} \widehat{\tau}_2$ and $\Sigma \vdash_{\text{s}} \xi_2 : \star$, we can apply lemma 3.34 in order to get $\Sigma \vdash_{\text{s}} \theta\beta_i : \kappa_{\beta_i'}$ for all i.

We can derive

$$\dfrac{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1} : \forall\overline{\beta_i' :: \kappa_{\beta_i'}}.\widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_2'\rangle \,\&\, \xi_1 \qquad \overline{\Sigma \vdash_{\text{s}} \theta\beta_i : \kappa_{\beta_i'}}}{\Sigma \mid \widehat{\Gamma} \vdash_{\text{te}} \widehat{t_1}\ \overline{\langle\theta\beta_i\rangle} : [\overline{\theta\beta_i\ /\ \beta_i'}]\ (\widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_1''\langle\xi_2'\rangle) \,\&\, \xi_1}\,\overline{[\text{T-AnnApp}]}_i$$

We define $\theta' = [\overline{\theta\beta_i\ /\ \beta_i'}]$. Note that $\theta' = \theta \circ [\overline{\beta_i/\beta_i'}]$. It follows, by using lemma 3.34, that $\theta'\widehat{\tau}_1' = \theta([\overline{\beta_i/\beta_i'}]\widehat{\tau}_1') = \theta\widehat{\tau}_2' = \widehat{\tau}_2$. Similarly, $\theta'\xi_1' = \theta([\overline{\beta_i/\beta_i'}]\xi_1') = \theta\beta = \xi_2$, $\theta'\widehat{\tau}_1'' = \theta([\overline{\beta_i/\beta_i'}]\widehat{\tau}_1'') = \theta\widehat{\tau}'$ and $\theta'\xi_2' = \theta([\overline{\beta_i/\beta_i'}]\xi_2') = \theta\xi'$.

This allows us to rewrite the above derivation as $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t_1}\ \overline{\langle\theta\beta_i\rangle} : \widehat{\tau}_2\langle\xi_2\rangle \to \theta\widehat{\tau}_1'\langle\theta\xi'\rangle \mathbin{\&} \xi_1$. By applying [T-Sub] and [Sub-Arr], we can further derive $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t_1}\ \overline{\langle\theta\beta_i\rangle} : \widehat{\tau}_2\langle\xi_2\rangle \to \theta\widehat{\tau}_1'\langle\xi_1 \sqcup \theta\xi'\rangle \mathbin{\&} \xi_1 \sqcup \theta\xi'$.

This has the right shape for applying [T-App] which lets us conclude $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \widehat{t_1}\ \overline{\langle\theta\beta_i\rangle}\ \widehat{t_2} : \theta\widehat{\tau}' \mathbin{\&} \xi_1 \sqcup \theta\xi'$.

Because all terms in the image of θ are of the right sort and we know $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}_1$, we can conclude $\Sigma \vdash_{\mathrm{wft}} \theta\widehat{\tau}'$ and $\Sigma \vdash_{\mathrm{s}} \theta\xi' : \star$ from its premises. Applying [S-Join] results in $\Sigma \vdash_{\mathrm{s}} \xi_1 \sqcup \theta\xi' : \star$.

Since $\widehat{\tau}_1$ is conservative, so is $\widehat{\tau}_1''$ and therefore also $\theta'\widehat{\tau}_1'' = \theta\widehat{\tau}'$.

$t = \mu x : \tau.t'$ In this case $\mathcal{R}(\widehat{\Gamma}; \Sigma; \mu x : \tau.t') = \mu x : \widehat{\tau}_n \mathbin{\&} \xi_n.\widehat{t'} : \widehat{\tau}_n \mathbin{\&} \xi_n$ for some $\widehat{t'}$, $\widehat{\tau}_n$ and ξ_n such that $\widehat{t'} : \widehat{\tau}_n \mathbin{\&} \xi_n = \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}_n \mathbin{\&} \xi_n; \Sigma; t')$. These exist because by assumption, $\mathcal{R}$ has produced a result, and that only happens if the fixpoint iteration terminated after a finite number of iterations. We denote the number of iterations by n.

We have $\xi_0 = \bot$. Hence, $\Sigma \vdash_{\mathrm{s}} \xi_0 : \star$ follows trivially from [S-Lat]. $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}_0$ and the fact that $\widehat{\tau}_0$ is conservative for $\widehat{\tau}_0 = \bot_\tau$ follows from lemma 3.25.

For every iteration i we can apply the induction hypothesis in order to get $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}_{i+1}$ and $\Sigma \vdash_{\mathrm{s}} \xi_{i+1} : \star$ and the conservativeness of $\widehat{\tau}_{i+1}$ from the facts established about the previous iteration.

By repeating this step n times, we can eventually derive $\Sigma \mid \widehat{\Gamma}, x : \widehat{\tau}_n \mathbin{\&} \xi_n \vdash_{\mathrm{te}} \widehat{t'} : \widehat{\tau}_n \mathbin{\&} \xi_n$, $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}_n$ and $\Sigma \vdash_{\mathrm{s}} \xi_n : \star$. Furthermore, $\widehat{\tau}_n$ is conservative.

We can conclude $\Sigma \mid \widehat{\Gamma} \vdash_{\mathrm{te}} \mu x : \widehat{\tau}_n \mathbin{\&} \xi_n.\widehat{t'} : \widehat{\tau}_n \mathbin{\&} \xi_n$ by [T-Fix].

□

Proof of Lemma 3.47. By induction on t.

$t = x$ We have $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; x) = x : \widehat{\Gamma_1}(x)$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; x) = x : \widehat{\Gamma_2}(x)$. The result follows by assumption.

$t = ()$ The assigned type and effect pair is always $\widehat{\mathrm{unit}} \mathbin{\&} \bot$, therefore the result is trivially true by reflexivity.

$t = \mathbf{ann}_\ell(t')$ We have $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; \mathrm{ann}_\ell(t')) = \mathrm{ann}_\ell(\widehat{t_1'}) : \widehat{\tau}_1' \mathbin{\&} [\![\xi_1' \sqcup \ell]\!]_\Sigma$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; \mathrm{ann}_\ell(t')) = \mathrm{ann}_\ell(\widehat{t_2'}) : \widehat{\tau}_2' \mathbin{\&} [\![\xi_2' \sqcup \ell]\!]_\Sigma$.

