KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Coherent Implicit Resolution

Koar Marntirosian

Supervisor:
Prof. dr. ir. Tom Schrijvers

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

August 2021

Coherent Implicit Resolution

Koar MARNTIROSIAN

Examination committee:
Prof. dr. ir. Omer Van der Biest, chair
Prof. dr. ir. Tom Schrijvers, supervisor
Prof. dr. ir. Frank Piessens
Prof. dr. Tias Guns
Prof. dr. Bruno C. d. S. Oliveira
(University of Hong Kong)
Prof. dr. Dariusz Biernacki
(University of Wrocław)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

August 2021

© 2021 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Koar Marntirosian, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

Acknowldegments I would like to thank my supervisor, Tom Schrijvers, for giving me the opportunity to start my Ph.D. studies and for his valuable support on the way to completing them. Working with you and within your team has shaped me as a researcher and has taught me a lot, both as a researcher and as a person.

I thank the members of my examination committee for their insightful remarks and comments on this thesis.

Many thanks go to my colleagues Alexander Vanderbroucke, Amr Hany Saleh, Birthe Van den Berg, César Santos, Emekcan Aras, George Karachalias, Gert-Jan Bottu, Ilias Tsigkenopoulos, Koen Pauwels, Ruben Pieters, Stefanos Peros, Stelios Tsampas, Steven Keuchel and Zubair Rafique for keeping the atmosphere cheerful, even in struggling times, and for the welcoming office environment we created together.

I cannot think of my doctoral studies in isolation from life. Quin, I thank you not only for your support throughout my years abroad, but also for raising me the way you did. Paul, our discussions on topics around programming have only been motivating. We would have so much more to say.

Lastly, I would like to thank my close friends Ani Deokmejian and Zeinep Achmet, whose refreshing visits helped recharging batteries. Albi, life in Leuven would have just been much harder without you. Thank you for being there for me. Giorgo, I don't know how to express my gratitude. You have been a great influence on me and, even though not planned, you did connect me to Leuven.

Abstract

In programming languages, type-directed resolution is a mechanism that automatically composes expressions using type information. On the one hand, resolution enables a variety of powerful implicit programming features, such as type classes in Haskell, implicits in Scala and traits in Rust. These features allow expressions to be omitted by the programmer and implicitly inferred through resolution, making programs more compact and easier to read. On the other hand, it may introduce semantical ambiguity in the language and break equational reasoning over programs of the language, if not carefully designed.

Fortunately, posing appropriate restrictions on the resolution mechanism can save it from unpredictable and counterintuitive behaviour. These restrictions vary with the design of the mechanism, resulting to different, but correct, implicit programming features. The correctness of resolution is formally stated via two properties: coherence, which ensures semantical unambiguity, and stability, which ensures equational reasoning.

This thesis studies three calculi, each supporting a different implicit programming feature. The first calculus, $\boldsymbol{\lambda}_{\mathrm{TC}}$, supports type class resolution, as in Haskell or PureScript, and isolates it from other language features. We show that its elaboration-based semantics is coherent with the use of an equivalent two-step elaboration that introduces an intermediate language. The second calculus, Cochis, enables implicit programming with higher-order and locally scoped implicits. We show that Cochis is stable and that its resolution mechanism is deterministic, thus trivially resulting to a coherent semantics. Furthermore, we present a resolution algorithm that is sound and complete with respect to its declarative specification. Lastly, the third calculus, λ_i^{MP}, features intersection types and a standard BCD-style subtyping system, but extended in a way that integrates resolution within subtyping, thus unifying the two features. Implicit programming in this setting enables first-class environments. We present a sound and complete algorithm for the extended subtyping and we formulate a coherence proof for λ_i^{MP} in terms of two conjectured properties. We use logical

relations for the coherence proofs of $\boldsymbol{\lambda}_{\mathrm{TC}}$ and λ_i^{MP}, whose resolution mechanisms result to a non-deterministic semantics.

In summary, this work presents techniques for reasoning about calculi that support implicit programming, a task that becomes harder in the presence of such features, and for delivering those calculi together with a verification of their correct behaviour. Furthermore, it illustrates the points that call for care in the design and in the implementation of type-directed resolution.

Beknopte samenvatting

Type-gestuurde resolutie is een mechanisme binnen programmeertalen dat programmacode automatisch samenstelt met behulp van type-informatie. Enerzijds maakt deze resolutie een verscheidenheid aan krachtige impliciete programmeertaalconcepten mogelijk, zoals type classes in Haskell, implicits in Scala en traits in Rust. Met deze taalconcepten kan de programmeur code weglaten die door resolutie afgeleid wordt, waardoor programma's compacter en makkelijker te lezen worden. Anderzijds kan onzorgvuldig ontworpen resolutie semantische dubbelzinnigheid in de programmeertaal introduceren en redeneertechnieken zoals "equational reasoning" bemoeilijken.

Gelukkig zijn er restricties die opgelegd kunnen worden om het resolutiemechanisme te behoeden voor onvoorspelbaar en contra-intuïtief gedrag. Deze restricties kunnen variëren afhankelijk van het ontwerp van het mechanisme en resulteren in verschillende correcte taalconcepten. De correctheid van resolutie wordt formeel uitgedrukt onder de vorm van twee eigenschapen: coherentie, die semantische eenduidigheid waarborgt, en stabiliteit, die equational reasoning waarborgt.

Deze thesis bestudeert drie calculi die elk een ander impliciet programmeertaalconcept bevatten. De eerste calculus, $\boldsymbol{\lambda}_{\mathrm{TC}}$, ondersteunt resolutie van type classes, zoals in Haskell of Purescript, en isoleert het van andere taalfuncties. We tonen aan dat de semantiek van deze calculus, op basis van elaboratie, coherent is door gebruik te maken van een equivalente tweestapselaboratie via een tussenliggende taal. De tweede calculus, Cochis, ondersteunt programmeren met hogere-orde impliciete uitdrukkingen binnen een lokaal bereik. We tonen aan dat Cochis stabiel is en dat zijn resolutiemechanisme deterministisch is, wat triviaal resulteert in een coherente semantiek. Bovendien stellen we een resolutiealgoritme voor dat correct en volledig is ten opzichte van de declaratieve specificatie van Cochis. Tot slot is de derde calculus, λ_i^{MP}, gebaseerd op intersectietypes en een standaard BCD-stijl subtyperingssysteem. We hebben dit uitgebreid met een nieuw resolutiemechanisme dat geïntegreerd is in de

subtypering, waardoor de twee taalconcepten samenwerken. We stellen een correct en volledig algoritme voor dat de uitgebreide subtypering nagaat en we bewijzen de coherentie van λ_i^{MP} in de veronderstelling dat aan twee niet-bewezen eigenschappen voldaan is. Voor de coherentiebewijzen van $\boldsymbol{\lambda}_{\mathrm{TC}}$ en λ_i^{MP}, die beide door het resolutiemechanismen een niet-deterministische semantiek hebben, maken we gebruik van logische relaties.

Kortom, dit werk stelt technieken voor om te redeneren over calculi die impliciet programmeren ondersteunen, een taak die danig bemoeilijkt wordt door de aanwezigheid van deze taalconcepten, en past ze toe op drie calculi om hun correctheid vast te stellen. Bovendien illustreren we de aandachtspunten in het ontwerp en de implementatie van type-gestuurde resolutie.

Contents

Abstract iii

Beknopte samenvatting v

Contents vii

List of Figures xi

1 Introduction 1
- 1.1 Types, Modularity and Polymorphism . . . 3
- 1.2 More Variants of Polymorphism . . . 5
  - 1.2.1 Ad-hoc Polymorphism . . . 5
  - 1.2.2 Subtype Polymorphism . . . 8
  - 1.2.3 Correctness Properties . . . 9
- 1.3 Thesis Overview . . . 10
  - 1.3.1 Coherence of Type-Class Resolution . . . 10
  - 1.3.2 COCHIS: Stable and Coherent Implicits . . . 11
  - 1.3.3 Resolution as Intersection Subtyping via Modus Ponens 12

2 Background 13
- 2.1 Type Systems and Type Soundness . . . 13
  - 2.1.1 Type Soundness . . . 14
- 2.2 Haskell Type Classes . . . 16
  - 2.2.1 Coherence in Type Classes . . . 18
  - 2.2.2 Stability in Type Classes . . . 20
  - 2.2.3 Global Uniqueness in Type Classes . . . 21
- 2.3 Scala Implicits and Stability . . . 22
  - 2.3.1 Instability in Scala . . . 25
- 2.4 Design Choices for Type-Directed Resolution . . . 26

3 Coherence of Type Class Resolution 29

3.1 Overview . . . 32
3.1.1 Dictionary-Passing Elaboration . . . 32
3.1.2 Nondeterminism and Coherence . . . 34
3.1.3 Contextual Difference . . . 36
3.1.4 Our Approach to Proving Coherence . . . 36
3.2 Source Language $\boldsymbol{\lambda}_{\text{TC}}$ . . . 38
3.3 Target Language $\mathbf{F}_{\{\}}$ . . . 44
3.3.1 Elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$ . . . 44
3.4 Coherence . . . 48
3.4.1 Contextual Equivalence . . . 48
3.4.2 Coherence . . . 49
3.5 Intermediate Language $\mathbf{F}_{\text{D}}$ . . . 50
3.5.1 Elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$ . . . 54
3.5.2 Elaboration from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$ . . . 57
3.5.3 Elaboration Decomposition . . . 58
3.6 Coherence Revisited . . . 58
3.6.1 Coherent Elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$ . . . 59
3.6.2 Deterministic Elaboration from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$ . . . 62
3.7 Discussion of Possible Extensions . . . 64
3.8 Related Work . . . 66
3.9 Scientific Output . . . 67

4 Cochis: Stable and Coherent Implicits 69
4.1 Overview . . . 71
4.1.1 An Overview of Cochis . . . 71
4.1.2 Overlapping Implicits and Stability in Cochis . . . 76
4.2 The Cochis Calculus . . . 77
4.2.1 Syntax . . . 78
4.2.2 Type System . . . 78
4.2.3 Resolution . . . 80
4.2.4 Type-Directed Resolution with Focusing . . . 81
4.2.5 Deterministic and Stable Resolution . . . 83
4.3 Resolution Algorithm . . . 92
4.3.1 Deferred Variable Instantiation . . . 92
4.3.2 Algorithmic Stability Check . . . 94
4.3.3 Scope-Aware Unification . . . 95
4.3.4 Termination of Resolution . . . 97
4.4 Type-Directed Translation to System F . . . 100
4.4.1 Type-Directed Translation . . . 100
4.4.2 Evidence Generation in the Algorithm . . . 104
4.4.3 Dynamic Semantics . . . 104
4.5 Discussion . . . 105
4.5.1 Predicative Instantiation . . . 105

4.5.2 Committed Choice . . . 106
4.5.3 Superclasses . . . 108
4.5.4 Coherence . . . 110
4.6 Related Work . . . 111
4.6.1 Implicit Programming with Local Scoping, but no Overlapping Rules . . . 111
4.6.2 Implicit Programming with Local Scoping and Overlapping Implicits . . . 112
4.6.3 Implicit Programming with Global Scoping . . . 113
4.6.4 Global Uniqueness and Same Instance Guarantee . . . 116
4.6.5 Focused Proof Search . . . 116
4.6.6 Stability . . . 117
4.7 Scientific Output . . . 117

5 Resolution as Intersection Subtyping via Modus Ponens 119
5.1 Introduction . . . 119
5.2 Background . . . 123
5.2.1 Coercive Subtyping, Resolution and Their Computational Interpretation . . . 123
5.2.2 Different Interpretations for Intersection Types in the Literature . . . 126
5.3 Overview . . . 127
5.3.1 The Merge Operator and Disjoint Intersection Types . . . 127
5.3.2 Disjoint Merges, Non-Overlapping Instances and Resolution 128
5.3.3 First-Class Environments . . . 129
5.3.4 Interaction Between Subtyping and Resolution . . . 133
5.3.5 Technical Challenges . . . 133
5.4 The λ_i^{MP} Calculus . . . 134
5.4.1 Declarative Subtyping . . . 136
5.4.2 Elaboration Semantics . . . 137
5.4.3 Target and Elaboration Metatheory . . . 138
5.4.4 Disjointness Conditions . . . 139
5.5 Algorithmic Subtyping . . . 142
5.5.1 The Subtyping Algorithm . . . 143
5.5.2 Algorithm Metatheory . . . 148
5.6 Coherence . . . 150
5.6.1 A Context Dependent Logical Relation . . . 152
5.6.2 Coherence Proof . . . 153
5.7 Related Work . . . 155
5.8 Scientific Output . . . 158

6 Conclusion and Future Work 160
6.1 Coherence of Type Class Resolution . . . 160

6.2 Cochis: Stable and Coherent Implicits 161
6.3 Resolution as Intersection Subtyping with Modus Ponens 161

A System F **163**
A.1 Pure System F 163
A.2 Extensions for $\mathbf{F}_{\{\}}$ 163

B Cochis Sketch of Proofs **165**
B.1 Type Preservation 165
B.2 Auxiliary Lemmas About Non-Determistic Resolution 166
B.3 Soundness of Deterministic Resolution 166
B.4 Deterministic Resolution is Deterministic 167
B.5 Resolution and Typing Stability 169
B.6 Auxiliary Lemmas About the Unification Algorithm 170
B.7 Soundness of the Algorithmic wrt Deterministic Resolution . . 171
B.8 Completeness of the Algorithm wrt Deterministic Resolution . 172

C λ_i^{MP} and λ_c^{MP} Related Sources **175**
C.1 Complete Specification of λ_c^{MP} 175
C.2 Context Typing 177
C.3 Additional Definitions and Meta-functions 178
C.4 Complete Derivation Examples of Typing and Algorithmic Subtyping 179
C.5 The Logical Relation for λ_i^{MP} 180
C.6 Termination of Algorithmic Subtyping 184
C.6.1 The proof 184
C.6.2 Arrow-loops are not a source of non-termination 187
C.6.3 Loop detection for termination of MP-loops 189

Bibliography **191**

List of Figures

3.1 The different calculi with elaborations . . . 37
3.2 $\boldsymbol{\lambda}_{\text{TC}}$ syntax . . . 39
3.3 Closure and unambiguity relations . . . 41
3.4 $\boldsymbol{\lambda}_{\text{TC}}$ typing, selected rules . . . 42
3.5 $\boldsymbol{\lambda}_{\text{TC}}$ instance declaration typing . . . 43
3.6 $\boldsymbol{\lambda}_{\text{TC}}$ constraint entailment . . . 44
3.7 $\mathbf{F}_{\{\}}$ syntax . . . 44
3.8 $\mathbf{F}_{\text{D}}$, selected syntax . . . 50
3.9 $\mathbf{F}_{\text{D}}$ typing, selected rules . . . 51
3.10 $\mathbf{F}_{\text{D}}$ dictionary typing . . . 52
3.11 $\mathbf{F}_{\text{D}}$ environment well-formedness, selected rules . . . 53
3.12 $\mathbf{F}_{\text{D}}$ operational semantics, selected rules . . . 53
3.13 $\boldsymbol{\lambda}_{\text{TC}}$ typing with elaboration to $\mathbf{F}_{\text{D}}$, selected rules . . . 54
3.14 $\boldsymbol{\lambda}_{\text{TC}}$ constraint entailment with elaboration to $\mathbf{F}_{\text{D}}$ . . . 55

4.1 Cochis syntax . . . 78
4.2 Type System and Type-directed Translation to System F . . . 79
4.3 Ambiguous Resolution . . . 80
4.4 Context types and simple types . . . 82
4.5 Focusing Resolution . . . 82
4.6 Unambiguous Context Types . . . 86
4.7 Valid Substitutions . . . 90
4.8 Deterministic Resolution and Translation to System F . . . 91
4.9 Resolution Algorithm . . . 93
4.10 Algorithmic Stability Check . . . 95
4.11 Unification Algorithm . . . 97
4.12 Termination Condition . . . 99

5.1 λ_i^{MP} Syntax . . . 134
5.2 Bidirectional Typing . . . 135

5.3 Declarative Specification of Subtyping 136
5.4 λ_c^{MP} Syntax 137
5.5 λ_c^{MP} Coercion Typing and Operational Semantics (new rules) . 138
5.6 Disjointness conditions 139
5.7 Example subtyping derivations 140
5.8 Algorithmic Subtyping 144
5.9 Visual Representation of the Execution of $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$ 146

A.1 System F Syntax and Typing Rules 164
A.2 $\mathbf{F}_{\{\}}$ syntax, extends Figure A.1 (top) 164
A.3 $\mathbf{F}_{\{\}}$ typing rules, extends Figure A.1 (bottom) 164

B.1 Unambiguous Type Environment 168
B.2 Annotated matching judgment 168
B.3 Auxiliary Definition R 170

C.1 Logical Relations 181
C.2 Dependency graph between algorithmic subtyping modes . . . 184
C.3 Visualization of the algorithmic subtyping process 185
C.4 Definition of intersection membership 186
C.5 Single arrow-loops, formalized 187
C.6 Single mp-loops, formalized 188

Chapter 1

Introduction

Imagine you want to cook a dish you've never made before. Since you have no prior experience, you decide to follow the recipe posted by your favorite chef. Although its instructions are clear to you, recipes usually lack a lot of information. For example, they describe what to do with the ingredients, but usually not where to get them from. This information depends on your household and habits: maybe the tomatoes are in the fridge, maybe you have left them on the balcony for a night. Moreover, the recipe may instruct you to cut several types of food, but does not inform you about which knife is most suitable for each. This information is implicit–you wouldn't attempt to cut meat with a butter knife, would you?

The chef counts on your ability to fill in the blanks, in order to make their recipes short and to the point, and to avoid tiring the hungry reader with too many details. However, a chef risks the quality of your end result, because they cannot be certain that you will *correctly* derive the missing information. If you suddenly realise you are out of wheat flour, who knows you won't replace it with some other ingredient of similar color and texture–say, powdered sugar[1]! Yet, succinctness is preferred over exactness: it makes recipes easier to read. However, if the chef wants to keep their audience engaged by providing short-and-clear instructions in a way that the majority will succeed, then the trade-off between the instructions' precision and the result's quality becomes an important matter.

Recipes are analogous to programs: they consist of instructions to produce a result–the dish–in a finite amount of time. Under this view, cooks are analogous to computers: their task to read and understand the recipe corresponds to

[1] Example taken from real life.

the computer's task to parse and compile the program, and the execution of the recipe (by the cook) corresponds to the execution of a program (by the computer). Naturally, the chef (and, in fact, any recipe author) corresponds to the programmer.

There is, however, a difference. A cooking recipe is intended for human-to-human communication. Thus, recipe authors can rely on the fact that humans use common sense, acquired through experience, to understand imprecise instructions. In contrast, programming is a human-to-computer communication and computers need mathematical precision, a level of exactitude which the human brain often cannot afford to handle. Taking also into account the complexity of modern software, this makes programming a very error-prone task, resulting in buggy software.

Numerous techniques have been proposed to mitigate this problem, ranging from best practices in programming and code organization to software verification. A complementary approach targets programming languages, the main tool of software development. To facilitate software productivity, languages have evolved into more expressive systems that embody processes for protecting programmers from a class of mistakes and for inferring implicit information. Thus, the tasks of error detection and implicit inference are transferred to programming languages by designing them to be *type sound* and to support *implicit programming*. While type soundness prevents basic programming errors, implicit programming can introduce sources of unpredictable behaviour when not designed carefully. Therefore, to further narrow down the cause of bugs in software, it is desired that this feature is well-designed. The programmer should be able to predict how the computer will fill in the implicit information.

The remainder of this chapter introduces and discusses the two aforementioned directions taken in programming language design. The first, discussed in Section 1.1, is the introduction of *types* in programming languages. Types prevent basic coding errors and enable reasoning about a language's type soundness. Despite their practical and meta-theoretical advantages, *simple* types hamper code modularity, a code organization principle by which a single concept should be coded only once within the program and re-used wherever needed. This loss is recovered with the introduction of *polymorphic* types. The second, discussed in Section 1.2, introduces two kinds of *polymorphism*: ad-hoc polymorphism, which is an example of implicit programming, and subtype polymorphism, which enables modularity and implicit programming, but in a restricted form. Lastly, Section 1.3 presents an overview of this thesis.