By induction, $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; t') = \widehat{t_1'} : \widehat{\tau}_1' \mathbin{\&} \xi_1'$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; t') = \widehat{t_2'} : \widehat{\tau}_2' \mathbin{\&} \xi_2'$ such that $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1' \leqslant \widehat{\tau}_2'$ and $\Sigma \vdash_{\mathrm{sub}} \xi_1' \sqsubseteq \xi_2'$. Using the lattice properties, we can derive $\Sigma \vdash_{\mathrm{sub}} \xi_1' \sqcup \ell \sqsubseteq \xi_2' \sqcup \ell$.

$t = \mathbf{seq}\ t'\ t''$ We have $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; \mathrm{seq}\ t'\ t'') = \mathrm{seq}\ \widehat{t_1'}\ \widehat{t_1''} : \widehat{\tau}_1'' \mathbin{\&} [\![\xi_1' \sqcup \xi_1'']\!]_\Sigma$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; \mathrm{seq}\ t'\ t'') = \mathrm{seq}\ \widehat{t_2'}\ \widehat{t_2''} : \widehat{\tau}_2'' \mathbin{\&} [\![\xi_2' \sqcup \xi_2'']\!]_\Sigma$.

By induction, $\Sigma \vdash_{\mathrm{sub}} \widehat{\tau}_1'' \leqslant \widehat{\tau}_2''$, $\Sigma \vdash_{\mathrm{sub}} \xi_1' \sqsubseteq \xi_2'$ and $\Sigma \vdash_{\mathrm{sub}} \xi_1'' \sqsubseteq \xi_2''$. Using the lattice properties, we can derive $\Sigma \vdash_{\mathrm{sub}} \xi_1' \sqcup \xi_1'' \sqsubseteq \xi_2' \sqcup \xi_2''$.

$t = (t_1, t_2)$ By applying [Sub-Prod] to the induction hypotheses for both recursive calls.

$t = \textsf{inl}_{\tau_2}(t_1)$ By applying [Sub-Sum] to the induction hypothesis for the recursive call and lemmas 3.23 and 3.26.

$t = \textsf{inl}_{\tau_1}(t_2)$ Analogous to the previous case.

$t = \textsf{proj}_i(t')$ By applying lemma 2.30 to the induction hypothesis.

$t = \textbf{case } t_1 \textbf{ of } \{\textsf{inl}(x) \to t_2; \textsf{inr}(x) \to t_3\}$ By induction, we get

$$\mathcal{R}(\widehat{\Gamma_1}; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1^l\langle\xi_1^l\rangle + \widehat{\tau}_1^r\langle\xi_1^r\rangle \,\&\, \xi_1 \text{ and } \mathcal{R}(\widehat{\Gamma_2}; \Sigma; t_1) = \widehat{t_4} : \widehat{\tau}_4^l\langle\xi_4^l\rangle + \widehat{\tau}_4^r\langle\xi_4^r\rangle \,\&\, \xi_4$$

such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^l\langle\xi_1^l\rangle + \widehat{\tau}_1^r\langle\xi_1^r\rangle \leqslant \widehat{\tau}_4^l\langle\xi_4^l\rangle + \widehat{\tau}_4^r\langle\xi_4^r\rangle$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_4$. By lemma 2.30, we get $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^l \leqslant \widehat{\tau}_4^l$ and $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^r \leqslant \widehat{\tau}_4^r$ as well as $\Sigma \vdash_{\text{sub}} \xi_1^l \sqsubseteq \xi_4^l$ and $\Sigma \vdash_{\text{sub}} \xi_1^r \sqsubseteq \xi_4^r$.

This allows us to apply the induction hypothesis to the recursive call for both branches of the case-expression, and we get

$$\mathcal{R}(\widehat{\Gamma_1}, x : \widehat{\tau}_1^l \,\&\, \xi_1^l; \Sigma; t_2) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2 \text{ and } \mathcal{R}(\widehat{\Gamma_2}, x : \widehat{\tau}_4^l \,\&\, \xi_4^l; \Sigma; t_2) = \widehat{t_5} : \widehat{\tau}_5 \,\&\, \xi_5$$

such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \leqslant \widehat{\tau}_5$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqsubseteq \xi_5$. Similarly, for the other branch we have

$$\mathcal{R}(\widehat{\Gamma_1}, x : \widehat{\tau}_1^r \,\&\, \xi_1^r; \Sigma; t_3) = \widehat{t_3} : \widehat{\tau}_3 \,\&\, \xi_3 \text{ and } \mathcal{R}(\widehat{\Gamma_2}, x : \widehat{\tau}_4^r \,\&\, \xi_4^r; \Sigma; t_3) = \widehat{t_6} : \widehat{\tau}_6 \,\&\, \xi_6$$

such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}_3 \leqslant \widehat{\tau}_6$ and $\Sigma \vdash_{\text{sub}} \xi_3 \sqsubseteq \xi_6$.

Using lemma 3.43 and lemma 3.44, we can derive

$$\begin{aligned} &((\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_3 \,\&\, \xi_3)) \sqcup ((\widehat{\tau}_5 \,\&\, \xi_5) \sqcup (\widehat{\tau}_6 \,\&\, \xi_6)) \\ \equiv_\Sigma &((\widehat{\tau}_2 \,\&\, \xi_2) \sqcup (\widehat{\tau}_5 \,\&\, \xi_5)) \sqcup ((\widehat{\tau}_3 \,\&\, \xi_3) \sqcup (\widehat{\tau}_6 \,\&\, \xi_6)) \\ \equiv_\Sigma &(\widehat{\tau}_5 \,\&\, \xi_5) \sqcup (\widehat{\tau}_6 \,\&\, \xi_6) \end{aligned}$$

and therefore $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \leqslant \widehat{\tau}_5 \sqcup \widehat{\tau}_6$ and $\Sigma \vdash_{\text{sub}} \xi_2 \sqcup \xi_3 \sqsubseteq \xi_5 \sqcup \xi_6$. Using the lattice properties, we also have $\Sigma \vdash_{\text{sub}} \xi_1 \sqcup \xi_2 \sqcup \xi_3 \sqsubseteq \xi_4 \sqcup \xi_5 \sqcup \xi_6$.

Since $\mathcal{R}(\widehat{\Gamma_1}; \Sigma; t) = \textbf{case } \widehat{t_1} \textbf{ of } \{\text{inl}(x) \to \widehat{t_2}; \text{inr}(x) \to \widehat{t_3}\} : [\![\widehat{\tau}_2 \sqcup \widehat{\tau}_3]\!]_\Sigma \,\&\, [\![\xi_1 \sqcup \xi_2 \sqcup \xi_3]\!]_\Sigma$ and $\mathcal{R}(\widehat{\Gamma_2}; \Sigma; t) = \textbf{case } \widehat{t_4} \textbf{ of } \{\text{inl}(x) \to \widehat{t_5}; \text{inr}(x) \to \widehat{t_6}\} : [\![\widehat{\tau}_5 \sqcup \widehat{\tau}_6]\!]_\Sigma \,\&\, [\![\xi_4 \sqcup \xi_5 \sqcup \xi_6]\!]_\Sigma$, this completes the proof.

$t = \lambda x : \tau_1.t'$ We note that the argument position can be treated invariantly due to the fact that pattern completions are unique. The desired result then follows by applying [Sub-Arr] using the induction hypothesis for the covariant position, followed by repeated applications of [Sub-Forall].

$t = t'\ t''$ By induction, we get $\mathcal{R}(\widehat{\Gamma_1};\Sigma;t') = \widehat{t'_1} : \widehat{\tau}'_1 \,\&\, \xi'_1$ and $\mathcal{R}(\widehat{\Gamma_2};\Sigma;t') = \widehat{t'_2} : \widehat{\tau}'_2 \,\&\, \xi'_2$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ and $\Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi'_2$ as well as $\mathcal{R}(\widehat{\Gamma_1};\Sigma;t'') = \widehat{t''_1} : \widehat{\tau}''_1 \,\&\, \xi''_1$ and $\mathcal{R}(\widehat{\Gamma_2};\Sigma;t'') = \widehat{t''_2} : \widehat{\tau}''_2 \,\&\, \xi''_2$ such that $\Sigma \vdash_{\text{sub}} \widehat{\tau}''_1 \leqslant \widehat{\tau}''_2$ and $\Sigma \vdash_{\text{sub}} \xi''_1 \sqsubseteq \xi''_2$.

We know that $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ are conservative and function types, therefore they are of the form $\widehat{\tau}'_1 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}'^a_1\langle\xi'^a_1\rangle \to \widehat{\tau}'^r_1\langle\xi'^r_1\rangle$ and $\widehat{\tau}'_2 = \forall\overline{\beta_j :: \kappa_{\beta_j}}.\widehat{\tau}'^a_2\langle\xi'^a_2\rangle \to \widehat{\tau}'^r_2\langle\xi'^r_2\rangle$.

We can deconstruct the derivation of $\Sigma \vdash_{\text{sub}} \widehat{\tau}'_1 \leqslant \widehat{\tau}'_2$ by repeated application of lemma 2.30, once for every quantifier and finally for the arrow type itself. We end up with $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \widehat{\tau}'^r_1 \leqslant \widehat{\tau}'^r_2$ and $\Sigma, \overline{\beta_j :: \kappa_{\beta_j}} \vdash_{\text{sub}} \xi'^r_1 \sqsubseteq \xi'^r_2$.

Without loss of generality we can assume that the $\overline{\beta_j}$ are fresh and therefore

$$\widehat{\tau}'^a_1\langle\xi'^a_1\rangle \to \widehat{\tau}'^r_1\langle\xi'^r_1\rangle \triangleright \overline{\beta_j :: \kappa_{\beta_j}} = \mathcal{I}(\widehat{\tau}'_1) \text{ and } \widehat{\tau}'^a_2\langle\xi'^a_2\rangle \to \widehat{\tau}'^r_2\langle\xi'^r_2\rangle \triangleright \overline{\beta_j :: \kappa_{\beta_j}} = \mathcal{I}(\widehat{\tau}'_2).$$

Since $\widehat{\tau}'^a_1 \& \xi'^a_1$ and $\widehat{\tau}'^a_2 \& \xi'^a_2$ are patterns of the same underlying type, we can assume that $\widehat{\tau}'^a_1 \,\&\, \xi'^a_1 = \widehat{\tau}'^a_2 \,\&\, \xi'^a_2$ because pattern types are unique up to renaming. We know that ξ'^a_1 and ξ'^a_2 must be single variables because they are part of a pattern, so suppose $\xi'^a_1 = \xi'^a_2 = \beta_1$.

Then we get one substitution for each call to $\mathcal{R}$ given by

$$\theta_1 = [\beta_1 \mapsto \xi''_1] \circ \mathcal{M}([\,];\widehat{\tau}'^a_1;\widehat{\tau}''_1)) \text{ and } \theta_2 = [\beta_1 \mapsto \xi''_2] \circ \mathcal{M}([\,];\widehat{\tau}'^a_2;\widehat{\tau}''_2)).$$

By lemma 3.45, $\Sigma \vdash_{\text{sub}} \theta_1(\gamma) \sqsubseteq \theta_2(\gamma)$ holds for all variables γ. That allows us to apply lemma 2.17 and lemma 3.46 in order to derive $\Sigma \vdash_{\text{sub}} \theta_1\widehat{\tau}'^r_1 \leqslant \theta_2\widehat{\tau}'^r_2$ and $\Sigma \vdash_{\text{sub}} \theta_1\xi'^r_1 \sqsubseteq \theta_2\xi'^r_2$. Together with $\Sigma \vdash_{\text{sub}} \xi'_1 \sqsubseteq \xi'_2$ deduced in the beginning, we also have $\Sigma \vdash_{\text{sub}} \xi'_1 \sqcup \xi'^r_1 \sqsubseteq \xi'_2 \sqcup \xi'^r_2$.