Code examples in the remainder of this chapter use a Haskell-like syntax.

1.1 Types, Modularity and Polymorphism

A basic class of errors in programs are *type errors*. The precise definition of what consitutes a type error depends on the programming language, but this category usually includes two cases: the application of a function on an argument for which the function is not defined, and the application of a non-function on an argument. For example, suppose the function `incr` that expects an integer and increments it by one. In a language that does not support implicit conversions, the expression `incr 'a'` contains a type error. In a sense, it is meaningless and its execution would cause the program to get stuck, or crash, since the system wouldn't know how to apply `incr` on a character.

A highly adopted technique to avoid such frivolous bugs, is the introduction of types in programming languages. The idea of types was introduced by Russell [1908], in his search for the foundations of mathematics, while the incorporation of types in the λ-calculus is attributed to Church [1940]. Types can be viewed as a way to classify expressions and distinguish between meaningful programs, those that are *well-typed* or *well-formed*, and meaningless ones, those that contain type errors. Every well-typed expression is classified by a type, and every ill-typed expression cannot be classified by any type.

With types at hand, the fact that `incr` expects an integer and returns an integer can be captured by assigning the type $\mathsf{Int} \to \mathsf{Int}$ to it, where Int is the type for integers. The compiler now can check whether the type of the argument passed to `incr` agrees with the function's type signature, and produce an executable or a compilation error, accordingly. This task is performed at the *type checking* phase of compilation. As a result, the programmer can be assured that a program that succeeds compilation is free of type errors. On the contrary, in a non-typed setting the compiler produces an executable straight-away and potential type errors manifest at run-time, requiring a software verification technique to detect them. Furthermore, types allow language designers to reason about the type soundness of a programming language, discussed in Section 2.1.

Simple Types The simplest type structure in a language is that of the simply typed λ-calculus (STLC) [Church 1940]. In STLC, a type can either be a *base type*, such as Int for integers and Bool for booleans, or a *function type* $\tau_1 \to \tau_2$, which is compound. A function of type $\tau_1 \to \tau_2$ expects an argument of type τ_1 and returns an expression of type τ_2. Because of their notation, function types are sometimes also called *arrow types*.

Along with the benefits that simple types bring in programming and language metatheory, they have the side effect of making programs larger, introducing a

burden on the programmer in two ways. Firstly, function parameters must now come with a type annotation and function declarations with a type signature. For example, in the untyped λ-calculus, the `incr` function is represented as an anonymous function by the term $\lambda x.\, x + 1$, while in STLC it is represented as $\lambda x : \mathsf{Int}.\, x + 1$. The Int annotation in the λ binder specifies that the parameter x ranges over integers and the programmer is required to write it[2]. Secondly and most importantly, functions that in the untyped λ-calculus can be re-used, in STLC need to be separated in different definitions depending on their type. As a simple example, consider the `id` identity function which expects an argument and just returns it, as shown next.

```
id x = x
```

This function can be safely applied to an argument of any type and can be passed as an argument to any higher-order function that expects another function whose result is of the same type as its argument's type. In STLC, unfortunately, one would need to make copies of this function per type:

```
idInt x :: Int -> Int
idInt x = x

idBool :: Bool -> Bool
idBool x = x
```

This not only makes programs larger, but also defies code reuse.

Polymorphic Types System F is a more advanced calculus that recovers the form of modularity lost in STLC, while preserving the advantages of a typing system. It is an extension of STLC, developed by Reynolds [1974]. In System F, types are extended with type variables, here denoted with the letter a, and polymorphic types $\forall a.\tau$ which abstract over types via the $\forall$ binder. At the term level, type abstractions are possible via the Λ binder. For example, in this calculus the identity function is written as $\Lambda a.\lambda x : a.\, x$ and has the polymorphic type $\forall a.a \to a$. To apply the polymorphic identity function on an argument, the programmer needs to specialize the abstract type a with the type of the provided argument. This is done through a *type application*–providing a type as argument to the type abstraction–before providing an expression as argument to the term abstraction. For example, to use it over integers, we write $(\Lambda a.\lambda x : a.\, x)\,\mathsf{Int}\,1$.

[2]In the simply typed setting, a type inference algorithm can relieve the programmer from writing type signatures, but in more expressive type systems this technique either does not entirely eliminate type annotations or requires sacrifices [Pierce 2002, Chapters 22 and 23].

Although the syntax of System F requires the programmer to deal with types[3] to a greater extent than STLC does, its type system allows a single definition of a program fragment that acts the same regardless of its type. This kind of polymorphism is called *parametric polymorphism* (or *generics*, in the object-oriented terminology), since terms can have types as parameters.

In this style of polymorphism, type signatures and type applications play no role in the behaviour of a program at run-time. They are useful merely for type checking and to ensure type soundness in the meta-theory of the language. The next section introduces two styles of polymorphism which differ from parametric polymorphism in that types can determine the behaviour of a program.

1.2 More Variants of Polymorphism

This section presents ad-hoc polymorphism and subtyping, and introduces two relevant correctness properties.

1.2.1 Ad-hoc Polymorphism

Ad-hoc polymorphism enables *overloading*, a language feature by which an identifier can take on multiple meanings depending on its type. To see an example, suppose the polymorphic function below that checks for membership in a list.

```
isElem :: ∀a. (a -> a -> Bool) -> [a] -> a -> Bool
isElem equals []     elem = False
isElem equals (x:xs) elem = case (equals x elem) of
                              True  -> True
                              False -> isElem equals xs elem
```

This function recursively compares each element of a given list for equality with the `elem` parameter by using the given `equals` function, until one of the list's elements is equal to `elem`, in which case the inspection halts and `isElem` returns `True`, or until all elements have been inspected and none equals `elem`, in which case `isElem` returns `False`. To use `isElem` on values of a concrete type such as `Int` or `Bool`, one would need to provide it with a type argument and an appropriate function for the `equals` parameter. Therefore, the behaviour of `isElem` remains the same for any instantiation of the `a` type

[3] Complete type inference is undecidable for System F. However, it is decidable for fragments of the calculus. Another option is partial reconstruction, which can decrease but not entirely eliminate type annotations in the code. See [Pierce 2002, Chapter 23] for more details.

variable, except for the `equals` function used in its body: because values of different types use different internal representations and equality checking depends on those, the argument provided in place of the `equals` parameter depends on the instantiation of `a`.

This kind of code modularity is an achievement of parametric polymorphism. However, now any function that directly or indirectly calls `isElem` generically, should carry an additional function parameter, to pass it to `isElem` as the `equals` function. Ad-hoc polymorphism allows the programmer to omit this additional argument. This is achieved through a *type-directed resolution* mechanism that implicitly infers an appropriate function for the omitted argument.

Haskell Type Classes A widely known implementation of ad-hoc polymorphism is the type classes feature of Haskell [P. Jones 2003], which enables the declaration of overloaded functions like equality checking.

```
class Eq a where
  (==) :: a -> a -> Bool
```

A type class declaration consists of: a class name, such as `Eq`; a type parameter, such as `a`; and a set of method declarations, such as `(==)`[4]. Each of the methods in the type class declaration should have at least one occurrence of the type parameter `a` in their signature.

Instances implement type classes. For example, `Eq` instances for integers and characters can be defined as follows:

```
instance Eq Int where
  (==) = primEqInt
instance Eq Char where
  (==) = primEqChar
```

The above instances provide the appropriate implementations in terms of primitive functions. Using type classes, the `equals` parameter can be omitted in the definition of `isElem`, significantly shortening the code of this function, as well as of any other function that calls `isElem` generically:

```
isElem :: ∀a. Eq a => [a] -> a -> Bool
isElem []     elem = False
isElem (x:xs) elem = case (x == elem) of
                       True  -> True
                       False -> isElem xs elem
```

[4]Haskell operators (just as `==` for equality) use infix notation, except when enclosed in parentheses, in which case they use prefix notation. When declared with their type signature, operators should be within parentheses.

The type of `isElem` is still polymorphic, but now contains the type class constraint `Eq a`, which signifies that the instantiation of the type variable `a` should have an `Eq` instance[5]. When `isElem` is called on values of a concrete type, Haskell's resolution mechanism will look up an `Eq` instance for that type and, if it finds one, it will replace the `==` operator with the implementation provided in the found instance. For example, when it is called over a list of integers, Haskell will replace the `==` occurrence in the body of `isElem` with `primEqInt`[6]. Because the previous `equals` parameter is now implicitly inferred, type classes (and resolution mechanisms, in general) are an example of implicit programming.

In the above example, simply looking up and selecting the appropriate instance from a set of defined instances–the *implicit environment*–was sufficient for resolution to succeed. However, an important feature of this mechanism, not illustrated above, is its ability to combine instances to produce new ones, when needed. For example, by adding the following instance

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x1, y1) == (x2, y2) = x1 == x2 && y1 == y2
```

an `eq` function for concrete product types, such as `(Int,Char)`, can now be inferred given that `Eq Int` and `Eq Char` instances are found. Note that there is no instance directly defined for `(Int,Char)`.

Parametric polymorphism can be beneficiary for the language's expressivity when combined with ad-hoc polymorphism. However, the one is not a requirement for the other. The calculus presented in Chapter 5 is an example of a simply typed language with resolution.

Apart from Haskell's type classes [Wadler & Blott 1989] (used in this subsection), other implementations of ad-hoc polymorphism include Coq's type classes [Sozeau & Oury 2008], C++'s concepts [Gregor et al. 2006] and JavaGI's generalized interfaces [Wehr et al. 2007].

This style of polymorphism is a variant of a wider class of language mechanisms that infer values by using type information, commonly referred to as *implicit programming*. Other variants include Scala's implicits [Odersky 2010], Agda's instance arguments [Devriese & Piessens 2011], Rust's traits [Mozilla Research 2017], and OCaml's modular implicits [White et al. 2015]. Implicit programming can also be viewed as a form of (type-directed) program synthesis [Manna &

[5]Note that, in Haskell, the single arrow (`->`) denotes a function type constructor, whereas the double arrow (`=>`) denotes a type with type class constraints (on the left of the arrow). In the `isElem` function, for example, `Eq a` are the constraints.

[6]Haskell will also take care of the infix vs. prefix notational difference between the `==` operator and the function `primEqInt`.

Waldinger 1980]. The programming is said to be *implicit*, because expressions (e.g., function arguments) can be omitted by the programmer. Instead, the necessary values are provided automatically via a type-directed resolution process. These implicit values are either fetched by type from the current (implicit) environment or constructed by type-directed rules.

1.2.2 Subtype Polymorphism

Subtyping is yet another way to support code modularity and re-use. It is most notably employed in object-oriented languages, but there are many other uses of subtyping in the literature [Barendregt et al. 1983; Chen 2003; Gay & Hole 2005; Pierce & Sangiorgi 1996].

While in STLC or System F, every well-formed expression can only have a unique type, subtype polymorphism often relies on a revocation of this condition and automatically converts the type of a value into another compatible type, when necessary.

For example, with subtyping, classes `Bird`, `Snake` and `Human` can be classified under the conceptually more general class `Animal`, where the latter provides some basic properties that apply to all animals, such as having an age and a gender, while the former classes inherit the properties of the `Animal` class and provide additional functionality that is specific to the corresponding categories of animals. We say that `Bird` is a *subclass* of `Animal` or, vice-versa, that `Animal` is a *superclass* of `Bird`. With this class hierarchy, a function that expects an argument of the `Animal` class can still be provided with a `Bird`, a `Snake` or a `Human` object. The system will automatically convert the provided argument to an `Animal` object, as long as its original class is a subclass of `Animal`.

The class inheritance mechanism found in Java is an example of *nominal* subtyping, where the user specifies hierarchies between identifiers of classes or interfaces. In *structural* subtyping, on the other hand, the subtyping relation between two classes or two types depends on their 'shape'. Chapter 5 uses the latter variation in combination with intersection types [Barendregt et al. 1983; Reynolds 1991]. This variation of subtyping can model multiple inheritance and provide it with formal foundations [Pierce 1991]. It has also been used to enable finitary function overloading [Castagna et al. 1992], bounded polymorphism [Pierce 1997] and evaluation-order polymorphism [Dunfield 2015].

For a simple illustration on its usage for overloading, consider addition, which is an operation defined on numeric values. There are several types for representing numerals and, just as equality checking, addition too depends on their internal representation. Thus, different implementations of addition are used for

numerals of different types. Assuming pre-defined functions `addInt` for adding `Int`s, `addDouble` for `Double`s and `addFloat` for `Float`s, with intersection types and a *merge operator* (`,,`) we can define an overloaded function `add` as

```
add :: (Int -> Int -> Int) & (Double -> Double -> Double)
       & (Float -> Float -> Float)
add = addInt ,, addDouble ,, addFloat
```

The type of `add` is the intersection of the three arrow types shown. Depending on the type of its two provided arguments, the `add` function will behave as `addInt`, `addDouble` or as `addFloat`. The automatic selection of the appropriate implementation is performed by the subtyping mechanism.

Though both ad-hoc and subtype polymorphism enable overloading, the latter supports a more restricted form of this feature. Just as resolution, subtyping is able to select an appropriate implementation from a given set of implementations, but has no counterpart for synthesizing new implementations as illustrated in the `Eq` type classes example for product types.

1.2.3 Correctness Properties

Designing a correct resolution mechanism–one that infers the intended implementation to replace an overloaded function name–is a task that needs care and entails a trade-off between the mechanism's capabilities and the simplicity of its design (further discussed in Section 2.4).

Coherence A natural enemy that can introduce unpredictable behaviour in implicit programming is *semantic ambiguity*, which may arise when resolution can infer more than one implementations for an overloaded identifier, thus resulting in a non-deterministic semantics for the program. A relevant correctness property is *coherence* [Reynolds 1991], which ensures that the various semantics of a program are equivalent from a user's point of view, all causing the same program behaviour. In the words of Reynolds [1991]:

> *"When a programming language has a sufficiently rich type structure, there can be more than one proof of the same typing judgment; potentially this can lead to semantic ambiguity since the semantics of a typed language is a function of such proofs. When no such ambiguity arises, we say that the language is coherent."*

A coherent language allows its implementor to freely adopt any choice strategy on the language's semantics, without compromising its correct behaviour, as long as that choice is permitted by the language's specification.

Stability Another relevant correctness property is *stability*, when implicit programming is combined with parametric polymorphism. The inference of an implicit value can be affected in different ways when instantiation of a polymorphic type occurs before or after resolution. As a result, seemingly harmless inlinings of code may cause different semantics, contributing to the unpredictability of a program's behaviour. Stability is enforced with the detection and prevention of unstable code, adding one more matter that calls for caution in the design of implicit programming.

1.3 Thesis Overview

This thesis studies implicit programming. In particular, it contributes to the exploration of the expressive power that can be reached through the various choices in the design of type-directed resolution, while emphasising the relevant correctness properties. This goal is realised through the study of $\boldsymbol{\lambda}_{\text{TC}}$, Cochis and λ_i^{MP}, three example calculi where each supports its own variation of type-directed resolution, as briefly explained in the next subsections.

The main part of this thesis consists of Chapters 3, 4 and 5, dedicated in the three example calculi. My contributions are mentioned in the end of each chapter. Chapter 2 introduces background knowledge necessary for the meta-theoretical study of the calculi and their resolution mechanisms in the next chapters. Chapter 6 concludes this thesis. It briefly revisits the goals of this thesis, it summarises the results shown in its main chapters, and presents interesting directions for future research.

1.3.1 Coherence of Type-Class Resolution

Type-class resolution is an example of implicit programming marked by two important design choices: (a) the second-class nature of type classes and instances, and (b) the non-overlapping instances restriction. Expressions are first-class citizens in functional programming, just as objects are in object-oriented programming. Yet, despite the fact that type-class instances contain expressions, this design of ad-hoc polymorphism allows declarations of type classes and their instances only at the top-level of a program's code. Thus,

type classes and instances are static and they have a global scope–they can be used for resolution from any point of the program. Furthermore, they cannot be manipulated (for example, produced or excluded) at run-time, giving them a second-class nature. The non-overlapping instances restriction imposes that every type class has at most one instance declaration per type, thus eliminating a source of ambiguity when an instance needs to be resolved for a type.

Chapter 3 proves coherence for $\boldsymbol{\lambda}_{\text{TC}}$, a Hindley-Milner style calculus extended with type classes and superclasses, devised to study type class resolution in isolation from other language features.

The material of Chapter 3 is drawn from the following publication:

> Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. Proc. ACM Program. Lang. 3, ICFP, Article 91 (August 2019), 28 pages. DOI: `https://doi.org/10.1145/3341695`

Chapter 3 simplifies parts of the coherence proof that are unnecessarily complicated in the original publication[7]. The changes are reflected specifically in the metatheory of Sections 3.4 and 3.6, which discuss coherence of $\boldsymbol{\lambda}_{\text{TC}}$.

1.3.2 Cochis: Stable and Coherent Implicits

Chapter 4 presents Cochis: the Calculus Of CoHerent ImplicitS. Cochis extends System F with an implicit programming mechanism that is more flexible than type class resolution in two main ways. Firstly, implicits in Cochis support local scoping and may overlap, as opposed to the global and non-overlapping instances of type classes. Secondly, unlike the second-class nature of instances in type class resolution, Cochis implicits are expressions and enjoy first-class citizenship. This, in turn, allows higher-order implicits.

In contrast to most previous work that supports these features, the calculus is shown to be not only type sound, but also stable and coherent. In addition to the meta-theoretical study of the calculus, Chapter 4 presents a resolution algorithm that agrees with its declarative specification, and studies its correctness properties.

Chapter 4 draws material from the following publication:

[7] I would like to thank Dariusz Biernacki for his insights, which led to the simplification of the proof.

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. Journal of Functional Programming, 29, E3. DOI: https://doi.org/10.1017/S0956796818000242

1.3.3 Resolution as Intersection Subtyping via Modus Ponens

Driven by the similarities between subtyping and resolution, while also ackgnowledging their differences, Chapter 5 presents a design for unifying the two features. λ_i^{MP}, the calculus used as a case study, is based on NeColus [Bi et al. 2018], a simply typed λ-calculus that features disjoint intersection types [Oliveira et al. 2016] and a BCD-style subtyping system [Barendregt et al. 1983]. To close the gap between the two features, the subtyping relation of NeColus is extended with a rule that resembles *modus ponens* from Logic. Modus ponens is the logical rule that models the intuitive and straight-forward line of thought in the following example derivation.

If it rains, the grass is wet. It rains. Therefore, the grass is wet.

It is shown that this formulation of subtyping subsumes forms of resolution and enables first-class implicit environments, a functionality that cannot be supported by existing approaches for implicit programming.

The calculus is normalizing and type sound, while its coherence property is conjectured and partially proven. On top of the specification's meta-theory, Chapter 5 presents a subtyping algorithm that follows its declarative specification, and studies its correctness properties.

Chapter 5 draws material from the following publication:

Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as intersection subtyping via Modus Ponens. Proc. ACM Program. Lang. 4, OOPSLA, Article 206 (November 2020), 30 pages. DOI: https://doi.org/10.1145/3428274

Chapter 2

Background

This chapter introduces background knowledge necessary for the next chapters. Type class resolution, of which Haskell type classes are the most prominent real-world implementation, is the main focus of Chapter 3. In addition, Haskell's type classes and Scala's implicits, two well established implicit programming features, are used in this thesis as a reference to draw examples from and to compare the resolution mechanisms involved in Chapters 4 and 5. Thus, Section 2.2 discusses Haskell type classes (presented in Section 1.2) in more detail, while Section 2.3 presents Scala implicits and compares the two IP features. Section 2.4 discusses the several choices in the design of implicit programming features. Lastly, Section 2.1 introduces common notions and notation used in the meta-theoretical study of the calculi in the next chapters.

2.1 Type Systems and Type Soundness

This section briefly introduces basic notions and notations that are heavily used in the presention of the calculi in the next chapters and their meta-theory.

Typing Judgment As mentioned in Section 1.1, types give rise to the notion of well-typed expressions. Formally, in the specification of languages with a sufficiently simple type system, well-typed expressions are denoted with the typing judgment

$$\Delta \vdash e : \tau$$

which reads 'expression e has type τ under typing context Δ'. A typing context keeps track of the free variables, along with their types, that occur in the expression e under consideration.

Type System Typing is realized through a set of *typing rules*, which indicate how expressions are typed. A typing rule consists of a number (possibly zero) of judgments as premises and a single typing judgment as conclusion. The premises are separated from the conclusion by a horizontal line, as shown here.

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \cdots \qquad \text{premise}_n}{\text{conclusion}} \ \text{LABEL}$$

The LABEL of a rule is merely a name for it and can alternatively be placed at the left or at the top of the rule. The above notation is common to formally express that if the premises hold then so does the conclusion. Hence, each rule can be construed as an argument for the typing judgment in its conclusion.

As arguments can be combined to prove a statement, typing rules can be combined to construct a typing derivation, where the latter comprises a formal proof of its conclusion. A derivation is constructed by drawing the conclusion of a rule whose premises are the conclusions of other sub-derivations, built-up in a similar manner. Naturally, if a rule has no premises, in which case we may also call it an *axiom*, it forms a trivial derivation that consists of only that axiom.

2.1.1 Type Soundness

The prevention of ill-typed expressions is not sufficient to ensure the correctness of programs. We need to also trust that the programming language is type sound, i.e. a well-typed program will not introduce type errors at run-time.

Operational Semantics One way to model the run-time of a program e is via *small-step operational semantics* [Plotkin 1981], a reduction relation over expressions, denoted as $e \longrightarrow e'$. Successive reductions model the intermediate states of a program when it is run. An irreducible term v is a *value* which represents the meaning of a program e that reduces to it after a finite number of reduction steps. We say that the program or expression e *evaluates* to v and write $e \longrightarrow^* v$, where $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$. For example, integer literals are values and the program `1+1` evaluates to `2`.

With the representation of a program's run-time, we can now state a first and basic correctness condition that entirely rules out type errors: well-typed

expressions of a well-designed language should either be well-typed values themselves or should reduce to an expression of the same type. The well established work of Wright & Felleisen [1992] splits this condition into two statements, formally expressed below as Properties 2.1.1 and 2.1.2.

Property 2.1.1 (Progress). *If* $\Delta \vdash e : \tau$*, then either*

- e *is a value, or*
- $e \longrightarrow e'$*, for some* e'*.*

Property 2.1.2 (Preservation). *If* $\Delta \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\Delta \vdash e' : \tau$*.*

Languages that satisfy the above two properties are *type safe* or *type sound*.

Elaboration Semantics Elaboration is a mechanism by which expressions of a source language translate to expressions of a target language. This way, the meaning of source language expressions is specified in terms of the target language semantics. This technique is standard for languages that support implicit programming, where usually a well-studied calculus that does not support implicits is chosen as a target language. Hence, the elaboration mechanism translates source expressions (which may contain implicits) into target expressions where the (previously) implicit information becomes explicit. Furthermore, reasoning about complex features such as implicits is transferred to reasoning about a well-studied simpler language.

In this thesis, elaboration is modelled by extending the typing judgment of the source language with a component for target expressions. We write

$$\Gamma \vdash E : \sigma \rightsquigarrow e$$

and read 'source expression E has (source) type σ and elaborates to target expression e under typing context Γ'.

To state type soundness for the source language, the progress and preservation properties, above, need to be adapted in the context of elaboration semantics. In this setting, progress holds by construction: the typing judgment of the source language is such that the elaboration of the source expression is generated in parallel with typing that expression. Thus, for a source expression E, it trivially holds that if it is well-typed, formally $\Gamma \vdash E : \sigma$, then there is a target expression e which is the elaborated form of E, formally $\Gamma \vdash E : \sigma \rightsquigarrow e$.

In the context of elaboration semantics, the preservation property states that the elaboration is *type preserving*, i.e. elaborating a well-typed source expression results to a target expression that is also well-typed (in the target type system). In its simple form, type preservation is expressed by means of a function $|\cdot|$

that translates source types and typing contexts into their target counterparts. With this function, the preservation property is formally expressed as follows.

Property 2.1.3 (Type Preserving Elaboration)**.**
If $\Gamma \vdash E : \sigma \rightsquigarrow e$*, then* $|\Gamma| \vdash e : |\sigma|$.

With a type preserving elaboration, if the target language is type sound, so is the source language.

2.2 Haskell Type Classes

As shown in the previous chapter, type classes enable the declaration of overloaded functions. For example, to define a generic sorting function, we start by declaring an `Ord` type class[1], similarly to the `Eq` type class of Section 1.2.