As we have $\mathcal{R}(\widehat{\Gamma_1};\Sigma;t'\ t'') = \widehat{t'_1}\ \widehat{t''_1} : [\![\theta_1\widehat{\tau}''^r_1]\!]_\Sigma \,\&\, [\![\xi'_1 \sqcup \xi'^r_1]\!]_\Sigma$ and $\mathcal{R}(\widehat{\Gamma_2};\Sigma;t'\ t'') = \widehat{t'_2}\ \widehat{t''_2} : [\![\theta_2\widehat{\tau}''^r_2]\!]_\Sigma \,\&\, [\![\xi'_2 \sqcup \xi'^r_2]\!]_\Sigma$, this completes the proof.

$t = \mu x : \tau.t'$ By assumption, we have $\mathcal{R}(\widehat{\Gamma_1};\Sigma;\mu x : \tau.t') = \mu x : \widehat{\tau}_1 \,\&\, \xi_1.\widehat{t'_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\widehat{\Gamma_2};\Sigma;\mu x : \tau.t') = \mu x : \widehat{\tau}_2 \,\&\, \xi_2.\widehat{t'_2} : \widehat{\tau}_2 \,\&\, \xi_2$. Therefore, both fixpoint iterations converged and there are sequences $(\widehat{\tau}^{(i)}_1 \,\&\, \xi^{(i)}_1)_{i\leqslant m}$ and $(\widehat{\tau}^{(i)}_2 \,\&\, \xi^{(i)}_2)_{i\leqslant n}$ such that $\widehat{\tau}^{(0)}_1 \,\&\, \xi^{(0)}_1 = \widehat{\tau}^{(0)}_2 \,\&\, \xi^{(0)}_2 = \bot_\tau \,\&\, \bot$, $t'^{(i+1)}_1 : \widehat{\tau}^{(i+1)}_1 \,\&\, \xi^{(i+1)}_1 = \mathcal{R}(\widehat{\Gamma_1}, x : \widehat{\tau}^{(i)}_1 \,\&\, \xi^{(i)}_1;\Sigma;t')$ and $t'^{(i+1)}_2 : \widehat{\tau}^{(i+1)}_2 \,\&\, \xi^{(i+1)}_2 = \mathcal{R}(\widehat{\Gamma_2}, x : \widehat{\tau}^{(i)}_2 \,\&\, \xi^{(i)}_2;\Sigma;t')$ for some sequences $(t'^{(i)}_1)_{1\leqslant i\leqslant m}$ and $(t'^{(i)}_2)_{1\leqslant i\leqslant n}$.

We proceed by showing that both sequences of type and effect pairs are monotonically increasing.

Claim: For all $i \leqslant m$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}^{(i)}_1 \leqslant \widehat{\tau}^{(i+1)}_1$ and $\Sigma \vdash_{\text{sub}} \xi^{(i)}_1 \sqsubseteq \xi^{(i+1)}_1$.

Proof: By induction on i.

$i = 0$ We have $\widehat{\tau}^{(0)}_1 = \bot_\tau$ and $\xi^{(0)}_1 = \bot_\star$. By lemma 3.26, $\Sigma \vdash_{\text{sub}} \bot_\tau \leqslant \widehat{\tau}^{(1)}_1$ and by lemma 3.23 $\Sigma \vdash_{\text{sub}} \bot_\star \sqsubseteq \xi^{(1)}_1$.

$i = i' + 1$ By induction, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i')} \leqslant \widehat{\tau}_1^{(i'+1)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i')} \sqsubseteq \xi_1^{(i'+1)}$. Since we have $\mathcal{R}(\widehat{\Gamma_1}, x{:}\widehat{\tau}_1^{(i')} \,\&\, \xi_1^{(i')}; \Sigma; t') = \widehat{t'_1}^{(i'+1)} {:} \widehat{\tau}_1^{(i'+1)} \,\&\, \xi_1^{(i'+1)}$ and $\mathcal{R}(\widehat{\Gamma_1}, x{:} \widehat{\tau}_1^{(i'+1)} \,\&\, \xi_1^{(i'+1)}; \Sigma; t') = \widehat{t'_1}^{(i'+2)} : \widehat{\tau}_1^{(i'+2)} \,\&\, \xi_1^{(i'+2)}$, we can apply the outer induction hypothesis, resulting in $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i'+1)} \leqslant \widehat{\tau}_1^{(i'+2)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i'+1)} \sqsubseteq \xi_1^{(i'+2)}$.

■

Claim: For all $i \leqslant n$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2^{(i)} \leqslant \widehat{\tau}_2^{(i+1)}$ and $\Sigma \vdash_{\text{sub}} \xi_2^{(i)} \sqsubseteq \xi_2^{(i+1)}$.

Proof: Analogously to that of the previous claim. ■

We can now relate both sequences.

Claim: For all $i \leqslant \min\{m, n\}$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i)} \leqslant \widehat{\tau}_2^{(i)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i)} \sqsubseteq \xi_2^{(i)}$.

Proof: By induction on i.

$i = 0$ $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(0)} \leqslant \widehat{\tau}_2^{(0)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(0)} \sqsubseteq \xi_2^{(0)}$ hold by [Sub-Refl] and the reflexivity of subsumption.

$i = i' + 1$ By induction, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i')} \leqslant \widehat{\tau}_2^{(i')}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i')} \sqsubseteq \xi_2^{(i')}$. Moreover, we can apply the outer induction hypothesis to $\mathcal{R}(\widehat{\Gamma_1}, x : \widehat{\tau}_1^{(i')} \,\&\, \xi_1^{(i')}; \Sigma; t') = \widehat{t'_1}^{(i'+1)} {:} \widehat{\tau}_1^{(i'+1)} \,\&\, \xi_1^{(i'+1)}$ and $\mathcal{R}(\widehat{\Gamma_2}, x{:}\widehat{\tau}_2^{(i')} \,\&\, \xi_2^{(i')}; \Sigma; t') = \widehat{t'_2}^{(i'+1)} : \widehat{\tau}_2^{(i'+1)} \,\&\, \xi_2^{(i'+1)}$. This results in $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i'+1)} \leqslant \widehat{\tau}_2^{(i'+1)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i'+1)} \sqsubseteq \xi_2^{(i'+1)}$.