```
class Ord a where
  (<=) :: a -> a -> Bool
```

This type class provides the `(<=)` function for comparison. Then, `Ord` instances for integers, characters, and pairs can be defined as follows:

```
instance Ord Int where
  x <= y  =  primIntLe x y
instance Ord Char where
  x <= y  =  primCharLe x y
instance (Ord a, Ord b) => Ord (a, b) where
  (x,x') <= (y,y') = x <= y && (not (y <= x) || x' <= y')
```

The first two instances provide the implementation of ordering for integers and characters, in terms of primitive functions. The third instance is more interesting, and provides the implementation of lexicographic ordering for pairs. In this case, the ordering instance itself *requires* an ordering instance for both components of the pair. These requirements are resolved by the compiler using the existing set of instances in a process called *resolution*. Using `Ord`, we can now define a generic sorting function

```
  sort :: Ord a => [a] -> [a]
```

that takes a list of elements of an arbitrary type `a` and returns a list of the same type, as long as ordering is supported for type `a`.

[1]The `Ord` type class defined here is a simplification of the predefined `Ord` type class in Haskell, which uses superclasses. Superclasses are discussed later in this section.

Implicit Programming Type classes are an implicit programming mechanism because implementations of type class operations are automatically *computed* from the set of instances during the resolution process. For example, a call to `sort` only type checks if a suitable type class instance can be found. Other than that, the caller does not need to worry about the type class context, as shown in the following interaction with a Haskell interpreter:

```
Prelude > sort [ (3,'a'), (2,'c'), (3,'b') ]
[(2,'c'),(3,'a'),(3,'b')]
```

In this example, the resolution process combines the three `Ord` instances to find a suitable implementation for `Ord (Int,Char)`. The declarations given are sufficient to resolve an infinite number of other instances, such as `Ord (Char,(Int,Int))` and the like.

One Instance Per Type A characteristic of (Haskell) type classes is that only one instance is allowed for a given type. For example, it is forbidden to include the alternative ordering model for pairs

```
instance (Ord a, Ord b) => Ord (a, b) where
  (xa,xb) <= (ya,yb) = xa <= ya && xb <= yb
```

in the same program as the previous instance because the compiler automatically picks the appropriate type class instance based on the type parameter of the type class. If there are two type class instances for the same type, the compiler does not know which of the two to choose.

Superclasses Haskell has supported superclasses since the introduction of type classes. The `Eq` and `Ord` classes, with `Eq` a superclass of `Ord`, constitute a standard example. The following simplified code shows the two classes, together with instances for integers:

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool

instance Eq Int where ...
instance Ord Int where ...
```

The class context `Eq a` in the definition of the `Ord` class specifies that `Eq a` is a superclass of `Ord a`. Superclasses allow the use of methods from the superclasses, even if only the subclass is part of the type class context. For example:

```
p :: Ord a => a -> a -> Bool
p = (==)            -- accepted because Eq is a superclass of Ord
```

Here the signature of function `p` assumes that an instance for `Ord a` is available. In the body of `p`, the method `(==)` of the class `Eq a` is used. This code is accepted in Haskell because `Eq a` is a superclass of `Ord a`.

2.2.1 Coherence in Type Classes

An IP design is *coherent* if any valid program has exactly one meaning (that is, the semantics is not ambiguous). Haskell imposes restrictions to guarantee coherence. For example, Haskell rejects the expression:

```
  show (read "3") == "3"
```

due to *ambiguity* of *type class resolution* [Jones 1992]. The functions `show` and `read` print and parse, respectively, values of any type `a` that implements the classes `Show` and `Read`:

```
class Show a where
  show :: a -> String
class Read a where
  read :: String -> a
```

The program is rejected because there is more than one possible choice for `a`. For example `a` can be instantiated to `Int`, `Float`, or `Char`. Choosing `a=Float` leads to `False`, since showing the float `3` would result in `"3.0"`, while choosing `a=Int` leads to `True`. In other words if this expression was accepted then it could have multiple possible semantics. To ensure coherence instead of making an arbitrary choice, Haskell rejects such programs.

Superclasses, Determinism and Coherence In the presence of superclasses, Haskell's resolution is not deterministic. Consider this variant of `pInt` from the previous section, defined only for integers:

```
pInt :: Int -> Int -> Bool
pInt = (==)
```

Haskell has two ways to resolve `Eq Int` to obtain the implementation of `==` in `pInt`. The first is to get the implementation of `Eq Int` directly from the `Eq Int`

instance. The second is to get an implementation of `Eq Int` as the superclass of `Ord Int`. These two resolution paths result to two distinct elaborations, shown in Subsection 3.1.2 after an overview of the dictionary-passing elaboration is presented. Since they are syntactically different, elaboration with Haskell superclasses is non-deterministic. Nonetheless, the elaboration is still coherent, since both elaborations yield the same semantics (they both execute the code of the one `Eq Int` instance).

Overlapping Instances Advanced features of type classes, such as overlapping instances, require additional restrictions to ensure coherence. The following program illustrates some of the issues:

```
class Trans a where trans :: a -> a
instance Trans a where trans x = x
instance Trans Int where trans x = x + 1
```

This program declares a type class `Trans a` for defining transformations on some type `a`. The first instance provides a default implementation for any type, the identity transformation, while the second instance regards integers only.

An important question here is what happens if we write an expression like `trans 3`. Shall we pick the first or the second instance? Ideally this question should be answered by the language specification. One possible choice for the specification (not the Haskell choice) would be to allow any matching instance to be used, but this choice would lead to incoherence, since `trans 3` could then both evaluate to `3` or `4`. Instead, for overlapping instances, the GHC documentation [GHC 2017a] makes a different choice and declares that the *most specific instance* should be chosen. For the expression `trans 3`, the most specific instance is `Trans Int` and the expression evaluates to `4`.

For the particular program `trans 3`, the Haskell specification manages to avoid incoherence by using the most specific instance, which ensures an unambiguous semantics. Thus, Haskell preserves coherence in the presence of a certain kind of overlapping instances, but there are other problematic overlapping instances that threaten the coherence property.

Incoherent Instances With overlapping instances, it is not always the case that a most specific instance exists. Consider the following type class and instance declarations:

```
class C a b where
  m :: a -> b -> Bool
```

```
instance C Bool a where
  m x y = x
instance C a Bool where
  m x y = y
```

If we write the following program

```
incoherent :: Bool            -- rejected without
incoherent = m True False     -- IncoherentInstances extension
```

then both instances are equally specific. In this case, even with the `OverlappingInstances` extension activated, Haskell rejects the program.

However, Haskell supports the additional extension `IncoherentInstances`, for allowing a more general kind of overlapping instances. Activating this extension permits the `incoherent` definition. The (informal) language specification [GHC 2017a] for `IncoherentInstances` essentially says that in such a situation any matching instance could be picked. Thus, either of the two instances above can be picked, producing different evaluation results for the expression. Thus, as the name indicates, the expression `incoherent` leads to incoherence.

2.2.2 Stability in Type Classes

Another important property, closely related to coherence, is *stability*. Informally, stability ensures that instantiation of type variables does not affect resolution. Unfortunately, overlapping instances threaten this property. Consider the following function, that uses the `trans` method from the type class `Trans` and the two instances declared previously:

```
bad :: a -> a
bad x = trans x    -- unstable definition!
```

Note here that the type of `bad` is `a -> a` instead of `Trans a => a -> a`. In Haskell both signatures can be accepted, but they are not equivalent. With `Trans a => a -> a` resolution is deferred to the call site of bad, allowing the instance of `Trans a` to be selected when `a` has been instantiated to a (potentially) more precise type. By contrast, with `a -> a` resolution is applied eagerly, and an instance must be selected at the definition site. If Haskell were to accept this definition, it would have to implement `trans` using the first instance, since `trans` is applied at the arbitrary type `a`. Unfortunately this would mean that `bad 3` returns `3` but `trans 3` returns `4`, even though `bad` and `trans` are defined to be equal, a nasty impediment to equational reasoning!

For this reason Haskell rejects the program by default. A programmer who really wants such behaviour can enable the `IncoherentInstances` compiler flag, which allows the program to type check.

Note that even though to allow `bad` we need the `IncoherentInstances` extension, which is suggestive of the definition breaking *coherence*, the issue here is not really coherence but rather stability. That is, type instantiation affects type class instantiation. As this example illustrates, instability is actually observable from a compiler implementation like GHC: we can observe that `bad 3` and `trans 3` behave differently. In contrast (in)coherence is not really observable from a compiler implementation: we need a language specification to understand whether there is incoherence or not. More concretely, with a single compiler implementation we can only observe the result of one possible elaboration of a program, but we cannot observe all the other possible elaborations allowed by the specification for that program. Therefore, we cannot observe incoherence.

The `IncoherentInstances` extension is understood to be highly problematic among Haskell programmers, since it can break both stability and coherence. Thus its use is greatly discouraged.

2.2.3 Global Uniqueness in Type Classes

A consequence of having at most one instance of a type class per type in a program is *global uniqueness* of instances [E. Zhang 2014]. That is, at any point in the program type class resolution for a particular type always resolves to the same value. Global uniqueness is a simple way to guarantee coherence, but it offers more than just coherence. The usefulness of this property is illustrated by a library that provides a datatype for sets that is polymorphic in the elements along with a `union` operation:

```
union :: Ord a => Set a -> Set a -> Set a
```

For efficiency reasons the sets are represented by a data structure that orders the elements in a particular way. It is natural to rely on the `Ord` type class to deal with ordering for the particular type `a`. To preserve the correct invariant, it is crucial that the ordering of elements in the set is always the same. The global uniqueness property guarantees this. If two distinct instances of `Ord` could be used in different parts of the program for the same type, then it would be possible to construct within the same program two sets using two different orderings (say ascending and descending order), and then break the ordering invariant by `union`-ing those two sets.

However, although global uniqueness is, in principle, a property that should hold in Haskell programs, Haskell implementations actually violate this property

in various circumstances [nponeccop 2012]. In fact, it is acknowledged that providing a global uniqueness check is quite challenging for Haskell implementations [Marlow 2012].

2.3 Scala Implicits and Stability

Scala implicits [Odersky et al. 2008] are an interesting alternative IP design. Unlike type classes, implicits are locally scoped. Consequently, Scala does not have the global uniqueness property, since different "instances" may exist for the same type in different scopes. Another important difference between implicits and type classes is that implicit parameters can be of any type, and there are no special constructs analogous to type class or instance declarations. Instead, implicits are modeled with ordinary types. They can be abstracted over and do not suffer from the second-class nature of type classes. These features mean that Scala implicits have a wider range of applications than type classes. Unlike Haskell type classes, however, with Scala implicits there is no way to enforce stability.

This thesis regards Scala 2.x implicits. Meanwhile, Scala 3 has been released with significant changes in the design and the syntax of implicits.

Modeling Type Classes with Implicits In Scala there is no special construct for defining the interface of a type class. Instead, we can use regular interfaces to model type class interfaces. Scala models OO interfaces with traits [Scharli et al. 2003]. For example, the type classes `Ord` (without superclasses), `Read` and `Show`, presented in Section 2.2, can be modeled as:

```
trait Ord[T] {def le(x:T,y:T) : Boolean}
trait Show[T] {def show (x : T) : String}
trait Read[T] {def read (x : String) : T}
```

Of course, by declaring traits like this, we still require explicit objects to call the methods on. To be able to use methods in the same way as Haskell type classes, the object (or dictionary) should be passed implicitly. This can be achieved by using Scala's implicits feature:

```
def cmp[A](x:A,y:A)(implicit ordD : Ord[A]) : Boolean = ordD.le(x,y)
```

The Scala definition of `cmp` plays the same role as `(<=)` in Haskell. The type of `cmp` states that `cmp` is parametrized in a type `A`, takes two (explicit) arguments of type `A`, and one implicit parameter (`ordD`). This is similar to a Haskell signature `Ord a => a -> a -> Bool`, except that the implicit argument comes

```
implicit val OrdInt = new Ord[Int] {
   def le(x : Int, y: Int) = primIntLe(x,y)
}

implicit val OrdChar = new Ord[Char] {
  def le(x : Char, y: Char) = primCharLe(x,y)
}

implicit def OrdPair[A : Ord, B : Ord] = new Ord[(A,B)] {
  def le(x : (A,B), y : (A,B)) =
    cmp(fst(x),fst(y)) && (!(cmp(fst(y),fst(x))) || cmp(snd(x),snd(y)))
}
```

Example 2.1: Scala encoding of `Ord` instances

last. Additionally, unlike Haskell, at call sites it is possible to pass the implicit argument explicitly, if desired.

Context Bounds and Queries An alternative way to declare functions with constraints (or implicit arguments) in Scala, is *context bounds*, a simple syntactic sugar built on top of implicit parameters. With context bounds, the `cmp` definition is rewritten into:

```
def cmp[A:Ord](x:A,y:A):Boolean = implicitly[Ord[A]].le(x,y)
```

The notation `A:Ord` is the so-called context bound. This notation enables us to declare a constraint on the type `A`. Namely, the type `A` should be an "instance" of `Ord[A]`. In the background, the Scala compiler rewrites definitions with context bounds into definitions with implicit arguments. In the body of `cmp` an additional mechanism, called an implicit *query*, is now necessary to query the environment for a value of type `Ord[A]`. This query mechanism in Scala is nothing more than a simple function taking an implicit argument. Its (slightly simplified) definition is:

```
def implicitly[T](implicit w:T):T = w
```

The `implicitly` function takes a single *implicit* argument of type `T` and returns that value. Hence operationally it is just the identity function. The key point is that, when used, the implicit argument is filled in automatically by the compiler. In the definition of `cmp`, the expression `implicitly[Ord[A]]` triggers the compiler to look for a value of type `Ord[A]` in the *implicit environment*. The implicit environment collects values that are declared to be implicit, and usable for automatic implicit resolution.

Implicit values, which correspond to type class instances in Haskell, are declared by using the `implicit` keyword. The three definitions in Example 2.1 capture the "instances" for `Ord`. With those definitions, it is now possible to declare functions, such as `sort`, that require `Ord` instances:

```
def sort[A : Ord](x : List[A]) = ...
```

This Scala `sort` function can be used in a similar way to its Haskell counterpart. For example the call `sort (List((3,'a'), (2,'c'), (3,'b')))` is valid and does not require an explicit argument of type `Ord[(Int,Char)]`. Instead, this argument is computed from the implicit definitions for `Ord`.

Wider Range of Applications for Implicits Scala implicits do allow for a wider range of applications than type classes. One example where implicits naturally address a problem that type classes do not address well is the problem of *implicit configurations* [Kiselyov & Shan 2004]. The following example, adapted from Kiselyov and Shan, illustrates this:

```
def add(x : Int, y : Int)(implicit modulus : Int) = (x + y) % modulus
def mul(x : Int, y : Int)(implicit modulus : Int) = (x * y) % modulus
implicit val defMod : Int = 4
def test = add(mul(3,3), mul(5,5))  // returns 2
```

Here the idea is to model *modular arithmetic*, where numbers that differ by multiples of a given modulus are treated as identical. For example $3 * 3 = 1 \pmod 4$ because 9 and 1 differ by a multiple of 4. The code shows the definition of addition and multiplication in modular arithmetic, where in Scala `%` is modulo division. Both addition and multiplication include a third (implicit) parameter, which is the modulus of the division. Although the modulus could be passed explicitly this would be extremely cumbersome. Instead it is desirable that the modulus is passed implicitly. Scala implicits allow this, by simply marking the `modulus` parameter in `add` and `mul` with the `implicit` keyword. The third line shows how to set up an implicit value for the modulus. Adding `implicit` before `val` signals that the value being defined is available for synthesising values of type `Int`. Finally, `test` illustrates how expressions doing modular arithmetic can be defined using the implicit modulus. Because Scala also has local scoping, different modulus values can be used in different scopes.

Several other examples of applications that can be covered by implicits, but are harder to achieve with type classes are found in the existing literature [Odersky et al. 2017, 2008; Oliveira et al. 2012]. In particular, Odersky et al. [2017] introduce *implicit function types*, which are a generalization of the original Scala implicits [Odersky et al. 2008], and demonstrate several interesting use cases for implicits.

```
1  trait A {
2    implicit def id[a] : a => a = x => x
3    def trans[a](x : a)(implicit f: a => a) = f(x)
4  }
5  object B extends A {
6    implicit def succ : Int => Int = x => x + 1
7    def bad[a](x:a) : a = trans[a](x)    // Unstable definition!
8    val v1 = bad[Int](3)                 // Evaluates to 3.
9    // val v2 = trans[Int](3)            // Substituting 'bad' by
10 }                                      // 'trans' is rejected
```

Example 2.2: Nested Scoping with Overlapping Rules in Scala

2.3.1 Instability in Scala

Although Scala allows *nested* local scoping and overlapping implicits, *stability* is not guaranteed. Example 2.2 illustrates the issue briefly, based on the example from Section 2.2.1. Line 2 defines a function `id` with type parameter `a`, which is simply the identity function of type `a => a`[2] . The `implicit` keyword in the declaration specifies that this value may be used to synthesise an implicit argument. Line 3 defines a function `trans` with type parameter `a`, which takes an implicit argument `f` of type `a => a` and returns `f(x)`. Here, the `implicit` keyword specifies that the actual argument should not be given explicitly; instead, an argument of the appropriate type will be synthesised from the available `implicit` declarations.

In the nested scope, line 6 defines function `succ` of type `Int => Int` that takes argument `x` and returns `x+1`. Again, the `implicit` keyword in the declaration specifies that `succ` may be used to synthesise implicit arguments. Line 7 defines a function `bad` with type parameter `a` which takes an argument `x` of type `a` and returns the value of function `trans` applied at type `a` to argument `x`. Line 8 shows that, as in the earlier example and for the same reason, `bad(3)` returns `3`. As with the Haskell example, accepting this definition is an equally nasty impediment to equational reasoning, since performing simple equational reasoning would lead to a different result. However, unlike in Haskell, it is the intended behaviour: it is enabled by default and cannot be disabled. Interestingly, the expression in line 9, accepted in Haskell, is actually rejected in Scala. Note that the expression in line 9 is simply the unfolding of the expression in line 8 (which is accepted in Scala). In the expression in line 9, the Scala compiler does detect two possible instances for `Int => Int`, but does not select the most specific one. In this case, the call in line 9 is considered

[2]In Scala, the `a => b` notation represents a function type.

ambiguous because Scala accounts for other factors when deciding whether there is ambiguity [Odersky et al. 2017, 2008]. Rejecting line 9 has another unfortunate effect. Not only is the semantics not preserved under unfolding, but typing is not either: i.e. going from line 8 to line 9 using a simple unfolding step makes the program ill-typed! Clearly, satisfying desirable properties such as stability and type preservation is a subtle matter in the presence of implicits and deserves careful study.

2.4 Design Choices for Type-Directed Resolution

Programming language design is usually guided by two, often conflicting, goals: *flexibility* and *ease of reasoning*. Many programming languages aim at providing powerful, flexible language constructs that allow programmers to achieve reuse, and develop programs rapidly and concisely. Other programming languages aim at easy reasoning about programs, as well as at avoiding programming pitfalls. Often the two goals are at odds with each other, since highly flexible programming mechanisms make reasoning harder. Arguably the art of programming language design is to reconcile both goals.

Currently there are two main schools of thought regarding the design of implicit programming mechanisms. Haskell's type classes [Wadler & Blott 1989] embody the first school of thought, which is guided by the *ease of reasoning* qualities of pure functional languages, and the *predictability* of programs. To ensure these goals the semantics of the language should satisfy a number of properties. One of those properties is *coherence* [Reynolds 1991], though in fact, Haskell type classes are designed with an even stronger property, the so-called *global uniqueness* of instances [E. Zhang 2014]. Global uniqueness ensures that *at any point in a program, and independently of the context* the type-directed resolution process always returns the same value for the same resolved type. This is a consequence of Haskell having a restriction of at most one instance of a type class per type in a program.

While both coherence and global uniqueness of instances are preserved in Haskell, this comes at a cost. Since the first implementations of type classes, Haskell imposes several restrictions to guarantee those properties. Various past work has indicated limitations of type classes [Camarão & Figueiredo 1999; Dijkstra & Swierstra 2005; Dreyer et al. 2007; Garcia et al. 2007; Kahl & Scheffczyk 2001; Morris & Jones 2010; Odersky et al. 2008; Oliveira et al. 2012]. In particular, type classes allow at most one instance per type (or severely restrict overlapping instances) to exist in a program. Thus, all instances must be visible globally and local scoping of instances is disallowed, a design that goes against modularity.

Other restrictions of type classes are that they are not first-class values and that the type-directed rules cannot be higher-order [Oliveira et al. 2012].

Advanced features of type classes, such as overlapping instances [GHC 2017a], also pose severe problems, since they go against the principle of one instance per type. One issue is that "*when more specific overlapping instances are added, the proofs of some predicates will change to use the new instances*" [Morris & Jones 2010]. In essence, special care (via restrictions) is needed to preserve coherence in the presence of overlapping instances.

Another important property that breaks in the presence of overlapping instances (if special care is not taken) is the so-called *stability* of type substitutions. The issue is that the behaviour of resolution for an expression e can change if e gets a more specific type, leading to a different evaluation result. This is problematic because seemingly harmless inlinings will actually have a different semantics before and after the inlining. Because of this problem, the design of Haskell type classes significantly restricts the set of valid overlapping instances to ensure that stability holds, and the meaning of an expression does not change simply due to a more specific type. In other words, resolution should resolve implicit values using the same rules before and after instantiation of type variables.

As an orthogonal remark, in the Haskell community, the term coherence is often colloquially used to encompass several different properties, including global uniqueness, stability and the original coherence definition by Reynolds [1991]. In the context of an elaboration semantics (which is the standard semantics style for IP mechanisms), the definition of coherence that follows from Reynolds' one is that *all possible elaborations for a program possess the same meaning.* However such a definition of coherence is narrower than the colloquial term usually used in the Haskell community. It is important to note that the three properties are distinct. For example global uniqueness implies coherence, but not the converse. Furthermore it is also possible to have coherence, but not stability. In this thesis we will use coherence in Reynolds' sense, and be precise about the different properties under discussion.

An alternative school of thought in the design of IP mechanisms favours *flexibility.* For example, Scala's implicits and Agda's instance arguments do not impose all of the type class restrictions. Scala supports local scoping of instances, which allows distinct instances to exist for the same type in different scopes of the same program. Scala also allows a powerful form of overlapping implicits [Odersky et al. 2008]. The essence of this style of implicit programming is modelled by the *implicit calculus* [Oliveira et al. 2012] or the more recent SI calculus [Odersky et al. 2017]. The implicit calculus supports a number of features that are not supported by type classes. Besides local scoping, in the implicit calculus *any type* can have an implicit value. In contrast, Haskell's type class model only allows

instances of classes (which can be viewed as a special kind of record) to be passed implicitly. Finally the implicit calculus supports higher-order instances/rules: that is rules whose requirements can themselves be other rules. The implicit calculus has been shown to be type-safe, and it also ensures coherence, but it lacks stability. The SI calculus lacks both coherence and stability, but the authors present a simply-typed subset that is coherent[3]. Unfortunately, while both the implicit/SI calculus and the various existing language mechanisms embody flexibility, the lack of important properties such as stability makes reasoning about the semantics of programs harder, and can prevent refactorings and compiler optimizations such as inlining.

The design of IP mechanisms has led to heated debate [Hulley 2009; Kmett 2015; E. Zhang 2014] about the pros and cons of each school of thought: ease of reasoning versus flexibility. Proponents of the Haskell school of thought argue that coherence, stability and uniqueness of instances are extremely desirable, and flexibility should not come at the cost of those properties. Proponents of flexible IP mechanisms argue that flexibility is more important, and that uniqueness of instances goes against modularity. As far as we are aware there are no current designs that support local scoping, overlapping instances and first-class and higher-order rules, while at the same time ensuring both coherence and stability.

[3]It makes no sense to talk about stability of type substitutions in a simply typed calculus, since this is a property that is only relevant for polymorphic calculi.

Chapter 3

Coherence of Type Class Resolution

Type classes were initially introduced in Haskell [P. Jones 2003] by Wadler & Blott [1989] to make ad-hoc overloading less ad hoc, and they have since become one of Haskell's core abstraction features. Moreover, their resounding success has spread far beyond Haskell: several languages have adopted them (e.g., Mercury [Henderson et al. 1996], Coq [Sozeau & Oury 2008], PureScript [P. Freeman 2017], Lean [Moura et al. 2015]), and they have inspired various alternative language features (e.g., Scala's implicits [Odersky et al. 2017, 2008], Rust's traits [Mozilla Research 2017], C++'s concepts [Gregor et al. 2006], Agda's instance arguments [Devriese & Piessens 2011]).

Type classes have also received a lot of attention from researchers with many proposals for extensions and improvements, including functional dependencies [Jones 2000], associated types [Chakravarty, Keller, & P. Jones 2005], quantified constraints [Bottu et al. 2017] among other extensions.

Given the extensive attention that type classes have received, it may be surprising that the metatheory of their elaboration-based semantics [Hall et al. 1996] has not yet been exhaustively studied. In particular, as far as we know, while there have been many informal arguments, the formal notion of *coherence* has never been proven.

Type classes give rise to two main (potential) sources of incoherence. The first source, discussed in Section 2.2.1, are *ambiguous type schemes* such as that of the function `foo`:

```
foo :: (Show a, Read a) => String -> String
foo s = show (read s)
```

To rule out this source of incoherence, Jones [1993] requires type schemes to be unambiguous and has formally proven that, for his system, this guarantees coherence. As mentioned earlier, Haskell rejects expressions with ambiguous type schemes.

The second source of ambiguity arises from the type class resolution mechanism itself. Such mechanisms check whether a particular type class constraint holds. Usually, they are styled after resolution-based proof search in logic, where type class instances act as Horn clauses and type scheme constraints as additional facts. Generally, this process is nondeterministic, but languages like Haskell, Mercury or PureScript contain it by requiring that type class instances do not overlap with each other or with locally given constraints. Nevertheless, superclasses remain as a source of nondeterminism; indeed, a superclass constraint can be resolved through any of its subclass constraints. Hence, in the presence of superclasses, type class resolution should properly be considered as a potential source for incoherence. Moreover, overlap between locally wanted constraints and global instances is often allowed (e.g., through GHC's `FlexibleContexts` pragma), but a formal argument for its harmlessness is also lacking. Jones [1993] considered neither of these aspects and simply assumed the coherence of resolution as a given. Morris [2014] side-steps these issues with a denotational semantics that is disconnected from the original elaboration-based semantics and its implementations (e.g., Hugs and GHC).

This chapter aims to fill this gap in the metatheory of programming languages featuring type classes, including industrial grade languages such as Haskell, by formally establishing that elaboration-based type class resolution is coherent in the presence of superclasses and flexible contexts. The proof of this property is considerably complicated by the indirect, elaboration-based approach used to give meaning to programs with type classes. Indeed, the meaning of such programs is commonly given in terms of their translation to a core language [Hall et al. 1996], like System F, the meaning of which is defined in the form of an operational semantics. In this translation process, type classes are elaborated into explicitly passed function dictionaries. These dictionaries can, however, often be constructed in more than one way, resulting in multiple possible translations for a single program. The problem is that different translations of the same source program may have different meanings in the core language. The reason for this discrepancy is that the core language is more expressive than the source language and admits programs — that cannot be expressed in the source language — in which the different dictionaries can be distinguished.

We solve this problem with a new two-step approach that splits the problem

into two subproblems. The midway point is an intermediate language that makes type class dictionaries explicit, but—inspired by fully abstract compilation—cannot distinguish between different elaborations from the same source language term [Abadi 1999]. We use a logical-relations approach to show that the nondeterministic elaboration from the source language to this intermediate language is coherent. Showing coherence for the elaboration from the intermediate language to the target language is much simpler, because we can formulate it in a deterministic fashion.

In summary, the contributions of this work are:

- We present a simple calculus $\boldsymbol{\lambda}_{\text{TC}}$ with full-blown type class resolution (incl. superclasses), which isolates nondeterministic resolution. Furthermore, we present an elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to the target language $\mathbf{F}_{\{\}}$, System F with records, which are used to encode dictionaries.
- We present an intermediate language $\mathbf{F}_{\text{D}}$ with explicit dictionary-passing. This language enforces the uniqueness of dictionaries, which captures the intention of type class instances. We study its metatheory, and define a logical relation to prove contextual equivalence.
- We present elaborations from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$ and from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$, and prove that a direct translation from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$ can always be decomposed into an equivalent translation through $\mathbf{F}_{\text{D}}$.
- We prove coherence of the elaboration between $\boldsymbol{\lambda}_{\text{TC}}$ and $\mathbf{F}_{\text{D}}$, using logical relations.
- We prove that coherence is also preserved through the elaboration from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$. As a consequence, by combining this with the previous result, we prove that the elaboration between $\boldsymbol{\lambda}_{\text{TC}}$ and $\mathbf{F}_{\{\}}$ is coherent. The latter coherence result implies coherence of elaboration-based type class resolution in the presence of superclasses and flexible contexts.

The purpose of our work is twofold: 1) To develop a proof technique to establish coherence of type class resolution. Because this result is achieved on a minimal calculus, this work becomes a basis for researchers investigating type class extensions and larger languages, as well as their impact on coherence. 2) To present a formal proof of coherence for language designers considering to adopt type classes. In doing so, we show that the informally trivial argument for the coherence of type class resolution is surprisingly hard to formalize.

As a guide to the reader, we present the source language $\boldsymbol{\lambda}_{\text{TC}}$ in black, the intermediate language $\mathbf{F}_{\text{D}}$ in green and the target language $\mathbf{F}_{\{\}}$ in red.

```
1  class Eq a where
2    (==) :: a -> a -> Bool
3
4  instance Eq Int where
5    (==) = primEqInt
6
7  instance (Eq a, Eq b) => Eq (a, b) where
8    (x1,y1) == (x2,y2)  =  x1 == x2 && y1 == y2
9
10 refl :: Eq a => a -> Bool
11 refl x = x == x
12
13 main :: Bool
14 main = refl (5,42)
```

Example 3.1: Program with type classes

3.1 Overview

This section provides some background on dictionary-passing elaboration of type class resolution and discusses the potential nondeterminism introduced by superclasses and local constraints. We then briefly introduce our calculi and discuss the key ideas of the coherence proof. Throughout this section we use Haskell-like syntax as the source language for examples, and the same syntax without type classes as the target language, to simplify our informal discussion.

3.1.1 Dictionary-Passing Elaboration

A program is coherent if it has exactly one meaning — i.e., its semantics is unambiguously determined. For type classes this is not as straightforward as it seems, because their dynamic semantics are not expressed directly but rather by type-directed elaboration into a simpler language without type classes such as System F. Thus, the dynamic semantics of type classes are given indirectly as the dynamic semantics of their elaborated forms.

Basic Elaboration. Consider the small program in Example 3.1, which repeats the declaration of the `Eq` type class and its `Eq Int` and `Eq (a, b)` instances for integers and tuples from Section 1.2.1. The function `refl` trivially tests whether an expression equals itself, and is called in `main`.

```
1 class Base a where
2   base :: a -> Bool
3
4 class Base a => Sub1 a where
5   sub1 :: a -> Bool
6
7 test1 :: Sub1 a => a -> Bool
8 test1 x = sub1 x && base x
```

Example 3.2: Program with superclasses

The dictionary-passing elaboration translates this program into a System F-like core language that does not feature type classes. The main idea of the elaboration is to map a type class declaration onto a datatype that contains the method implementations, a so-called *(function) dictionary*.

```
data EqD a = EqD { (==) :: a -> a -> Bool }
```

Then simple instances give rise to dictionary values:

```
eqInt :: EqD Int
eqInt = EqD { (==) = primEqInt }
```

Instances with a non-empty context are translated to functions that expect context dictionaries and produce the corresponding instance dictionary.

```
eqPair :: (EqD a, EqD b) -> EqD (a,b)
eqPair (da, db) =
  EqD { (==) = \(x1,y1) (x2,y2) -> (==) da x1 x2 && (==) db y1 y2 }
```

Functions with qualified types, like `refl`, are translated to functions that take explicit dictionaries as arguments.

```
refl :: EqD a -> a -> Bool
refl d x = (==) d x x
```

Finally, calls to functions with a qualified type are mapped to calls that explicitly pass the appropriate dictionary.

```
main :: Bool
main = refl (eqPair eqInt eqInt) (5,42)
```

Elaboration of Superclasses. Superclasses require a small extension to the above elaboration scheme. Consider the small program in Example 3.2 where `Sub1` is a subclass of `Base`. The function `test1` has `Sub1 a` in the context and calls `sub1` and `base` in its definition.

The standard approach to encode superclasses is to embed the superclass dictionary in that of the subclass. For this case, `Sub1D a` contains a field `super1` that points to the superclass:

```
data BaseD a = BaseD { base :: a -> Bool }
data Sub1D a = Sub1D { super1 :: BaseD a
                     , sub1 :: a -> Bool }
```

This way we can extract the superclass from the subclass when needed. The function `test1` is then encoded as:

```
test1 :: Sub1 a -> a -> Bool
test1 d x = sub1 d x && base (super1 d) x
```

Resolution. Calls to functions with a qualified type generate type class constraints. The process for checking whether these constraints can be satisfied, is known as *resolution*. For the sake of dictionary-passing elaboration, this resolution process is augmented with the construction of the appropriate dictionary that witnesses the satisfiability of the constraint.

3.1.2 Nondeterminism and Coherence

For Haskell'98 programs there is usually only one way to construct a dictionary for a type class constraint. Yet, in the presence of superclasses, there may be multiple ways. Suppose we extend Example 3.2 with an additional subclass and the following function:

```
class Base a => Sub2 a where
  sub2 :: a -> Bool

test2 :: (Sub1 a, Sub2 a) => a -> Bool
test2 x = base x
```

There are two possible ways to resolve the `Base a` constraint that arises from the call to `base` in function `test2`, resulting in the following two translations:

```
test2a, test2b :: (Sub1D a, Sub2D a) -> a -> Bool
test2a (d1,d2) x = base (super1 d1) x
test2b (d1,d2) x = base (super2 d2) x
```

The desired constraint can be resolved either as the superclass of the given `Sub1 a` constraint or as the superclass of the given `Sub2 a` constraint. Fortunately, this nondeterminism is harmless because the difference between the two elaborations cannot be observed. Indeed, for any given type `A`, Haskell'98 only allows a

single instance `Base A`, and it does not matter whether we access its dictionary directly or through one of its subclass instances. More generally, this suggests that type class resolution in Haskell'98 is coherent.

Example Program `pInt` A similar situation arises with the small example program `pInt` from Section 2.2.1. The `Ord a` type class (which has `Eq` as superclass) elaborates to the `OrdD a` dictionary type,

```
data OrdD a = OrdD { superOrdEq :: EqD a , (<=) :: a -> a -> Bool }
```

and the **`Ord Int`** instance elaborates to the `ordInt` dictionary value.

```
ordInt :: OrdD Int
ordInt = { superOrdEq = eqInt , (<=) = primIntLe }
```

Then, the two possible elaborations of `pInt` are shown below. `pInt1` uses the global **`Eq Int`** instance, while `pInt2` extracts the **`Eq Int`** instance as a superclass of the global **`Ord Int`** instance.

```
pInt1, pInt2 :: Int -> Int -> Bool
pInt1 x y = (==) eqInt x y
pInt2 x y = (==) (superOrdEq ordInt) x y
```

Flexible Contexts If we relax the Haskell'98 non-overlap condition for locally given constraints and adopt flexible contexts (to allow arbitrary types in class constraints, rather than simple type variables), another source of nondeterminism arises. Consider:

```
isZero :: Eq Int => Int -> Bool
isZero n = n == 0
```

There are three ways to resolve the wanted `Eq Int` constraint that arises from the use of `(==)`. Like before, we can use the global `Ord Int` constraint from which we can extract its `Eq Int` superclass (in `isZero1`), or we can use directly the global `Eq Int` instance (in `isZero2`). Moreover, there is a third option to use the locally given `Eq Int` constraint, passed as argument `d` (in `isZero3`):

```
isZero1, isZero2, isZero3 :: EqD Int -> Int -> Bool
isZero1 d n = (==) (superOrdInt ordInt) n 0
isZero2 d n = (==) eqInt n 0
isZero3 d n = (==) d     n 0
```

Haskell'98 does not allow the `Eq Int` constraint in `isZero`'s signature, which overlaps with the global `Eq Int` instance; it only allows constraints on type variables in function signatures. This prevents the above nondeterminism in

the elaboration. Yet, the nondeterminism is, once more, harmless; the supplied dictionary `d` can be nothing other than the global instance's dictionary `eqInt`. Informally, resolution remains coherent in the presence of flexible contexts.