■

We distinguish three cases.

- If $m = n$, from the previous claim we get $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(m)} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(m)} \sqsubseteq \xi_2^{(n)}$.
- If $m < n$, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(m)} \leqslant \widehat{\tau}_2^{(m)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(m)} \sqsubseteq \xi_2^{(m)}$ as well as $\Sigma \vdash_{\text{sub}} \widehat{\tau}_2^{(i)} \leqslant \widehat{\tau}_2^{(i+1)}$ and $\Sigma \vdash_{\text{sub}} \xi_2^{(i)} \sqsubseteq \xi_2^{(i+1)}$ for all $m \leqslant i < n$. By transitivity, we get $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(m)} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(m)} \sqsubseteq \xi_2^{(n)}$.
- If $m > n$, we can prove the following claim by induction.

  Claim: For all $n \leqslant i < m$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i)} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i)} \sqsubseteq \xi_2^{(n)}$.

  Proof: By induction on i.

  $i = n$ This follows from a previous claim.

  $i = i' + 1$ By induction, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i')} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i')} \sqsubseteq \xi_2^{(n)}$. Moreover, we can apply the outer induction hypothesis to $\mathcal{R}(\widehat{\Gamma_1}, x{:}\widehat{\tau}_1^{(i')} \,\&\, \xi_1^{(i')}; \Sigma; t') = \widehat{t'_1}^{(i'+1)} : \widehat{\tau}_1^{(i'+1)} \,\&\, \xi_1^{(i'+1)}$ and $\mathcal{R}(\widehat{\Gamma_2}, x : \widehat{\tau}_2^{(n)} \,\&\, \xi_2^{(n)}; \Sigma; t') = \widehat{t'_2}^{(n)} : \widehat{\tau}_2^{(n)} \,\&\, \xi_2^{(n)}$. The latter holds because $\widehat{\tau}_2^{(n)} \,\&\, \xi_2^{(n)}$ is a fixpoint. This results in $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(i'+1)} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(i'+1)} \sqsubseteq \xi_2^{(n)}$.

■

In particular, we can conclude $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1^{(m)} \leqslant \widehat{\tau}_2^{(n)}$ and $\Sigma \vdash_{\text{sub}} \xi_1^{(m)} \sqsubseteq \xi_2^{(n)}$.

Since $\widehat{\tau}_1 \,\&\, \xi_1 = \widehat{\tau}_1^{(m)} \,\&\, \xi_1^{(m)}$ and $\widehat{\tau}_2 \,\&\, \xi_2 = \widehat{\tau}_2^{(m)} \,\&\, \xi_2^{(m)}$, we have shown the desired result $\Sigma \vdash_{\text{sub}} \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ and $\Sigma \vdash_{\text{sub}} \xi_1 \sqsubseteq \xi_2$.

□

Proof of Theorem 3.48. By induction on t.

$t = x$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; x) = x : \widehat{\Gamma}(x)$. Suppose $\widehat{\Gamma}(x) = \widehat{\tau} \,\&\, \xi$. By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} x : \tau$. As [U-VAR] is the only matching rule, we have $\lfloor\widehat{\Gamma}\rfloor(x) = \tau$ and therefore $\lfloor\widehat{\tau}\rfloor = \tau$. Also, $\lfloor x \rfloor = x$.

$t = ()$ We have $\mathcal{R}(\widehat{\Gamma}; \Sigma; ()) = () : \widehat{\text{unit}} : \bot$ and clearly $\lfloor () \rfloor = ()$. By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} () : \tau$. As the only rule that applies is [U-UNIT], we have $\tau = \text{unit}$ and therefore $\lfloor\widehat{\text{unit}}\rfloor = \text{unit} = \tau$.

$t = \textbf{ann}_\ell(t')$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} \text{ann}_\ell(t') : \tau$. This must be by [U-ANN], and therefore $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t' : \tau$. By induction, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t') = \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ such that $\lfloor\widehat{t'}\rfloor = t'$ and $\lfloor\widehat{\tau}'\rfloor = \tau$.

Then we also have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{ann}_\ell(t')) = \text{ann}_\ell(\widehat{t'}) : \widehat{\tau}' \,\&\, [\![\xi \sqcup \ell]\!]_\Sigma$ with $\lfloor \text{ann}_\ell(\widehat{t'}) \rfloor = \text{ann}_\ell(\lfloor\widehat{t'}\rfloor) = \text{ann}_\ell(t')$ and $\lfloor\widehat{\tau}'\rfloor = \tau$.

$t = \textbf{seq}\ t_1\ t_2$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} \text{seq}\ t_1\ t_2 : \tau$. The only rule that applies is [U-SEQ], and therefore $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t_1 : \tau_1$ for some τ_1 and $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t_2 : \tau$.

By induction, we have $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_2) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$ such that $\lfloor\widehat{t_1}\rfloor = t_1$, $\lfloor\widehat{t_2}\rfloor = t_2$, $\lfloor\widehat{\tau}_1\rfloor = \tau_1$ and $\lfloor\widehat{\tau}_2\rfloor = \tau$.

Then we also have $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{seq}\ t_1\ t_2) = \text{seq}\ \widehat{t_1}\ \widehat{t_2} : \widehat{\tau}_2 \,\&\, [\![\xi_1 \sqcup \xi_2]\!]_\Sigma$ with $\lfloor \text{seq}\ \widehat{t_1}\ \widehat{t_2} \rfloor = \text{seq}\ \lfloor\widehat{t_1}\rfloor\ \lfloor\widehat{t_2}\rfloor = \text{seq}\ t_1\ t_2$ and $\lfloor\widehat{\tau}_2\rfloor = \tau$.

$t = (t_1, t_2)$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} (t_1, t_2) : \tau$. The only matching rule is [U-PAIR], therefore we have $\tau = \tau_1 \times \tau_2$ for some τ_1, τ_2, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t_1 : \tau_1$ and $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t_2 : \tau_2$.