3.1.3 Contextual Difference

While it is easy to provide an informal argument for the coherence of type class resolution, formally establishing the property is much harder. The indirect, elaboration-based semantics in particular is a complicating factor, since it requires us to reason about two languages simultaneously. Unfortunately, there is another factor that further complicates the proof: different elaborations of the same source program can actually be distinguished in the target language. Consider, for instance, the target program below:

```
discern :: ((Sub1D (), Sub2D ()) -> () -> Bool) -> Bool
discern f =
  let b1 = BaseD { base = \() -> True }
      b2 = BaseD { base = \() -> False }
      d1 = Sub1D { super1 = b1 }
      d2 = Sub2D { super2 = b2 }
  in f (d1,d2) ()
```

Observe that `discern test2a` evaluates to `True` and `discern test2b` evaluates to `False`. Hence, since `discern` can differentiate between them, `test2a` and `test2b` clearly do not have the same meaning in the target language.

The dictionaries for `Sub1 ()` and `Sub2 ()` have different implementations for their `Base ()` superclass. The source language would never allow this, but the target language has no notion of type classes and happily admits `discern`'s violation of source language rules.

The problem is that the target language is more expressive than the source language. While `test2a` and `test2b` cannot be distinguished in any program context that arises from the source language, we can write target programs like `discern` that are not the image of any source program and thus do not have to play by the source language rules.

3.1.4 Our Approach to Proving Coherence

To avoid the problem with contextual difference in the target language, we employ a novel two-step approach. We prove that any elaboration of a source language program into a dictionary-passing encoding in the target language, can

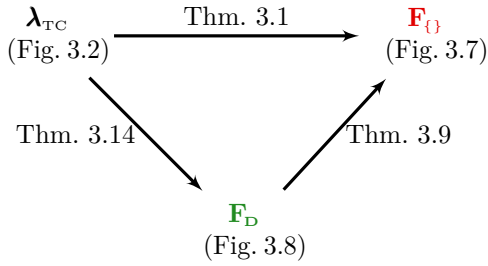


Figure 3.1: The different calculi with elaborations

be split in two separate elaborations through an intermediate language, thus obtaining two simpler problems for proving coherence of type class resolution.

The source language, $\boldsymbol{\lambda}_{\text{TC}}$, features full-fledged type class resolution, and simplifies term typing with a bidirectional type system (a technique popularized by Pierce & Turner [2000]) to not distract from the main objective of coherent resolution.

The intermediate language, $\mathbf{F}_{\text{D}}$ (presented in green), is an extension of System F that explicitly passes type class dictionaries, and preserves the source language invariant that there is at most one such dictionary value for any combination of class and type. We show $\mathbf{F}_{\text{D}}$ is type-safe and strongly normalizing, and define a logical relation that captures the contextual equivalence of two $\mathbf{F}_{\text{D}}$ terms.

The target language, $\mathbf{F}_{\{\}}$ (presented in red), is a different variant of System F without direct support for type class dictionaries; instead it features records–used to encode dictionaries–and does not enforce uniqueness of instances.

The different calculi are presented in Figure 3.1, where the edges denote possible elaborations.

The coherence proof consists of two main parts:

Coherent Elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$. Our elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ into $\mathbf{F}_{\text{D}}$ is nondeterministic, but type preserving. Furthermore, we show that any two $\mathbf{F}_{\text{D}}$ elaborations of the same $\boldsymbol{\lambda}_{\text{TC}}$ term are logically related, and prove that this logical relation implies contextual equivalence. This establishes that the elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$ is coherent.

Deterministic Elaboration from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$. Because of the syntactic similarity between $\mathbf{F}_{\text{D}}$ and $\mathbf{F}_{\{\}}$, the elaboration from the former into the latter

is a more straightforward affair. In addition to being type preserving, it is also deterministic, and preserves contextual equivalence.

These results are easily combined to show the coherence of the elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$, which implies coherence of elaboration-based type class resolution. The full proofs and definitions can be found in the Appendix of [Bottu et al. 2019], which is not included in this thesis because of its great size. Note that the proofs depend on a number of standard boilerplate conjectures (e.g., substitution lemmas), which can be found in the Appendix Sections J.1 and K.1 of [Bottu et al. 2019].

3.2 Source Language $\boldsymbol{\lambda}_{\text{TC}}$

This section presents our source language $\boldsymbol{\lambda}_{\text{TC}}$, a basic calculus which isolates nondeterministic resolution. The calculus only supports features that are essential for type class resolution and its coherence.

Consequently, the language is strongly normalizing, and thus does not support recursive let expressions, mutual recursion or recursive methods. This is a sensible choice, as recursion does not affect the fundamentals of the coherence proof. This work could include recursion through step indexing [Ahmed 2006], a well-known technique, but this would significantly clutter the proof. Recursion is discussed in more detail in Section 3.7.

Furthermore, two notable design decisions were made in the support of superclasses in $\boldsymbol{\lambda}_{\text{TC}}$. Firstly, similar to GHC, $\boldsymbol{\lambda}_{\text{TC}}$ derives all possible superclass constraints from their subclass constraints in advance, instead of deriving them "just-in-time" during resolution. The advantage of this approach is that it streamlines the actual resolution process.

Secondly, similar to Coq [Sozeau & Oury 2008] and unlike Wadler & Blott [1989], we pass superclass dictionaries alongside their subclass dictionaries, i.e., in a *flattened* form, instead of nesting them inside their subclass dictionaries. As it is not too difficult to see that both approaches are isomorphic, flattening the superclasses does not impact the coherence of resolution. It does, however, considerably simplify the proof, since this way neither our type class resolution mechanism, nor the intermediate language $\mathbf{F}_{\text{D}}$ (Section 3.5) need to have any support for superclasses and can treat them as regular local constraints. A more structured representation would give rise to additional complexity, but would not alter the essence of the proof.

Monotype	τ	::=	$\mathit{Bool} \mid a \mid \tau_1 \to \tau_2$
Qualified Type	ρ	::=	$\tau \mid Q \Rightarrow \rho$
Type Scheme	σ	::=	$\rho \mid \forall a.\sigma$
Class Constraint	Q	::=	$\mathit{TC}\,\tau$
Constraint Scheme	C	::=	$\forall \overline{a}.\overline{Q} \Rightarrow Q$
Term	e	::=	$\mathit{True} \mid \mathit{False} \mid x \mid m \mid \lambda x.e \mid e_1\, e_2 \mid$ **let** $x : \sigma = e_1$ **in** $e_2 \mid e :: \tau$
$\boldsymbol{\lambda}_{\mathrm{TC}}$ Program	pgm	::=	$e \mid \mathit{cls}; \mathit{pgm} \mid \mathit{inst}; \mathit{pgm}$
Class Declaration	cls	::=	**class** $\overline{\mathit{TC}_i\ a} \Rightarrow \mathit{TC}\ a$ **where** $\{m : \sigma\}$
Instance Declaration	inst	::=	**instance** $\overline{Q} \Rightarrow \mathit{TC}\ \tau$ **where** $\{m = e\}$
Typing Environment	Γ	::=	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q$
Class Environment	Γ_C	::=	$\bullet \mid \Gamma_C, m : \overline{\mathit{TC}_i\ a} \Rightarrow \mathit{TC}\, a : \sigma$
Program Context	P	::=	$\bullet \mid P, (D : C).m \mapsto \Gamma : e$
Expression Context	M	::=	$[\bullet] \mid \lambda x.M \mid M\, e \mid e\, M \mid M :: \tau \mid$ **let** $x : \sigma = M$ **in** $e \mid$ **let** $x : \sigma = e$ **in** M

Figure 3.2: $\boldsymbol{\lambda}_{\mathrm{TC}}$ syntax

Syntax. Figure 3.2 presents the, mostly standard, syntax. Programs consist of a number of class (with superclasses) and instance declarations, and an expression. For the sake of simplicity and well-foundedness, the declarations are ordered and can only refer to previous declarations.

Following Jones [1994]'s qualified types framework, we distinguish between three sorts of types: monotypes τ, qualified types ρ which include constraints, and type schemes σ which include type abstractions. Constraints differentiate between full constraint schemes C and simple class constraints Q. Observe that we allow flexible contexts in the qualified types; they are not restricted to constraints on type variables.

The definition of expressions e is standard, but with a few notable exceptions. Firstly, regular variables x are syntactically different from method names m, which are introduced in class declarations. Secondly, type annotations $e :: \tau$ allow the programmer to manually assign a monotype to an expression. This is useful for resolving ambiguity—see the *Typing* paragraph below. Finally, **let** bindings include type annotations with a type scheme σ, allowing the programmer to introduce local constraints—also discussed in the *Typing* paragraph. Note that we use Haskell syntax for class and instance declarations.

There are three $\boldsymbol{\lambda}_{\text{TC}}$ environments: two global ones and one local environment. Firstly, the global class environment Γ_C stores all class declarations. Each entry in Γ_C contains the method name m, any superclasses $\overline{TC_i\ a}$, the class $TC\,a$ itself and the corresponding method type σ.

Secondly, the global program context P contains all instance declarations. Each entry in P consists of a unique dictionary constructor D, its corresponding constraint C, the method name m and its implementation e, together with the context Γ under which e should be interpreted. This context contains the local axioms available in this instance declaration, as well as any axioms which explicitly annotate the method type signature.

Thirdly, the local typing environment Γ, besides containing the default term and type variables x and a, also stores any local axioms Q. As opposed to the program context P, Γ does not contain any type class instances. Instead, the (local) axioms are associated with a dictionary variable δ. Sections 3.5 and 3.5.1 explain the use of these dictionaries.

Typing. Our type system features two design choices to eliminate the possibility of ambiguous type schemes. This allows us to focus on the coherence of type class resolution, by making our proof orthogonal to ambiguous type schemes, the source of ambiguity which has already been studied by Jones [1993]. We thus side-step an already solved problem and focus on tackling the full problem of resolution coherence.

Firstly, we require type signatures to be unambiguous (Figure 3.3, bottom) to make sure that all newly introduced type variables are bound in the head of the type (the remaining monotype after dropping all type and constraint abstractions). This prevents ambiguous expressions such as:

```
let f : ∀a . Eq a => Int -> Int -- ambiguous
      = \ x . x + 1 in f 42
```

Secondly, we use a bidirectional type system rather than a fully declarative one. A bidirectional type system distinguishes between two typing modes: *inference* and *check* mode. The former synthesizes a type from the given expression, while the latter checks whether a given expression is of a given type. Special in our setting is that variables can only be typed in check mode, to ensure that only a single instantiation exists. This avoids the ambiguity that can arise when instantiating type variables in inference mode. Consider the following example:

```
let y : ∀a . Eq a => a -> a = ...
in const 1 y
```

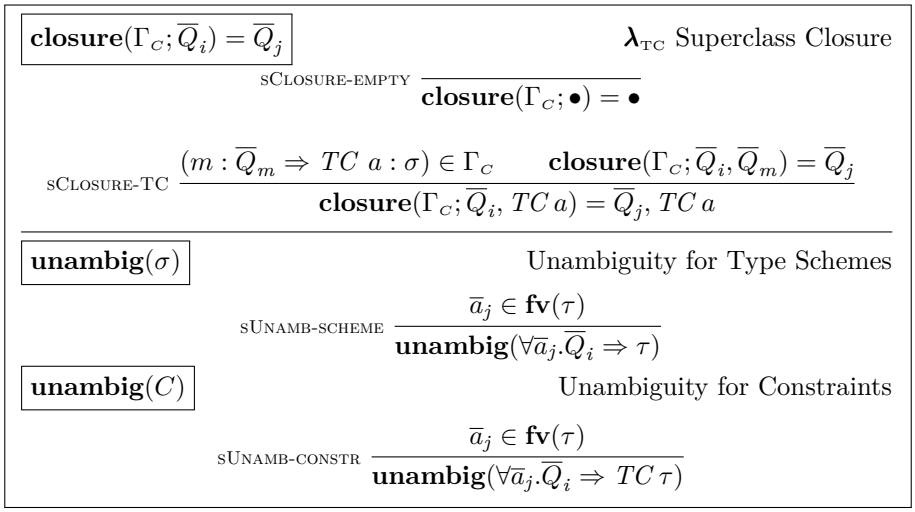


Figure 3.3: Closure and unambiguity relations

where `const x` is the constant function, which evaluates to `x` for any input. The instantiation of `y`'s type scheme is not uniquely determined by the context in which it is used. In a declarative type system or in inference mode, this ambiguity would result in multiple distinct typings and corresponding elaborations. While this ambiguity is harmless, it is not the focus of this work. Hence, to focus exclusively on the resolution, we use a bidirectional type system with check mode for variables to eliminate this irrelevant source of ambiguity.

Figure 3.4[1] shows selected typing rules. The full set of rules can be found in the Appendix Section B.2 of [Bottu et al. 2019]. We ignore the red (elaboration related) parts for now and explain them in detail in Section 3.3.1. The judgments $P; \Gamma_C; \Gamma \vdash_{\text{tm}} e \Rightarrow \tau$ and $P; \Gamma_C; \Gamma \vdash_{\text{tm}} e \Leftarrow \tau$ denote typing in inference mode and check mode, respectively, under environments P, Γ_C and Γ. The omitted constraint and type well-formedness relations, $\vdash_{\text{Q}}$ and $\vdash_{\text{ty}}$, are standard well-scopedness checks and can be found in the Appendix Section B.1 of [Bottu et al. 2019].

Through a **let** binding (rule sTm-infT-let), the programmer provides a type scheme for a variable, thus potentially introducing local constraints. As explained above, the unambiguity check from Figure 3.3 (bottom) requires the provided type scheme to be unambiguous. In order to flatten the superclasses,

[1]Note that lists, such as $\overline{\tau}_i$, are denoted by overlines, whereas collections of predicates are annotated by their range. For instance, $\left(\Gamma_C; \Gamma \vdash_{\text{ty}} \tau_i \rightsquigarrow \sigma_i\ ,\ \forall i\right)$ iterates over i.

$\boxed{P; \Gamma_C; \Gamma \vdash_{\text{tm}} e \Rightarrow \tau \rightsquigarrow e}$ Inference Mode

$$\textsc{sTm-infT-let}\ \frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \qquad \mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_k \qquad \left(\Gamma_C; \Gamma \vdash_{\text{Q}} Q_k \rightsquigarrow \sigma_k\ ,\ \forall k\right) \\ \Gamma_C; \Gamma \vdash_{\text{ty}} \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_k \to \sigma \\ \overline{\delta}_k\ \mathbf{fresh} \qquad P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{Q}_k \vdash_{\text{tm}} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \vdash_{\text{tm}} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \end{array}}{\begin{array}{l} P; \Gamma_C; \Gamma \vdash_{\text{tm}} \mathbf{let}\ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau_2 \\ \rightsquigarrow \mathbf{let}\ x : \forall \overline{a}_j.\overline{\sigma}_k \to \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_k : \overline{\sigma}_k.e_1\ \mathbf{in}\ e_2 \end{array}}$$

$\boxed{P; \Gamma_C; \Gamma \vdash_{\text{tm}} e \Leftarrow \tau \rightsquigarrow e}$ Check Mode

$$\textsc{sTm-checkT-meth}\ \frac{\begin{array}{c} (m : \overline{Q}'_k \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \qquad P; \Gamma_C; \Gamma \vDash TC\,\tau \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{\text{ty}} \tau \rightsquigarrow \sigma \qquad \left(P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i \rightsquigarrow e_i\ ,\ \forall i\right) \\ \left(\Gamma_C; \Gamma \vdash_{\text{ty}} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j\right) \qquad \vdash_{\text{ctx}} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{\text{tm}} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow e.m\,\overline{\sigma}_j\,\overline{e}_i}$$

Figure 3.4: $\boldsymbol{\lambda}_{\text{TC}}$ typing, selected rules

the rule takes the closure over the superclass relation (Figure 3.3, top) of the user provided constraints $\overline{Q}_i$. It then adds the resulting set of constraints $\overline{Q}_k$ to the typing environment, under which to typecheck e_1. Finally, the type of e_2 is inferred under the extended environment.

Rule sTm-checkT-meth types a method call m in check mode, like regular variables, to avoid any ambiguity in the instantiation of the type variables in the method's type scheme. This includes both the type variable a from the class and any additional free variables $\overline{a}_j$ in the method type. Thus, although the rule uses substitutions in the type of m, which is unusual in check-mode, this is no threat to the decidability or determinism of typing: the type τ' and the variables $\overline{a}_j$ and a, subject to substitution, are found by a look-up in the environment Γ_C. Then, in a more algorithmic formulation of this rule, a unification process would calculate the types $\overline{\tau}_j$ and τ substituted in τ'. Furthermore, the rule uses the unambig-relation to avoid ambiguity in the method type scheme itself, by requiring that $\overline{a}_j$ and a occur in τ'. This would ensure uniqueness of the unification's solutions, in the algorithmic formulation. The rule also checks that all required constraints $\overline{Q}_i$ from the method type can be entailed.

The instance typing rule can be found in Figure 3.5. The judgment $P; \Gamma_C \vdash_{\text{inst}} inst : P'$ expresses that an instance declaration $inst$ results in a $\boldsymbol{\lambda}_{\text{TC}}$ program context P', while being typed under environments P and Γ_C. The unambig-

$\boxed{P; \Gamma_C \vdash_{\mathrm{inst}} inst : P'}$ Instance Declaration Typing

$$\textsc{sInstT-inst}\ \frac{\begin{array}{c}(m : \overline{Q}_i' \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_h' \Rightarrow \tau_1) \in \Gamma_C \qquad \Gamma_C; \bullet, \overline{b}_k \vdash_{\mathrm{ty}} \tau \\ \mathbf{closure}(\Gamma_C; \overline{Q}_p) = \overline{Q}_q \qquad \mathbf{unambig}(\forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\,\tau) \\ \overline{\delta}_q\ \mathbf{fresh} \qquad \overline{\delta}_h'\ \mathbf{fresh} \qquad \left(P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q \vDash [\tau/a]Q_i'\ ,\ \forall i\right) \\ P; \Gamma_C; \Gamma' \vdash_{\mathrm{tm}} e \Leftarrow [\tau/a]\tau_1 \qquad \left(\Gamma_C; \bullet, \overline{b}_k \vdash_{\mathrm{Q}} Q_q\ ,\ \forall q\right) \\ D\ \mathbf{fresh} \qquad (D' : \forall \overline{b}_m'.\overline{Q}_n' \Rightarrow TC\,\tau_2).m' \mapsto \Gamma'' : e' \notin P \\ \text{where } [\overline{\tau}_m'/\overline{b}_m']\tau_2 = [\overline{\tau}_k'/\overline{b}_k]\tau \text{ and } \overline{b}_k = \mathbf{fv}(\tau) \\ \text{and } \Gamma' = \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}_h' : [\tau/a]\overline{Q}_h' \end{array}}{\begin{array}{l} P; \Gamma_C \vdash_{\mathrm{inst}} \mathbf{instance}\ \overline{Q}_p \Rightarrow TC\,\tau\ \mathbf{where}\ \{m = e\} \\ \qquad : (D : \forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\,\tau).m \mapsto \Gamma' : e \end{array}}$$

Figure 3.5: $\boldsymbol{\lambda}_{\mathrm{TC}}$ instance declaration typing

relation for constraints (Figure 3.3, bottom), similarly to the unambig-relation for types, checks that all free type variables $\overline{b}_k$ in the instance context occur in the instance type τ as well, in order to avoid ambiguity. Like in the sTmInfT-let rule explained above, the superclasses of the instance context $\overline{Q}_p$ are flattened into additional local constraints $\overline{Q}_q$ and added to the environment. The superclasses $\overline{Q}_i'$ of the instantiated type class are then checked to be entailed under this extended environment. The rule checks that no overlapping instance declarations D' have been defined. Finally, the program context is extended with the new instance axiom D, consisting of a constraint scheme that requires the full set of local constraints $\overline{Q}_q$.

Type Class Resolution. The type class resolution rules can be found in Figure 3.6, where $P; \Gamma_C; \Gamma \vDash Q$ denotes that a class constraint Q is entailed under the environments P, Γ_C and Γ. A wanted constraint Q can either be resolved using a locally available constraint δ (rule sEntailT-local) or through a global instance declaration D (rule sEntailT-inst). The former is entirely straightforward. The latter is more involved as an instance D may have an instance context $\overline{Q}_i'$, which has to be recursively resolved. Before resolving the context, the type variables $\overline{a}_j$ are instantiated with the corresponding concrete types $\overline{\tau}_j$, originating from the wanted constraint Q.

Note that the type class resolution mechanism does not require any specific support for superclasses, as these have all been flattened into regular local constraints.

$\boxed{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow e}$ Constraint Entailment

$$\textsc{sEntailT-local}\ \frac{(\delta : Q) \in \Gamma \qquad \vdash_{\text{ctx}} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma}{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow \delta}$$

$$\textsc{sEntailT-inst}\ \frac{\begin{array}{c} P_1; \Gamma_C; \Gamma' \vdash_{\text{tm}} e \Rightarrow \tau \rightsquigarrow e \qquad \left(\Gamma_C; \bullet, \overline{a}_j \vdash_{\text{Q}} Q'_i \rightsquigarrow \sigma'_i\ ,\ \forall i\right) \\ \left(\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_{\text{Q}} Q_h \rightsquigarrow \sigma''_h\ ,\ \forall h\right) \qquad \left(\Gamma_C; \Gamma \vdash_{\text{ty}} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j\right) \\ \left(P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j/\overline{a}_j]Q'_i \rightsquigarrow e_i\ ,\ \forall i\right) \qquad \vdash_{\text{ctx}} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ \text{where } P = P_1, (D : \forall \overline{a}_j.\overline{Q}'_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \text{and } \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}'_i, \overline{b}_k, \overline{\delta}_h : \overline{Q}_h \text{ and } Q = [\overline{\tau}_j/\overline{a}_j]Q' \end{array}}{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow (\Lambda \overline{a}_j.\lambda \overline{\delta}'_i : \overline{\sigma}'_i.\{m = \Lambda \overline{b}_k.\lambda \overline{\delta}_h : \overline{\sigma}''_h.e\})\, \overline{\sigma}_j\, \overline{e}_i}$$

Figure 3.6: $\boldsymbol{\lambda}_{\text{TC}}$ constraint entailment

Type	σ	::=	$\mathit{Bool} \mid a \mid \forall a.\sigma \mid \sigma_1 \to \sigma_2 \mid \{\overline{m_i : \sigma_i}^{\,i}\}$
Term	e	::=	$\mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma.e \mid e_1\, e_2 \mid \Lambda a.e \mid e\,\sigma \mid$ $\{\overline{m_i = e_i}^{\,i}\} \mid e.m \mid \textbf{let}\, x : \sigma = e_1\, \textbf{in}\, e_2$
Context	Γ	::=	$\bullet \mid \Gamma, a \mid \Gamma, x : \sigma$

Figure 3.7: $\mathbf{F}_{\{\}}$ syntax

3.3 Target Language $\mathbf{F}_{\{\}}$

This section covers our target calculus $\mathbf{F}_{\{\}}$ and the elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$.

The target language is System F with records, which we consider a reasonable subcalculus of those used by Haskell compilers. Its syntax is shown in Figure 3.7, while its typing rules are shown in Appendix A of this thesis. We omit its call-by-name operational semantics and refer the reader to [Pierce 2002, Chapter 23], or the Appendix Section E of [Bottu et al. 2019].

3.3.1 Elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$

The red aspects in Figure 3.4 denote the elaboration of $\boldsymbol{\lambda}_{\text{TC}}$ terms to $\mathbf{F}_{\{\}}$. We have adopted the convention that any red $\mathbf{F}_{\{\}}$ types are the elaborated forms of their identically named and black-colored $\boldsymbol{\lambda}_{\text{TC}}$ counterparts. This elaboration maps most $\boldsymbol{\lambda}_{\text{TC}}$ forms on identical $\mathbf{F}_{\{\}}$ terms, with the exception of a few notable cases: (a) The interesting aspect of elaborating let expressions (rule sTm-infT-let) is that, as mentioned previously, superclasses are flattened into additional

local constraints. The elaborated expression thus explicitly requires both the type variables and the full closure of the local constraints. (b) As opposed to $\boldsymbol{\lambda}_{\text{TC}}$, dictionary and type application are made explicit in $\mathbf{F}_{\{\}}$. When elaborating variables x and method references m (rule sTm-checkT-meth), all previously substituted types $\overline{\tau}_j$ are now explicitly applied, together with the dictionary expressions $\overline{e}_i$. Furthermore, method names m are elaborated to $\mathbf{F}_{\{\}}$ record labels m and therefore cannot appear by themselves, but must be applied to a record expression e, which originates from resolving the class constraint.

Type class resolution (Figure 3.6) of a $\boldsymbol{\lambda}_{\text{TC}}$ constraint Q results in a $\mathbf{F}_{\{\}}$ expression e. When resolving the wanted constraint using a locally available constraint δ (rule sEntailT-local), this results in a regular term variable δ (which keeps the name of its $\boldsymbol{\lambda}_{\text{TC}}$ counterpart for readability). On the other hand, when resolving with the use of a global instance declaration D (rule sEntailT-inst), a record expression is constructed, containing the method name m and its corresponding implementation e. This method implementation now explicitly abstracts over the type variables $\overline{b}_k$ and term variables $\overline{\delta}_h$ originating from the method types's class constraints $\overline{Q}_h$, which annotate the class declaration. Furthermore, the record expression is nested in abstractions over the type variables $\overline{a}_j$ and term variables $\overline{\delta}'_i$ arising from the corresponding instance constraints $\overline{Q}'_i$. These abstractions are immediately instantiated by applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint Q to the instance declaration Q' and (b) the expressions $\overline{e}_i$ constructed by resolving the instance context constraints $\overline{Q}'_i$.

Example 3.1 ($\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$). Typing the program from Example 3.1 results in the following environments:

$$\Gamma_C = (==) : Eq\, a : a \to a \to Bool$$

$$\begin{aligned} P = {} & (D_1 : Eq\ Int).(==) \mapsto \bullet : primEqInt \\ & , (D_2 : \forall a, b. Eq\, a \Rightarrow Eq\, b \Rightarrow Eq\, (a, b)).(==) \mapsto a, b, \delta_1 : Eq\, a, \delta_2 : Eq\, b : \\ & \lambda(x_1, y_1).\lambda(x_2, y_2).\ (\&\&)\ ((==)\, x_1\, x_2)\ ((==)\, y_1\, y_2) \end{aligned}$$

The Eq class straightforwardly gets stored in the class environment Γ_C. Instances are stored in the $\boldsymbol{\lambda}_{\text{TC}}$ program context P (containing the dictionary constructor, the corresponding constraint, the method implementation and the environment under which to interpret this expression). Storing the instance declaration for $Eq\ Int$ is clear-cut. The instance for tuples on the other hand is somewhat more complex, since it requires an instance context, containing the local constraints $Eq\, a$ and $Eq\, b$. These constraints are made explicit, that is, the corresponding dictionaries are required by the elaborated implementation.

Elaborating the $\boldsymbol{\lambda}_{\text{TC}}$ expression results in the following $\mathbf{F}_{\{\}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \mathit{refl}\colon \forall a.\{(==) : a \to a \to \mathit{Bool}\} \to a \to \mathit{Bool}\\
&\qquad = \Lambda a.\lambda \delta_3 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda x : a.\delta_3.(==)\, x\, x\\
&\textbf{in}\ \textbf{let}\ \mathit{main}\colon \mathit{Bool}\\
&\qquad = \mathit{refl}\,(\mathit{Int}, \mathit{Int})\\
&\qquad\quad (\Lambda a.\Lambda b.\lambda \delta_4 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda \delta_5 : \{(==) : b \to b \to \mathit{Bool}\}.\\
&\qquad\qquad \{(==) = \lambda (x_1, y_1) : (a, b).\lambda (x_2, y_2) : (a, b).\\
&\qquad\qquad\quad (\&\&)\,(\delta_4.(==)\, x_1\, x_2)\,(\delta_5.(==)\, y_1\, y_2)\})\\
&\qquad\quad \mathit{Int}\ \mathit{Int}\,\{(==) = \mathit{primEqInt}\}\,\{(==) = \mathit{primEqInt}\}\,(5, 42)\\
&\textbf{in}\ \mathit{main}
\end{aligned}
$$

In this expression, the $\mathit{Eq}\, a$ constraint is made explicit in the implementation of *refl*, by abstracting over the constraint (elaborated to $\mathbf{F}_{\{\}}$ as the record type $\{(==) : a \to a \to \mathit{Bool}\}$, which stores the method name and its corresponding type) with the use of the record variable δ_3. When this function is called in *main*, both the type and the dictionary variable are instantiated. The latter is performed by (recursively) constructing a dictionary expression, using the type class resolution mechanism, as explained above in Section 3.3.1.

Note that, in the informal presentation of the dictionary-passing elaboration technique in Section 3.1.1, the elaboration of the `main` function uses the `eqInt` and `eqPair` identifiers in its body, while here these functions are inlined. The difference is that in the informal presentation, method implementations are elaborated into top-level **let**-definitions and type-class constraints are resolved in terms of the introduced **let**-bound variables, while here method implementations are directly stored in a global map P and dictionaries are constructed by directly elaborating the corresponding method implementations.

Concretely for this example, the resulting expression using the approach in Section 3.1.1 would be the following:

$$
\begin{aligned}
&\textbf{let}\ \mathit{eqInt} : \{(==) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\} = \{(==) = \mathit{primEqInt}\}\\
&\textbf{in}\ \textbf{let}\ \mathit{eqPair}\colon \forall a.\forall b.\{(==) : a \to a \to \mathit{Bool}\} \to \{(==) : b \to b \to \mathit{Bool}\}\\
&\qquad\qquad \to \{(==) : (a, b) \to (a, b) \to \mathit{Bool}\}\\
&\qquad\qquad = \Lambda a.\Lambda b.\lambda \delta_1 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda \delta_2 : \{(==) : b \to b \to \mathit{Bool}\}.\\
&\qquad\qquad\quad \{(==) = \lambda (x_1, y_1) : (a, b).\lambda (x_2, y_2) : (a, b).\\
&\qquad\qquad\qquad\quad (\&\&)\,(\delta_1.(==)\, x_1\, x_2)\,(\delta_2.(==)\, y_1\, y_2)\}\\
&\textbf{in}\ \textbf{let}\ \mathit{refl}\colon \forall a.\{(==) : a \to a \to \mathit{Bool}\} \to a \to \mathit{Bool}\\
&\qquad\qquad = \Lambda a.\lambda \delta : \{(==) : a \to a \to \mathit{Bool}\}.\lambda x : a.\,\delta.(==)\, x\, x\\
&\textbf{in}\ \textbf{let}\ \mathit{main} : \mathit{Bool} = \mathit{refl}\,(\mathit{Int}, \mathit{Int})\,(\mathit{eqPair}\ \mathit{Int}\ \mathit{Int}\ \mathit{eqInt}\ \mathit{eqInt})\,(5, 42)\\
&\textbf{in}\ \mathit{main}
\end{aligned}
$$

The two presentations are essentially the same. We use the top-level **let**-definitions in Section 3.1.1 as a more human-friendly version of the elaboration mechanism of this section. However, from a meta-theoretical point of view, the top-level **let**-definitions would only add an unnecessary layer of elaboration steps in the language specification.

Example 3.2 ($\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_{\{\}}$). Below is the environment generated by typing the Example 3.2 program (including the Section 3.1.2 extension), which features superclasses.

$$\begin{aligned}\Gamma_C = \; & \mathit{base} : \mathit{Base}\, a : a \to \mathit{Bool} \\ , \; & \mathit{sub}_1 : \mathit{Base}\, a \Rightarrow \mathit{Sub}_1\, a : a \to \mathit{Bool} \\ , \; & \mathit{sub}_2 : \mathit{Base}\, a \Rightarrow \mathit{Sub}_2\, a : a \to \mathit{Bool}\end{aligned}$$

The class environment Γ_C contains three classes, two of which have superclasses. However, since the example does not contain any instance declarations, the resulting program context P is empty.

Elaborating `test1` results in the following $\mathbf{F}_{\{\}}$ expression:

$$\begin{aligned}&\textbf{let}\;\; \mathit{test}_1 \colon \forall a.\{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_1 : a \to \mathit{Bool}\} \to a \to \mathit{Bool} \\ &\qquad = \Lambda a.\lambda \delta_1 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda \delta_2 : \{\mathit{sub}_1 : a \to \mathit{Bool}\}.\lambda x : a. \\ &\qquad \delta_1.\mathit{base}\, x \;\&\&\; \delta_2.\mathit{sub}_1\, x \\ &\textbf{in}\;\; \ldots\end{aligned}$$

Note that while the $\boldsymbol{\lambda}_{\mathrm{TC}}$ expression requires only the local constraint $\mathit{Sub}_1\, a$, after flattening the superclasses and adding them to the local constraints, the elaborated $\mathbf{F}_{\{\}}$ expression requires (the elaborated form of) the $\mathit{Base}\, a$ and $\mathit{Sub}_1\, a$ constraints.

Similarly, elaborating `test2`, results in the following $\mathbf{F}_{\{\}}$ expression:

$$\begin{aligned}&\textbf{let}\;\; \mathit{test}_2 \colon \forall a.\{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_1 : a \to \mathit{Bool}\} \\ &\qquad \to \{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_2 : a \to \mathit{Bool}\} \to a \to \mathit{Bool} \\ &\qquad = \Lambda a.\lambda \delta_1 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda \delta_2 : \{\mathit{sub}_1 : a \to \mathit{Bool}\}. \\ &\qquad \lambda \delta_3 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda \delta_4 : \{\mathit{sub}_2 : a \to \mathit{Bool}\}.\lambda x : a.\; \delta_1.\mathit{base}\, x \\ &\textbf{in}\;\; \ldots\end{aligned}$$

Here, after flattening the superclasses and adding them to the local constraints, the elaborated $\mathbf{F}_{\{\}}$ expression requires (the elaborated form of) the $\mathit{Base}\, a$, $\mathit{Sub}_1\, a$, $\mathit{Base}\, a$ and $\mathit{Sub}_2\, a$ constraints. Notice the duplicate $\mathit{Base}\, a$ entry. Either of these two entries can be used for calling the method base. We have arbitrarily selected the first here. The next section proves that both options are equivalent and can be used interchangeably.

3.4 Coherence

This section provides an outline for our coherence proof, and defines the required notions[2]. We first provide a definition of *contextual equivalence* [Morris Jr 1969], which captures that two expressions have the same meaning.

3.4.1 Contextual Equivalence

In order to formally discuss the concept of contextual equivalence, we first define the notion of an *expression context*.

Expression Contexts. An expression context M is an expression with a single hole, for which another expression e can be filled in, denoted as $M[e]$. The syntax can be found in Figure 3.2.

The typing judgment for an expression context M is of the form

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M.$$

which expresses that for a term e, if $P; \Gamma_C; \Gamma \vdash_{\text{tm}} e \Rightarrow \tau \rightsquigarrow e$, then $P; \Gamma_C; \Gamma' \vdash_{\text{tm}} M[e] \Rightarrow \tau' \rightsquigarrow e'$. For example, the simplest expression context is the empty context $[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$. Following regular $\boldsymbol{\lambda}_{\text{TC}}$ term typing, context typing spans all combinations of type inference and checking mode: $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$, $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$ and $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$.

Now we can formally define contextual equivalence. The small step operational semantics can be found in the Appendix Section E.4 of [Bottu et al. 2019], while the environment and type well-formedness judgments in the Appendix Sections B.4 and B.1 of [Bottu et al. 2019], respectively.

Definition 3.1 (Kleene Equivalence)**.**
Two $\mathbf{F}_{\{\}}$ *expressions* e_1 *and* e_2 *are Kleene equivalent, written* $e_1 \simeq e_2$*, if there exists a value* v *such that* $e_1 \longrightarrow^* v$*, and* $e_2 \longrightarrow^* v$*.*

Kleene equivalence is also called *observational equivalence*, since the behaviour of the two subject expressions observed by an external user is determined by the same value v to which they evaluate.

[2]I would like to thank Dariusz Biernacki whose comments and feedback helped in simplifying the original coherence proof.

Definition 3.2 (Contextual Equivalence).
Two target expressions, e_1 *and* e_2*, are contextually equivalent, written* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$*, if and only if they are well-formed (*$\Gamma \vdash_{tm} e_i : \sigma$*, where* $i = 1, 2$*,* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma$*) and* $M[e_1] \simeq M[e_2]$ *holds for any* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M$.

The definition is adapted from Harper [2016, Chapter 46]. Put into words, two expressions are contextually equivalent, if they are observationally indistinguishable when used in any program that appropriately instantiates the expressions' free variables.

3.4.2 Coherence

We can now make a first attempt to prove that different translations of the same source program are contextually equivalent. The program typing judgment can be found in the Appendix Section B.2 of [Bottu et al. 2019].

Theorem 3.1 (Coherence).
If $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$
then $\Gamma_{C1} = \Gamma_{C2}$, $P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

We first set out to prove the simpler variant, which only considers expressions. Because we use bidirectional typing, this theorem is duplicated.

Theorem 3.2 (Expression Coherence).

- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$,
  then $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.
- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e_2$,
  then $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Note that bidirectional typing causes some theorem statements to be repeated for each typing mode. However, because this duplcation is not interesting from a metatheoretical perspectice, in the remaining chapter we explicitly show such theorem statements only for the inference mode.

The main requirement which makes type class resolution coherent is that type class instances do not overlap. However, $\mathbf{F}_{\{\}}$ does not enforce this crucial uniqueness property, since it uses records to encode dictionaries. In order to prove Theorem 3.1, we introduce an intermediate language $\mathbf{F}_{\mathbf{D}}$, which captures the invariant that type class instances do not overlap, and makes it explicit.

Type	σ, τ	::=	$\ldots \mid Q \Rightarrow \sigma$
Class Constraint	Q	::=	$TC\,\sigma$
Constraint	C	::=	$\forall \overline{a}.\overline{Q} \Rightarrow Q$
Dictionary	d	::=	$\delta \mid D\,\overline{\sigma}\,\overline{d}$
Dictionary Value	dv	::=	$D\,\overline{\sigma}\,\overline{dv}$
Expression	e	::=	$\ldots \mid \lambda\delta : Q.e \mid e\,d \mid d.m$
Typing Environment	Γ	::=	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q$
Class Environment	Γ_C	::=	$\bullet \mid \Gamma_C, m : TC\,a : \sigma$
Method Environment	Σ	::=	$\bullet \mid \Sigma, (D : C).m \mapsto e$

Figure 3.8: $\mathbf{F}_{\mathrm{D}}$, selected syntax

3.5 Intermediate Language $\mathbf{F}_{\mathrm{D}}$

This section presents our intermediate language $\mathbf{F}_{\mathrm{D}}$, which is modeled with three main goals in mind: (a) $\mathbf{F}_{\mathrm{D}}$ should explicitly pass type class dictionaries, which are implicit in $\boldsymbol{\lambda}_{\mathrm{TC}}$; (b) the $\mathbf{F}_{\mathrm{D}}$ type system should capture the uniqueness of dictionaries, thus enforcing the elaboration from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to preserve full abstraction; and (c) $\mathbf{F}_{\mathrm{D}}$ expressions should elaborate straightforwardly and deterministically to the target language $\mathbf{F}_{\{\}}$ (System F with records, see Section 3.3).