By induction we get $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_2) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$ such that $\lfloor\widehat{t_1}\rfloor = t_1$, $\lfloor\widehat{\tau}_1\rfloor = \tau_1$, $\lfloor\widehat{t_2}\rfloor = t_2$ and $\lfloor\widehat{\tau}_2\rfloor = \tau_2$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1, t_2) = (\widehat{t_1}, \widehat{t_2}) : \widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot$ and we have $\lfloor(\widehat{t_1}, \widehat{t_2})\rfloor = (\lfloor\widehat{t_1}\rfloor, \lfloor\widehat{t_2}\rfloor) = (t_1, t_2)$ and $\lfloor\widehat{\tau}_1\langle\xi_1\rangle \times \widehat{\tau}_2\langle\xi_2\rangle\rfloor = \lfloor\widehat{\tau}_1\rfloor \times \lfloor\widehat{\tau}_2\rfloor = \tau_1 \times \tau_2 = \tau$.

$t = \textbf{inl}_{\tau_2}(t_1)$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} \text{inl}_{\tau_2}(t_1) : \tau$. The only matching rule is [U-INL], therefore we have $\tau = \tau_1 + \tau_2$ for some τ_1 such that $\lfloor\widehat{\Gamma}\rfloor \vdash_{\text{t}} t_1 : \tau_1$.

By induction we get $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ such that $\lfloor\widehat{t_1}\rfloor = t_1$ and $\lfloor\widehat{\tau}_1\rfloor = \tau_1$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; \text{inl}_{\tau_2}(t_1)) = \text{inl}_{\tau_2}(\widehat{t_1}) : \widehat{\tau}_1\langle\xi_1\rangle + \bot_{\tau_2}\langle\bot\rangle \,\&\, \bot$ and we have $\lfloor\text{inl}_{\tau_2}(\widehat{t_1})\rfloor = \text{inl}_{\tau_2}(\lfloor\widehat{t_1}\rfloor) = \text{inl}_{\tau_2}(t_1)$ and $\lfloor\widehat{\tau}_1\langle\xi_1\rangle + \bot_{\tau_2}\langle\bot\rangle\rfloor = \lfloor\widehat{\tau}_1\rfloor + \lfloor\bot_{\tau_2}\rfloor = \tau_1 + \tau_2 = \tau$. In the second to last step, $\lfloor\bot_{\tau_2}\rfloor = \tau_2$ follows from Lemma 3.25.

$t = \textbf{inl}_{\tau_1}(t_2)$ Analogously to the previous case.

$t = \textsf{\textbf{proj}}_i(t')$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \mathrm{proj}_i(t') : \tau$. The only matching rule is [U-Proj], therefore we have $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} t' : \tau_1 \times \tau_2$ for some τ_1 and τ_2 such that $\tau = \tau_i$.

By induction, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t') = \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ such that $\lfloor\widehat{t'}\rfloor = t'$ and $\lfloor\widehat{\tau}'\rfloor = \tau_1 \times \tau_2$. By theorem 3.36, $\widehat{\tau}'$ is conservative. Therefore, $\widehat{\tau}' = \widehat{\tau}'_1\langle\xi'_1\rangle \times \widehat{\tau}'_2\langle\xi'_2\rangle$ such that $\lfloor\widehat{\tau}'_1\rfloor = \tau_1$ and $\lfloor\widehat{\tau}'_2\rfloor = \tau_2$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; \mathrm{proj}_i(t')) = \mathrm{proj}_i(\widehat{t'}) : \widehat{\tau}'_i \,\&\, \llbracket\xi \sqcup \xi_i\rrbracket_\Sigma$ and $\lfloor\mathrm{proj}_i(\widehat{t'})\rfloor = \mathrm{proj}_i(\lfloor\widehat{t'}\rfloor) = \mathrm{proj}_i(t')$, $\lfloor\widehat{\tau}'_i\rfloor = \tau_i = \tau$.

$t = \textbf{case } t' \textbf{ of } \{\textsf{\textbf{inl}}(x) \to t_1; \textsf{\textbf{inr}}(x) \to t_2\}$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \textbf{case } t' \textbf{ of } \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(x) \to t_2\} : \tau$. Only [U-Case] applies, hence $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} t' : \tau_1 + \tau_2$ for some τ_1, τ_2 and $\lfloor\widehat{\Gamma}\rfloor, x : \tau_1 \vdash_{\mathrm{t}} t_1 : \tau$ and $\lfloor\widehat{\Gamma}\rfloor, x : \tau_2 \vdash_{\mathrm{t}} t_2 : \tau$.

By induction, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t') = \widehat{t'} : \widehat{\tau}' \,\&\, \xi'$ such that $\lfloor\widehat{t'}\rfloor = t'$ and $\lfloor\widehat{\tau}'\rfloor = \tau_1 + \tau_2$. By theorem 3.36, $\widehat{\tau}'$ is conservative and $\Sigma \vdash_{\mathrm{wft}} \widehat{\tau}'$. Therefore, $\widehat{\tau}' = \widehat{\tau}'_1\langle\xi'_1\rangle + \widehat{\tau}'_2\langle\xi'_2\rangle$ such that $\lfloor\widehat{\tau}'_1\rfloor = \tau_1$ and $\lfloor\widehat{\tau}'_2\rfloor = \tau_2$.

Note that only [W-Sum] applies to the well-formedness judgment. But then $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ are also conservative and $\Sigma \vdash_{\mathrm{s}} \xi'_1 : \star$ and $\Sigma \vdash_{\mathrm{s}} \xi'_2 : \star$ hold. Hence, $\widehat{\Gamma}, x : \widehat{\tau}'_1 \,\&\, \xi'_1$ and $\widehat{\Gamma}, x : \widehat{\tau}'_2 \,\&\, \xi'_2$ are both well-formed under Σ. Moreover, $\lfloor\widehat{\Gamma}, x : \widehat{\tau}'_1 \,\&\, \xi'_1\rfloor = \lfloor\widehat{\Gamma}\rfloor, x : \tau_1$ and $\lfloor\widehat{\Gamma}, x : \widehat{\tau}'_2 \,\&\, \xi'_2\rfloor = \lfloor\widehat{\Gamma}\rfloor, x : \tau_2$.

Now we can apply the induction hypothesis to both case branches, resulting in $\mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}'_1 \,\&\, \xi'_1; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}'_2 \,\&\, \xi'_2; \Sigma; t_2) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$ such that $\lfloor\widehat{t_1}\rfloor = t_1$, $\lfloor\widehat{\tau}_1\rfloor = \tau$, $\lfloor\widehat{t_2}\rfloor = t_2$ and $\lfloor\widehat{\tau}_2\rfloor = \tau$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; \textbf{ case } t' \textbf{ of } \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(x) \to t_2\}) = \textbf{case } \widehat{t'} \textbf{ of } \{\mathrm{inl}(x) \to \widehat{t_1}; \mathrm{inr}(x) \to \widehat{t_2}\} : \llbracket\widehat{\tau}_1 \sqcup \widehat{\tau}_2\rrbracket_\Sigma \,\&\, \llbracket\xi' \sqcup \xi_1 \sqcup \xi_2\rrbracket$.. Now, clearly

$$
\begin{aligned}
&\lfloor \textbf{ case } \widehat{t'} \textbf{ of } \{\mathrm{inl}(x) \to \widehat{t_1}; \mathrm{inr}(x) \to \widehat{t_2}\}\rfloor \\
=&\textbf{case } \lfloor\widehat{t'}\rfloor \textbf{ of } \{\mathrm{inl}(x) \to \lfloor\widehat{t_1}\rfloor; \mathrm{inr}(x) \to \lfloor\widehat{t_2}\rfloor\} \\
=&\textbf{case } t' \textbf{ of } \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(x) \to t_2\}.
\end{aligned}
$$

By lemma 3.35, $\lfloor\llbracket\widehat{\tau}_1 \sqcup \widehat{\tau}_2\rrbracket_\Sigma\rfloor = \lfloor\widehat{\tau}_1 \sqcup \widehat{\tau}_2\rfloor = \tau$.

$t = \lambda x : \tau_1.t'$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_{\mathrm{t}} \lambda x : \tau_1.t' : \tau$. The only rule that applies is [U-Abs], therefore $\tau = \tau_1 \to \tau_2$ for some τ_2 and the premise is $\lfloor\widehat{\Gamma}\rfloor, x : \tau_1 \vdash_{\mathrm{t}} t' : \tau_2$.

We have $\widehat{\tau}_1 \,\&\, \beta \rhd \overline{\beta_i :: \kappa_i} = \mathcal{C}([]; \tau_1)$. We define $\widehat{\Gamma}' = \widehat{\Gamma}, x : \widehat{\tau}_1 \,\&\, \beta$. By lemma 3.20 we have $\lfloor\widehat{\tau}_1\rfloor = \tau_1$. Then $\lfloor\widehat{\Gamma}'\rfloor = \lfloor\widehat{\Gamma}\rfloor, x : \lfloor\widehat{\tau}_1\rfloor = \lfloor\widehat{\Gamma}\rfloor, x : \tau_1$. By a reasoning similar to the corresponding soundness proof, we know $\widehat{\Gamma}'$ is well-formed under $\Sigma, \overline{\beta_i :: \kappa_i}$.

We can now apply the induction hypothesis, so we have $\mathcal{R}(\widehat{\Gamma}'; \Sigma, \overline{\beta_i :: \kappa_i}; t') = \widehat{t'} : \widehat{\tau}_2 \,\&\, \xi_2$ such that $\lfloor\widehat{t'}\rfloor = t'$ and $\lfloor\widehat{\tau}_2\rfloor = \tau_2$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; \lambda x : \tau_1.t') = \Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \,\&\, \beta.\widehat{t'} : \forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \,\&\, \bot$ with $\lfloor\Lambda\overline{\beta_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \,\&\, \beta.\widehat{t'}\rfloor = \lfloor\lambda x : \widehat{\tau}_1 \,\&\, \beta.\widehat{t'}\rfloor = \lambda x : \lfloor\widehat{\tau}_1\rfloor.\lfloor\widehat{t'}\rfloor = \lambda x : \tau_1.t'$ and $\lfloor\forall\overline{\beta_i :: \kappa_i}.\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle\rfloor = \lfloor\widehat{\tau}_1\langle\beta\rangle \to \widehat{\tau}_2\langle\xi_2\rangle\rfloor = \lfloor\widehat{\tau}_1\rfloor \to \lfloor\widehat{\tau}_2\rfloor = \tau_1 \to \tau_2$.

$t = t_1\ t_2$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_t t_1\ t_2 : \tau$. Only [U-App] applies, therefore $\lfloor\widehat{\Gamma}\rfloor \vdash_t t_1 : \tau_2 \to \tau$ and $\lfloor\widehat{\Gamma}\rfloor \vdash_t t_2 : \tau_2$ for some τ_2.

By induction, $\mathcal{R}(\lfloor\widehat{\Gamma}\rfloor; \Sigma; t_1) = \widehat{t_1} : \widehat{\tau}_1 \,\&\, \xi_1$ and $\mathcal{R}(\lfloor\widehat{\Gamma}\rfloor; \Sigma; t_2) = \widehat{t_2} : \widehat{\tau}_2 \,\&\, \xi_2$ such that $\lfloor\widehat{t_1}\rfloor = t_1$, $\lfloor\widehat{\tau}_1\rfloor = \tau_2 \to \tau$, $\lfloor\widehat{t_2}\rfloor = t_2$ and $\lfloor\widehat{\tau}_2\rfloor = \tau_2$. By theorem 3.36, $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are conservative.

As $\widehat{\tau}_1$ is a function type and $\mathcal{I}$ is structurally recursive, the call always succeeds, yielding $\widehat{\tau}_2'\langle\beta\rangle \to \widehat{\tau}\langle\xi\rangle \triangleright \overline{\beta_i} = \mathcal{I}(\widehat{\tau}_1)$. The call to $\mathcal{M}$ match also succeeds by lemma 3.34, resulting in $\theta = [\beta \mapsto \xi_2] \circ \mathcal{M}([\,]; \widehat{\tau}_2'; \widehat{\tau}_2)$.