To this end, $\mathbf{F}_{\mathrm{D}}$ is an extension of System F with built-in support for dictionaries. These dictionaries differ from those commonly used in Haskell compilers in that they are special constants rather than a record of method implementations. A separate global map Σ from dictionaries to method implementations gives access to the latter. This setup does not allow programs to introduce new (and possibly overlapping) dictionaries dynamically. Note that a $\boldsymbol{\lambda}_{\mathrm{TC}}$ program (*pgm* in Figure 3.2) consists of a sequence of type class declarations and instance definitions, followed by an expression e. These instances are stored in a source program context P, which is elaborated into an $\mathbf{F}_{\mathrm{D}}$ environment Σ. Then, the $\boldsymbol{\lambda}_{\mathrm{TC}}$ expression e is elaborated under the P program context and the generated $\mathbf{F}_{\mathrm{D}}$ term e is considered under the global map Σ. Thus, all dictionaries are provided upfront and uniqueness is easily enforced.

Syntax. Figure 3.8 shows selected syntax of $\mathbf{F}_{\mathrm{D}}$; the basic System F constructs are omitted and replaced by an ellipsis ($\ldots$). The full syntax can be found in the Appendix Section A.2 of [Bottu et al. 2019]. $\mathbf{F}_{\mathrm{D}}$ introduces a new syntactic sort of dictionaries d that can either be a dictionary variable δ or a dictionary

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} e : \sigma}$ $\mathbf{F}_\mathrm{D}$ Term Typing

$$\textsc{iTm-method}\ \frac{(m : TC\, a : \sigma') \in \Gamma_C \qquad \Sigma; \Gamma_C; \Gamma \vdash_d d : TC\, \sigma}{\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} d.m : [\sigma/a]\sigma'}$$

$$\textsc{iTm-constrE}\ \frac{\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} e : Q \Rightarrow \sigma \qquad \Sigma; \Gamma_C; \Gamma \vdash_d d : Q}{\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} e\, d : \sigma}$$

$$\textsc{iTm-constrI}\ \frac{\Gamma_C; \Gamma \vdash_{\mathrm{Q}} Q \qquad \Sigma; \Gamma_C; \Gamma, \delta : Q \vdash_{\mathrm{tm}} e : \sigma}{\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} \lambda\delta : Q.e : Q \Rightarrow \sigma}$$

Figure 3.9: $\mathbf{F}_\mathrm{D}$ typing, selected rules

constructor D. A dictionary constructor has a number (possibly zero) of type and dictionary parameters and always appears in fully-applied form. Each constructor corresponds to a unique instance declaration, and is mapped to its method implementation by the global environment Σ.

Expressions have explicit application and abstraction forms for dictionaries. Furthermore, similarly to $\mathbf{F}_{\{\}}$, method names can no longer be used alone. Instead, they must be applied explicitly to a dictionary, in the form $d.m$.

$\mathbf{F}_\mathrm{D}$ types σ or τ are identical to the well-known System F types, with the addition of a special function type $Q \Rightarrow \sigma$ for dictionary abstractions.

Similarly to $\boldsymbol{\lambda}_{\mathrm{TC}}$, $\mathbf{F}_\mathrm{D}$ features two global and a single local environment Γ. The latter is similar to the $\boldsymbol{\lambda}_{\mathrm{TC}}$ typing environment Γ. However, there are notable differences between the global environments. The $\mathbf{F}_\mathrm{D}$ class environment Γ_C does not contain any superclass information. This is because, as previously mentioned in Section 3.2, superclass constraints in $\boldsymbol{\lambda}_{\mathrm{TC}}$ are flattened into local constraints, and stored in the typing environment Γ. The analog to the $\boldsymbol{\lambda}_{\mathrm{TC}}$ program context P is the $\mathbf{F}_\mathrm{D}$ method environment Σ, storing information about all dictionary constructors D. Each constructor corresponds to a unique instance declaration, and stores the accompanying method implementations.

Typing. Figure 3.9 shows selected typing rules for $\mathbf{F}_\mathrm{D}$ expressions. The red parts can be safely ignored for now, as they will be explained in detail in Section 3.5.2. The judgment $\Sigma; \Gamma_C; \Gamma \vdash_{\mathrm{tm}} e : \sigma$ expresses that the $\mathbf{F}_\mathrm{D}$ term e of type σ is well-typed under environments Σ, Γ_C and Γ. As shown by rule iTm-method, the type of a method variable applied to a dictionary is simply the

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e}$ Dictionary Typing

$$\textsc{d-var}\ \frac{(\delta : Q) \in \Gamma \qquad \vdash_{\text{ctx}} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \rightsquigarrow \delta}$$

$$\textsc{d-con}\ \frac{\begin{array}{c}\left(\Gamma_C; \bullet, \overline{a}_j \vdash_{\text{Q}} Q_i \rightsquigarrow \sigma'_i\ ,\ \forall i\right) \qquad \left(\Gamma_C; \Gamma \vdash_{\text{ty}} \sigma_j \rightsquigarrow \sigma_j\ ,\ \forall j\right) \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{\text{tm}} e : [\sigma_q/a]\sigma_m \rightsquigarrow e \\ \left(\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_j/\overline{a}_j]Q_i \rightsquigarrow e_i\ ,\ \forall i\right) \\ \Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{Q}_i.e, \Sigma_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D\,\overline{\sigma}_j\,\overline{d}_i : TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q \rightsquigarrow (\Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{\sigma}'_i.\{m = e\})\,\overline{\sigma}_j\,\overline{e}_i}$$

$\boxed{\Gamma_C; \Gamma \vdash_{\text{Q}} Q \rightsquigarrow \sigma}$ Constraint Well-Formedness

$$\Gamma_C; \Gamma \vdash_{\text{ty}} \sigma \rightsquigarrow \sigma$$

$$\textsc{iQ-TC}\ \frac{\Gamma_C = \Gamma_{C1}, m : TC\,a : \sigma', \Gamma_{C2} \qquad \Gamma_{C1}; \bullet, a \vdash_{\text{ty}} \sigma' \rightsquigarrow \sigma'}{\Gamma_C; \Gamma \vdash_{\text{Q}} TC\,\sigma \rightsquigarrow [\sigma/a]\{m : \sigma'\}}$$

Figure 3.10: $\mathbf{F}_{\text{D}}$ dictionary typing

corresponding method type (as stored in Γ_C), where the type variable has been substituted for the corresponding dictionary type.

Figure 3.10 (top) shows the typing rules for dictionaries. The relation $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q$ denotes that dictionary d of dictionary type Q is well-formed under environments Σ, Γ_C and Γ. Similarly to regular term variables x, the type of a dictionary variable δ (rule D-VAR) is obtained from the typing environment Γ. The type of a dictionary constructor D (rule D-CON), on the other hand, is obtained by finding the corresponding entry in the method environment Σ and substituting any types $\overline{\sigma}_j$ applied to it in the corresponding class constraint $TC\,\sigma_q$. All applied dictionaries $\overline{d}_i$ have to be well-typed with the corresponding constraint. Finally, the corresponding method implementation has to be well-typed in the reduced method environment Σ_1, which only contains the instances declared before D. As mentioned in Section 3.2, this reduced environment disallows recursive method implementations, as this would significantly clutter the coherence proof while, as a feature, recursion is completely orthogonal to the desired property.

Non-Overlapping Instances. The main requirement for achieving coherence of type class resolution, is that type class instances do not overlap. This requirement is common in Haskell and is for example enforced in GHC (though strongly discouraged, the `OverlappingInstances` pragma disables it). By

$\boxed{\vdash_{\text{ctx}} \Sigma; \Gamma_C; \Gamma}$ $\mathbf{F_D}$ Environment Well-Formedness

$$\textsc{iCtx-MEnv}\ \frac{\begin{array}{c}\mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma) \qquad \Gamma_C; \bullet \vdash_C \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma \\ (m : TC\,a : \sigma') \in \Gamma_C \qquad \Sigma; \Gamma_C; \bullet \vdash_{\text{tm}} e : \forall \overline{a}_j.\overline{Q}_i \Rightarrow [\sigma/a]\sigma' \\ D \notin \mathbf{dom}(\Sigma) \qquad (D' : \forall \overline{a}'_m.\overline{Q}'_n \Rightarrow TC\,\sigma').m' \mapsto e' \notin \Sigma \\ \vdash_{\text{ctx}} \Sigma; \Gamma_C; \Gamma \qquad \text{where } [\overline{\sigma}_j/\overline{a}_j]\sigma = [\overline{\sigma}'_m/\overline{a}'_m]\sigma'\end{array}}{\vdash_{\text{ctx}} \Sigma, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma).m \mapsto e; \Gamma_C; \Gamma}$$

Figure 3.11: $\mathbf{F_D}$ environment well-formedness, selected rules

$\boxed{\Sigma \vdash e \longrightarrow e'}$ $\mathbf{F_D}$ Evaluation

$$\textsc{iEval-DAppAbs}\ \frac{}{\Sigma \vdash (\lambda\delta : Q.e)\,d \longrightarrow [d/\delta]e} \qquad \textsc{iEval-method}\ \frac{(D : C).m \mapsto e \in \Sigma}{\Sigma \vdash (D\,\overline{\sigma}\,\overline{d}).m \longrightarrow e\,\overline{\sigma}\,\overline{d}} \qquad \textsc{iEval-DApp}\ \frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\,d \longrightarrow e'\,d}$$

Figure 3.12: $\mathbf{F_D}$ operational semantics, selected rules

storing all method implementations (with their corresponding instances) in a single environment Σ, this invariant can easily be made explicit.

Figure 3.11 shows the environment well-formedness condition $\vdash_{\text{ctx}} \Sigma; \Gamma_C; \Gamma$ for the method environment. Besides stating well-scopedness, it denotes that the method environment Σ cannot contain a second instance, for which the head of the constraint overlaps with $TC\,\sigma$, up to renaming. This key property will be exploited in our coherence proof.

Operational Semantics. As $\mathbf{F_D}$ is an extension of System F, its call-by-name operational semantics are mostly standard. The non-standard rules can be found in Figure 3.12 (bottom right), where $\Sigma \vdash e \longrightarrow e'$ denotes expression e evaluating to e' in a single step, under method environment Σ.

The evaluation rules for dictionary application (rules iEval-DApp and iEval-DAppAbs) are identical to those for term and type application. More interesting, however, is the evaluation for methods (rule iEval-method). A method name applied to a dictionary evaluates in one step to the method implementation, as stored in the environment Σ.

$\boxed{P; \Gamma_C; \Gamma \vdash^{\mathrm{M}}_{\mathrm{tm}} e \Leftarrow \tau \rightsquigarrow e}$ Source Term Checking

$$\textsc{Tm-check-meth}\ \frac{\begin{array}{c}(m : \overline{Q}'_k \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \qquad P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} TC\ \tau \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash^{\mathrm{M}}_{ty} \tau \rightsquigarrow \sigma \qquad \left(P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} [\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i \rightsquigarrow d_i\ ,\ \forall i\right) \\ \left(\Gamma_C; \Gamma \vdash^{\mathrm{M}}_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j\right) \qquad \vdash^{\mathrm{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma\end{array}}{P; \Gamma_C; \Gamma \vdash^{\mathrm{M}}_{\mathrm{tm}} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\,\overline{\tau}_j\,\overline{d}_i}$$

Figure 3.13: $\boldsymbol{\lambda}_{\mathrm{TC}}$ typing with elaboration to $\mathbf{F}_{\mathrm{D}}$, selected rules

Metatheory. $\mathbf{F}_{\mathrm{D}}$ is type safe. That is, the common progress and preservation properties hold:

Theorem 3.3 (Progress)**.**
If $\Sigma; \bullet; \bullet \vdash_{tm} e : \sigma$*, then either* e *is a value, or there exists* e' *such that* $\Sigma \vdash e \longrightarrow e'$*.*

Theorem 3.4 (Preservation)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$*, and* $\Sigma \vdash e \longrightarrow e'$*, then* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e' : \sigma$*.*

Analogously to $\boldsymbol{\lambda}_{\mathrm{TC}}$, $\mathbf{F}_{\mathrm{D}}$ rejects recursive expressions (including mutual recursion and recursive methods). This allows for a normalizing language, that is, any well-typed expression evaluates to a value, after a finite number of steps. Note that since the small step operational semantics are deterministic, normalization implies strong normalization.

Theorem 3.5 (Strong Normalization)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$*, then all possible evaluation derivations for* e *terminate. Formally,* $\exists v : \Sigma \vdash e \longrightarrow^* v$*.*

The proof follows the familiar structure for proving normalization using logical relations, as presented by Ahmed [2015], and can be found in the Appendix Section K.4 of [Bottu et al. 2019].

3.5.1 Elaboration from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_{\mathrm{D}}$

The green aspects in Figure 3.13 denote the elaboration of $\boldsymbol{\lambda}_{\mathrm{TC}}$ terms to $\mathbf{F}_{\mathrm{D}}$. Similarly to the elaboration from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_{\{\}}$, we have adopted the convention that any green $\mathbf{F}_{\mathrm{D}}$ types or constraints are the elaborated forms of their identically named $\boldsymbol{\lambda}_{\mathrm{TC}}$ counterparts. This elaboration works analogously to the elaboration

$\boxed{P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} Q \rightsquigarrow d}$ Constraint Entailment

$$\textsc{sEntail-local}\ \frac{(\delta : Q) \in \Gamma \qquad \vdash^{\mathrm{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} Q \rightsquigarrow \delta}$$

$$\textsc{sEntail-inst}\ \frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i, \overline{b}_k, \overline{\delta}_h : \overline{Q}_h \\ Q = [\overline{\tau}_j / \overline{a}_j] Q' \qquad \left(\Gamma_C; \Gamma \vdash^{\mathrm{M}}_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j\right) \\ \vdash^{\mathrm{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \qquad \left(P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} [\overline{\tau}_j / \overline{a}_j] Q_i \rightsquigarrow d_i\ ,\ \forall i\right) \end{array}}{P; \Gamma_C; \Gamma \vDash^{\mathrm{M}} Q \rightsquigarrow D\, \overline{\sigma}_j\, \overline{d}_i}$$

Figure 3.14: $\boldsymbol{\lambda}_{\mathrm{TC}}$ constraint entailment with elaboration to $\mathbf{F_D}$

from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_{\{\}}$, as shown in Figure 3.4. The full set of rules can be found in the Appendix Section C.2 of [Bottu et al. 2019].

The only notable case is rule sTm-check-meth, where the entailment relation for solving the type class constraint $TC\,\tau$ now results in a dictionary d. As explained at the start of Section 3.5, unlike $\mathbf{F}_{\{\}}$, $\mathbf{F_D}$ differentiates syntactically between dictionaries and normal expressions.

Type class resolution (Figure 3.14) of a $\boldsymbol{\lambda}_{\mathrm{TC}}$ constraint Q results in a $\mathbf{F_D}$ dictionary d. When using a locally available constraint to resolve the wanted constraint (rule sEntail-local), the corresponding dictionary variable δ is returned. On the other hand, when resolving using a global instance declaration (rule sEntail-inst), a dictionary is constructed by taking the corresponding constructor D and applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint to the instance declaration and (b) the dictionaries $\overline{d}_i$, constructed by resolving the instance context constraints.

Metatheory. We discuss the coherence of the elaboration from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F_D}$ in detail in Section 3.6, and mention here that it is type preserving:

Theorem 3.6 (Typing Preservation - Expressions)**.**
If $P; \Gamma_C; \Gamma \vdash^{\mathrm{M}}_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and if* $\vdash^{\mathrm{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and if* $\Gamma_C; \Gamma \vdash^{\mathrm{M}}_{ty} \tau \rightsquigarrow \sigma$*, then* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$*.*

The same theorem holds for the check mode. The full proofs can be found in the Appendix Section J.3 of [Bottu et al. 2019].

Example 3.1 ($\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$). Elaborating the $\boldsymbol{\lambda}_{\text{TC}}$ environments that originate from Example 3.1, results in the following $\mathbf{F}_{\text{D}}$ environments:

$$
\begin{aligned}
\Gamma_C &= (==) : Eq\, a : a \to a \to Bool \\
\Sigma &= (D_1 : Eq\, Int).(==) \mapsto primEqInt \\
&\quad , (D_2 : \forall a, b. Eq\, a \Rightarrow Eq\, b \Rightarrow Eq\, (a, b)).(==) \mapsto \\
&\qquad \Lambda a. \Lambda b. \lambda \delta_1 : Eq\, a. \lambda \delta_2 : Eq\, b. \lambda (x_1, y_1) : (a, b). \lambda (x_2, y_2) : (a, b). \\
&\qquad\quad (\&\&)\, (\delta_1.(==)\, x_1\, x_2)\, (\delta_2.(==)\, y_1\, y_2)
\end{aligned}
$$

Both the class environment Γ_C and the program context Σ are largely direct translations of their $\boldsymbol{\lambda}_{\text{TC}}$ counterparts. One notable difference is that the environment Γ under which to interpret the $\boldsymbol{\lambda}_{\text{TC}}$ method implementation is now explicitly abstracted over in the $\mathbf{F}_{\text{D}}$ method implementation. Consider for instance the case of D_2, where the variables a, b, δ_1 and δ_2, which in $\boldsymbol{\lambda}_{\text{TC}}$ are implicitly provided by the typing environment, are now explicit in the term level.

Elaborating the $\boldsymbol{\lambda}_{\text{TC}}$ expression results in the following $\mathbf{F}_{\text{D}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \ refl : \forall a. Eq\, a \Rightarrow a \to Bool \\
&\qquad\quad = \Lambda a. \lambda \delta_3 : Eq\, a. \lambda x : a. \delta_3.(==)\, x\, x \\
&\textbf{in let}\ \ main : Bool \\
&\qquad\quad = refl\, (Int, Int)\, (D_2\, Int\, Int\, D_1\, D_1)\, (5, 42) \\
&\textbf{in}\ \ main
\end{aligned}
$$

Unlike the corresponding $\mathbf{F}_{\{\}}$ expression, shown in Section 3.3.1, records storing the method types and implementations do not need to be passed around explicitly. In $\mathbf{F}_{\text{D}}$, they are replaced by class constraints and dictionaries, respectively. The construction of these dictionaries through type class resolution is shown in Figure 3.14.

Example 3.2 ($\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\text{D}}$). Elaborating Example 3.2, including the extension from Section 3.1.2, results in the following $\mathbf{F}_{\text{D}}$ class environment (since no instance declarations exist, the method environment Σ remains empty):

$$
\begin{aligned}
\Gamma_C = \ & base : a : a \to Bool \\
& , sub_1 : a : a \to Bool \\
& , sub_2 : a : a \to Bool
\end{aligned}
$$

The $\mathbf{F}_{\text{D}}$ class environment no longer needs to store superclasses, as these are all flattened into additional local constraints during elaboration.

Similarly to Section 3.3.1, we focus solely on elaborating `test2`, which results in the following $\mathbf{F_D}$ expression:

$$\begin{aligned}&\textbf{let}\ \ test_2 : \forall a.\, Base\, a \Rightarrow Sub_1\, a \Rightarrow Base\, a \Rightarrow Sub_2\, a \Rightarrow a \to Bool\\ &\qquad\quad = \Lambda a.\lambda\delta_1 : Base\, a.\lambda\delta_2 : Sub_1\, a.\lambda\delta_3 : Base\, a.\lambda\delta_4 : Sub_2\, a.\\ &\qquad\quad\ \ \lambda x : a.\delta_1.base\, x\\ &\textbf{in}\ \ True\end{aligned}$$

The only difference with the $\mathbf{F}_{\{\}}$ elaboration is that we now use class constraints instead of passing around a record type (storing the method types).

3.5.2 Elaboration from $\mathbf{F_D}$ to $\mathbf{F}_{\{\}}$

As both $\mathbf{F_D}$ and $\mathbf{F}_{\{\}}$ are extensions of System F, the elaboration from former to latter is mostly trivial, leaving common features unchanged. The mapping of $\mathbf{F_D}$ dictionaries into $\mathbf{F}_{\{\}}$ records, however, is non-trivial. Briefly, dictionary types are elaborated into record types, as shown in Figure 3.10 (bottom), and dictionaries into record expressions, possibly nested within type and term abstractions and applications, as shown in Figure 3.10 (top).

In particular, a dictionary type, TC, which corresponds to a unique entry $(m : a : \sigma')$ in the class environment Γ_C, elaborates to a record type whose field has the same name as the dictionary type's method name, m, and the type of that field is determined by the elaboration of σ'. A $TC\,\sigma$ dictionary elaborates to a record expression which is surrounded, firstly, by abstractions over type and term variables that arise from the method type's class constraints and, secondly, by type and term applications that properly instantiate those abstractions.