Since $\lfloor\widehat{\tau}_1\rfloor = \tau_2 \to \tau$, we must also have $\lfloor\widehat{\tau}_2'\langle\xi_2'\rangle \to \widehat{\tau}\langle\xi\rangle\rfloor = \tau_2 \to \tau$ by definition of $\mathcal{I}$. That implies $\lfloor\widehat{\tau}\rfloor = \tau$.

Then, $\mathcal{R}(\widehat{\Gamma}; \Sigma; t_1\ t_2) = \widehat{t_1}\ \langle\overline{\theta\ \beta_i}\rangle\ \widehat{t_2} : [\![\theta\widehat{\tau}]\!]_\Sigma \,\&\, [\![\xi_1 \sqcup \theta\ \xi]\!]_\Sigma$. We have $\lfloor\widehat{t_1}\ \langle\overline{\theta\ \beta_i}\rangle\ \widehat{t_2}\rfloor = \lfloor\widehat{t_1}\rfloor\ \lfloor\widehat{t_2}\rfloor = t_1\ t_2$ and $\lfloor[\![\theta\widehat{\tau}]\!]_\Sigma\rfloor = \lfloor\theta\widehat{\tau}\rfloor = \lfloor\widehat{\tau}\rfloor = \tau$. The second step is justified by the fact that substitutions do not change the structure of types.

$t = \mu x : \tau.t'$ By assumption, $\lfloor\widehat{\Gamma}\rfloor \vdash_t \mu x : \tau.t' : \tau$. The only matching rule is [U-Fix], therefore $\lfloor\widehat{\Gamma}\rfloor, x : \tau \vdash_t t' : \tau$ must hold as its premise.

We define the sequences $(\widehat{t'}_i)_{i\in\mathbb{N}^+}$ and $(\widehat{\tau}_i \,\&\, \xi_i)_{i\in\mathbb{N}}$ by

$$\begin{aligned} \widehat{\tau}_0 \,\&\, \xi_0 &:= \bot_\tau \,\&\, \bot_\star \\ \widehat{t'}_{i+1} : \widehat{\tau}_{i+1} \,\&\, \xi_{i+1} &:= \mathcal{R}(\widehat{\Gamma}, x : \widehat{\tau}_i \,\&\, \xi_i; \Sigma; t') \end{aligned}$$

The well-definedness of this definition follows from the induction hypothesis. Moreover, $\lfloor\widehat{\tau}_i\rfloor = \tau$ follows from lemma 3.25 for $i = 0$ and from the induction hypothesis for $i > 0$.

Claim: For all $i \in \mathbb{N}$ we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_i \leqslant \widehat{\tau}_{i+1}$ and $\Sigma \vdash_{\text{sub}} \xi_i \sqsubseteq \xi_{i+1}$.

Proof: By induction on i.

$i = 0$ We have $\widehat{\tau}_0 = \bot_\tau$ and $\xi_0 = \bot_\star$. By lemma 3.26, $\Sigma \vdash_{\text{sub}} \bot_\tau \leqslant \widehat{\tau}_1$ and by lemma 3.23 $\Sigma \vdash_{\text{sub}} \bot_\star \sqsubseteq \xi_1$.

$i = i' + 1$ By induction, we have $\Sigma \vdash_{\text{sub}} \widehat{\tau}_{i'} \leqslant \widehat{\tau}_{i'+1}$ and $\Sigma \vdash_{\text{sub}} \xi_{i'} \sqsubseteq \xi_{i'+1}$. Since we have $\mathcal{R}(\widehat{\Gamma}, x{:}\widehat{\tau}_{i'}\&\xi_{i'}; \Sigma; t') = \widehat{t'}_{i'+1}{:}\widehat{\tau}_{i'+1}\&\xi_{i'+1}$ and $\mathcal{R}(\widehat{\Gamma}, x{:}\widehat{\tau}_{i'+1}\&\xi_{i'+1}; \Sigma; t') = \widehat{t'}_{i'+2} : \widehat{\tau}_{i'+2} \,\&\, \xi_{i'+2}$, we can apply lemma 3.47, resulting in $\Sigma \vdash_{\text{sub}} \widehat{\tau}_{i'+1} \leqslant \widehat{\tau}_{i'+2}$ and $\Sigma \vdash_{\text{sub}} \xi_{i'+1} \sqsubseteq \xi_{i'+2}$.

■

This implies $[\widehat{\tau}_i \,\&\, \xi_i]_\Sigma \sqsubseteq_\Sigma [\widehat{\tau}_{i+1} \,\&\, \xi_{i+1}]_\Sigma$ for all i by definition. By lemma 3.39, there is only a finite number of such equivalence classes. Hence, there is an index j such that $[\widehat{\tau}_i \,\&\, \xi_i]_\Sigma = [\widehat{\tau}_{i+1} \,\&\, \xi_{i+1}]_\Sigma$ for all $i \geqslant j$. But then $\widehat{\tau}_i \,\&\, \xi_i \equiv_\Sigma \widehat{\tau}_{i+1} \,\&\, \xi_{i+1}$ for all $i \geqslant j$. In particular, the termination condition of the fixpoint iteration is fulfilled after the j-th iteration.

We can conclude $\mathcal{R}(\widehat{\Gamma}; \Sigma; t) = \mu x : \widehat{\tau}_{j+1} \,\&\, \xi_{j+1}.\widehat{t'}_{j+1} : \widehat{\tau}_{j+1} \,\&\, \xi_{j+1}$. Clearly,

$$\lfloor\mu x : \widehat{\tau}_{j+1} \,\&\, \xi_{j+1}.\widehat{t'}_{j+1}\rfloor = \mu x : \lfloor\widehat{\tau}_{j+1}\rfloor.\lfloor\widehat{t'}_{j+1}\rfloor = \mu x : \tau.t'.$$

□