Metatheory. The following theorems confirm that the $\mathbf{F_D}$-to-$\mathbf{F}_{\{\}}$ elaboration is indeed appropriate.

The first theorem states that a well-typed $\mathbf{F_D}$ expression always elaborates to a $\mathbf{F}_{\{\}}$ expression that is also well-typed in the translated context.

Theorem 3.7 (Type Preservation).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$, *then there are unique* Γ *and* σ *such that* $\Gamma \vdash_{tm} e : \sigma$, *where* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Secondly, and more importantly, the dynamic semantics is also preserved by the elaboration.

Theorem 3.8 (Semantic Preservation).
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Sigma \vdash e \longrightarrow^* v$, *then there exists a* v *such that* $\Sigma; \Gamma_C; \bullet \vdash_{tm} v : \sigma \rightsquigarrow v$ *and* $e \simeq v$.

Thirdly, the elaboration is entirely deterministic.

Theorem 3.9 (Determinism).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_1$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_2$, *then* $e_1 = e_2$.

3.5.3 Elaboration Decomposition

An elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$ can always be decomposed into two elaborations through $\mathbf{F}_{\text{D}}$. This intuition is formalized in Theorems 3.10 and 3.11 respectively.

Theorem 3.10 (Elaboration Equivalence - Expressions).
If $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma \vdash^{\text{M}}_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$
where $\vdash^{\text{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash^{\text{M}}_{ty} \tau \rightsquigarrow \sigma$.

Theorem 3.11 (Elaboration Equivalence - Dictionaries).
If $P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma \vDash^{\text{M}} Q \rightsquigarrow d$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e$
where $\vdash^{\text{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Theorem 3.10 also has a type checking mode counterpart. The full proofs can be found in the Appendix Section L of [Bottu et al. 2019].

3.6 Coherence Revisited

As mentioned previously in Section 3.4, the invariant that type class instances do not overlap is crucial in proving Theorem 3.1. This uniqueness property is made explicit in $\mathbf{F}_{\text{D}}$. Our proof thus proceeds by elaborating the $\boldsymbol{\lambda}_{\text{TC}}$ expression to two possibly different $\mathbf{F}_{\text{D}}$ expressions and subsequently elaborating these $\mathbf{F}_{\text{D}}$ expressions to $\mathbf{F}_{\{\}}$ expressions. Consequently, the proof is split in two main steps. The first part is the most involved, where we use a technique based on logical relations to prove that any two $\mathbf{F}_{\text{D}}$ expressions originating from the same $\boldsymbol{\lambda}_{\text{TC}}$ expression are contextually equivalent. The second part proves that the elaboration from $\mathbf{F}_{\text{D}}$ to $\mathbf{F}_{\{\}}$ preserves contextual equivalence. This step follows straightforwardly from the fact that the $\mathbf{F}_{\text{D}}$-to-$\mathbf{F}_{\{\}}$ elaboration is deterministic. Together, these prove that the elaboration from $\boldsymbol{\lambda}_{\text{TC}}$ to $\mathbf{F}_{\{\}}$ through $\mathbf{F}_{\text{D}}$ is coherent. Theorem 3.2 follows from this result, together with Theorem 3.10.

The remainder of this section explains the techniques we used to prove Theorem 3.1 in detail.

3.6.1 Coherent Elaboration from λ_{TC} to $\mathbf{F}_{\mathbf{D}}$

Logical Relations:

Logical relations [Plotkin 1973; Statman 1985; Tait 1967] are key to proving contextual equivalence. In our type system, the logical relation for expressions is mostly standard, though the relation for dictionaries is novel.

Dictionaries. The logical relation over two open dictionaries is defined by means of an auxiliary relation on closed dictionaries. We define this value relation for closed dictionaries as follows. Note that from now on, we will omit elaborations when they are entirely irrelevant.

Definition 3.3 (Value Relation for Dictionaries)**.**
The value relation for two dictionary values, $D\,\overline{\sigma}_j\,\overline{dv}_i$ *and* $D\,\overline{\sigma}_j\,\overline{dv}'_i$*, is defined as:*

$$
\begin{aligned}
&(\Sigma : D\,\overline{\sigma}_j\,\overline{dv}_i, \Sigma' : D\,\overline{\sigma}_j\,\overline{dv}'_i) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}_R \triangleq \\
&\quad \big((\Sigma : dv_i, \Sigma' : dv'_i) \in \mathcal{V}[\![[\overline{\sigma}_j/\overline{a}_j]Q_i]\!]^{\Gamma_C}_R \ ,\ \forall i\big) \\
&\quad \wedge\ \Sigma;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv}_i : R(Q)\ \wedge\ \Sigma';\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv}'_i : R(Q), \\
&\quad \textit{where } (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q').m \mapsto e \in \Sigma \textit{ and } Q = [\overline{\sigma}_j/\overline{a}_j]Q'
\end{aligned}
$$

The value relation is indexed by the dictionary type Q. We require both dictionaries to be well-typed, and their dictionary arguments to be in the value relation as well. The relation has four additional parameters: the contexts Σ and Σ', which annotate the dictionaries, the class environment Γ_C, used in the well-typing condition, and the type substitution R.

In order to define logical equivalence between open dictionaries, we substitute all free variables with closed terms, thus reducing them to closed dictionary values. Three kinds of variables exist (term variables x, type variables a and dictionary variables δ). This results in three separate semantic interpretations of the typing context Γ. The type substitution $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ maps all type variables $a \in \Gamma$ onto closed types. $\phi \in \mathcal{G}[\![\Gamma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_R$ maps each term variable $x \in \Gamma$ to two closed expressions e_1 and e_2 that are logically related (see Definition 3.6), and $\gamma \in \mathcal{H}[\![\Gamma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_R$ maps each dictionary variable $\delta \in \Gamma$ to two logically related dictionary values. We use ϕ_1 and ϕ_2 to denote the substitution for the first and second expression, respectively, and we do the same for γ.

Definition 3.4 (Logical Equivalence for Dictionaries)**.**
We define logical equivalence for two dictionaries, d *and* d'*, as:*
$\Gamma_C; \Gamma \vdash \Sigma : d \simeq_{log} \Sigma' : d' : Q \triangleq$

$$\forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}, \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma,\Sigma',\Gamma_C}, \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma,\Sigma',\Gamma_C} :$$
$$(\Sigma : \gamma_1(\phi_1(R(d))), \Sigma' : \gamma_2(\phi_2(R(d')))) \in \mathcal{V}[\![Q]\!]_R^{\Gamma_C}$$

Two dictionaries d and d' are logically equivalent iff substituting related expressions / dictionaries for their free variables yields related dictinary values.

Expressions. The value relation for expressions is mostly standard, with two notable deviations. Firstly, the relation is defined over two different method environments Σ and Σ'. Hence, both expressions are annotated with their respective environment. Secondly, the dictionary abstraction case is novel.

Definition 3.5 (Value Relation for Expressions)**.**
We define the value relation for two values, v *and* v'*, as:*
$(\Sigma : \mathit{True}, \Sigma' : \mathit{True}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$
$(\Sigma : \mathit{False}, \Sigma' : \mathit{False}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$

$$(\Sigma : v, \Sigma' : v') \in \mathcal{V}[\![a]\!]_R^{\Gamma_C} \triangleq (a \mapsto (\sigma, r)) \in R \ \wedge \ (v, v') \in r$$
$$\wedge \ \Sigma; \Gamma_C; \bullet \vdash_{tm} v : \sigma \ \wedge \ \Sigma'; \Gamma_C; \bullet \vdash_{tm} v' : \sigma$$

$$(\Sigma : v, \Sigma' : v') \in \mathcal{V}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma; \Gamma_C; \bullet \vdash_{tm} v : R(\sigma_1 \to \sigma_2) \ \wedge \ \Sigma'; \Gamma_C; \bullet \vdash_{tm} v' : R(\sigma_1 \to \sigma_2)$$
$$\wedge \forall (\Sigma : e, \Sigma' : e') \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} : (\Sigma : v\, e, \Sigma' : v'\, e') \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C}$$

$$(\Sigma : v, \Sigma' : v') \in \mathcal{V}[\![Q \Rightarrow \sigma]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma; \Gamma_C; \bullet \vdash_{tm} v : R(Q \Rightarrow \sigma) \ \wedge \ \Sigma'; \Gamma_C; \bullet \vdash_{tm} v' : R(Q \Rightarrow \sigma)$$
$$\wedge \forall (\Sigma : dv, \Sigma' : dv') \in \mathcal{V}[\![Q]\!]_R^{\Gamma_C} : (\Sigma : v\, dv, \Sigma' : v'\, dv') \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

$$(\Sigma : v, \Sigma' : v') \in \mathcal{V}[\![\forall a.\sigma]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma; \Gamma_C; \bullet \vdash_{tm} v : R(\forall a.\sigma) \ \wedge \ \Sigma'; \Gamma_C; \bullet \vdash_{tm} v' : R(\forall a.\sigma)$$
$$\wedge \forall \sigma', \forall r \in \mathit{Rel}[\sigma'] : \ \Gamma_C; \bullet \vdash_{ty} \sigma' \Rightarrow (\Sigma : v\, \sigma', \Sigma' : v'\, \sigma') \in \mathcal{E}[\![\sigma]\!]_{R, a \mapsto (\sigma', r)}^{\Gamma_C}$$

Consider the interesting case of dictionary abstraction. The relation requires the terms to be well-typed, and the applications for all related input dictionaries to be in the expression relation $\mathcal{E}$. The definition of this $\mathcal{E}$ relation is as follows:

Definition 3.6 (Expression Relation).
We define the expression relation for two terms, e_1 *and* e_2*, as:*
$(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \triangleq$

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_1 : R(\sigma) \ \wedge \ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_2 : R(\sigma) \ \wedge$$
$$\exists v_1, v_2, \left(\Sigma_1 \vdash e_1 \longrightarrow^* v_1 \ \wedge \ \Sigma_2 \vdash e_2 \longrightarrow^* v_2 \ \wedge \ (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C}\right)$$

In this definition, expressions are reduced to values and those values must be in the value relation. This is well-defined because $\mathbf{F}_\mathbf{D}$ is strongly normalizing (Theorem 3.5).

Finally, we can give the definition of logical equivalence for open expressions:

Definition 3.7 (Logical Equivalence for Expressions).
We define logical equivalence of two expressions, e_1 *and* e_2*, as:*
$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \triangleq$

$$\forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}, \ \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}, \ \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} :$$
$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

Proof of $\boldsymbol{\lambda}_{\mathrm{TC}}$-to-$\mathbf{F}_\mathbf{D}$ Coherence:

With the above definitions we are ready to formally state the metatheory and establish the coherence theorems from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_\mathbf{D}$.

Design Principle of $\mathbf{F}_\mathbf{D}$. We emphasize that $\mathbf{F}_\mathbf{D}$ captures the intention of type class instances. Theorem 3.12 states that any two dictionary values for the same constraint are logically related:

Theorem 3.12 (Value Relation for Dictionary Values).
If $\Sigma_1; \Gamma_C; \bullet \vdash_d dv_1 : Q$ *and* $\Sigma_2; \Gamma_C; \bullet \vdash_d dv_2 : Q$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$*, then* $(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]_\bullet^{\Gamma_C}$.

Note that two environments Σ_1 and Σ_2 are logically equivalent under Γ_C, written $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$, when they contain the same dictionary constructors and the corresponding method implementations are logically equivalent. The full definition can be found in the Appendix Section G.3 of [Bottu et al. 2019].

Coherent Resolution. We now prove that constraint resolution is semantically coherent, that is, if multiple resolutions of the same constraint exist, they are logically equivalent.

Theorem 3.13 (Logical Coherence of Dictionary Resolution).
If $P;\Gamma_C;\Gamma \vDash^{\mathrm{M}} Q \rightsquigarrow d_1$ *and* $P;\Gamma_C;\Gamma \vDash^{\mathrm{M}} Q \rightsquigarrow d_2$
and $\vdash^{\mathrm{M}}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^{\mathrm{M}}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$,
then $\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : Q$, *where* $\Gamma_C;\Gamma \vdash^{\mathrm{M}}_{Q} Q \rightsquigarrow Q$.

Coherent Elaboration. Furthermore, in order to prove that the elaboration from $\boldsymbol{\lambda}_{\mathrm{TC}}$ to $\mathbf{F}_{\mathbf{D}}$ is coherent, we show that all elaborations of the same expression are logically equivalent[3].

Theorem 3.14 (Logical Coherence of Expression Elaboration).
If $P;\Gamma_C;\Gamma \vdash^{\mathrm{M}}_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P;\Gamma_C;\Gamma \vdash^{\mathrm{M}}_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
and $\vdash^{\mathrm{M}}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^{\mathrm{M}}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$,
then $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$, *where* $\Gamma_C;\Gamma \vdash^{\mathrm{M}}_{ty} \tau \rightsquigarrow \sigma$.

Contextual Equivalence. We prove that all logically equivalent expressions are contextually equivalent. Together with Theorem 3.14, this shows coherence of the $\boldsymbol{\lambda}_{\mathrm{TC}}$-to-$\mathbf{F}_{\mathbf{D}}$ elaboration.

We first provide a formal definition of contextual equivalence for $\mathbf{F}_{\mathbf{D}}$ expressions. Kleene equivalence for $\mathbf{F}_{\mathbf{D}}$ is defined similarly to Definition 3.1 and can be found in the Appendix Section I.1 of [Bottu et al. 2019].

Definition 3.8 (Contextual Equivalence for $\mathbf{F}_{\mathbf{D}}$ Expressions).
Two well-typed expressions $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma$ *are contextually equivalent, written* $\Sigma;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$, *if and only if forall* $M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\bullet \Rightarrow Bool)$ *it holds that* $\Sigma : M[e_1] \simeq \Sigma : M[e_2]$.

Theorem 3.15 (Logical Equivalence implies Contextual Equivalence).
If $\Gamma_C;\Gamma \vdash \Sigma : e_1 \simeq_{log} \Sigma : e_2 : \sigma$, *then* $\Sigma;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$.

3.6.2 Deterministic Elaboration from $\mathbf{F}_{\mathbf{D}}$ to $\mathbf{F}_{\{\}}$

Contextual Equivalence:

Similarly to expressions, the elaboration from a $\boldsymbol{\lambda}_{\mathrm{TC}}$ context M to a $\mathbf{F}_{\{\}}$ context M can always be decomposed into two elaborations, through a $\mathbf{F}_{\mathbf{D}}$ context M. The syntax and typing judgments can be found in the Appendix Sections A and F of [Bottu et al. 2019], respectively.

[3]Theorem 3.14 also has a type checking mode counterpart.

We now formally define contextual equivalence for $\mathbf{F}_{\{\}}$ expressions through $\mathbf{F}_{\mathrm{D}}$ contexts.

Definition 3.9 (Contextual Equivalence in $\mathbf{F}_{\mathrm{D}}$ Context)**.**
Two target expressions e_1 *and* e_2 *are contextually equivalent, written* $\Sigma; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$*, if and only if they are well-formed (*$\Gamma \vdash_{tm} e_i : \sigma$ *for* $i = 1, 2$*, and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$*) and* $M[e_1] \simeq M[e_2]$ *holds for any* $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M$*.*

Proof of $\mathbf{F}_{\mathrm{D}}$-to-$\mathbf{F}_{\{\}}$ Coherence:

We continue by proving that contextual equivalence is preserved by the elaboration from $\mathbf{F}_{\mathrm{D}}$ to $\mathbf{F}_{\{\}}$:

Theorem 3.16 (Elaboration preserves Contextual Equivalence)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$*,*
and $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma \rightsquigarrow e_1$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2$
then $\Sigma; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$*.*

Proof of $\boldsymbol{\lambda}_{\mathrm{TC}}$-to-$\mathbf{F}_{\{\}}$ Coherence:

Finally, in order to link back to Theorem 3.2 (which has no notion of $\mathbf{F}_{\mathrm{D}}$), we prove that contextual equivalence with $\mathbf{F}_{\mathrm{D}}$ contexts implies contextual equivalence with $\boldsymbol{\lambda}_{\mathrm{TC}}$ contexts:

Theorem 3.17 ($\mathbf{F}_{\mathrm{D}}$ Contextual Equivalence implies $\boldsymbol{\lambda}_{\mathrm{TC}}$ Contextual Equivalence)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \sigma$*, then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$*,*
where $\vdash^{\mathrm{M}}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \vdash^{\mathrm{M}}_{ty} \tau \rightsquigarrow \sigma$*.*

For clarity, we restate coherence Theorems 3.2 and 3.1:

Theorem 3.2 (Expression Coherence - Restated)**.**
If $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$*,*
then $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$*.*

Theorem 3.2 follows by combining Theorems 3.10, 3.14, 3.15, 3.16 and 3.17.

Theorem 3.1 (Coherence - Restated)**.**
If $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$*,*
then $\Gamma_{C1} = \Gamma_{C2}$*,* $P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$*.*

Finally, we show that Theorem 3.1 follows from Theorem 3.2. The full proofs can be found in the Appendix Section M of [Bottu et al. 2019].

3.7 Discussion of Possible Extensions

As the goal of this work was to find a proof technique to formally establish coherence for type class resolution, a stripped down source calculus was employed in order not to clutter the proof. This section provides a brief discussion of extending our coherence proof to support several mainstream language features.

Ambiguous Type Schemes. As mentioned previously, our work is orthogonal to ambiguous type schemes, which have already been extensively studied by Jones [1993]. We believe our work and the proof by Jones can be combined, which would then relax the restriction of bidirectional type checking, and prove coherence for both ambiguous type schemes and type class resolution.

General Recursion. Recursion is an important feature, present in any real world programming language. It is important to note that, while $\boldsymbol{\lambda}_{\text{TC}}$ does not feature recursion on the expression level (as it does not affect the essence of the coherence proof), type class resolution itself is recursive. Dictionary values are constructed dynamically from a statically given set of dictionary constructors (one constructor per type class instance). The system can thus recursively generate an arbitrary number of dictionaries from a finite set of instances.

Our logical relations can be adapted to support general recursion, through well-known techniques, such as step indexing [Ahmed 2006]. While this results in a significantly longer and more cluttered proof, we do not anticipate any major complications.

Multi-Parameter Type Classes. Just like regular type class instances, multi-parameter instances (as supported by GHC) are subject to the no-overlap rule. Hence, they respect our main assumption. They may indeed give rise to more ambiguity, but this is the kind of ambiguity that is studied by Jones [1993], not the kind that shows up during resolution. Note that functional dependencies were originally introduced by Jones [2000] as a way to resolve the ambiguity caused by multi-parameter instances.

Dependent Types. Dependently typed languages, e.g., Agda [Devriese & Piessens 2011] and Idris [Brady 2013], include language features that are inspired by type classes. Proving resolution coherence in a dependently typed setting requires significant extension of our calculi, as dependent types collapse the term and type levels into a single level and thus enable more powerful type signatures for classes and instances. Furthermore, our logical relation needs to be extended to support dependent types [Bernardy et al. 2012] as well. Fortunately, the essence of our proof strategy still applies. That is, the intermediate language incorporates separate binding structures for dictionaries, and enforces the uniqueness of dictionaries. We thus believe a non-trivial extension of our proof methodology can be used to prove coherence for type class resolution in the setting of dependently typed languages.

Non-overlapping Instances. Our work is built on top of the assumption that type class instances do not overlap. This is enforced during the type checking of instance declarations, and made explicit in the intermediate language. Whether a constraint is entailed directly from an instance, through user provided constraints in a type annotation, or through local evidence, is not actually relevant, as all evidence ultimately has to originate from a non-overlapping instance declaration.

Therefore, our work can be extended to include features where the assumption holds true. This includes, among others, GADT's [P. Jones et al. 2006], implication constraints [Bottu et al. 2017], type constructors, higher kinded types and constraint kinds [Orchard & Schrijvers 2010], e.g. Bottu et al. [2017] informally discuss the coherence of implication constraints based on the same assumption. These features are all included in GHC.

Modules. Modules, as supported by GHC, pose an interesting challenge, as they are known to cause a form of ambiguity.[4] GHC does not statically check the uniqueness of instances across modules, thus indirectly allowing users to write overlapping instances, as long as no ambiguity arises during resolution. Adapting our global uniqueness assumption to accommodate this additional freedom remains an interesting challenge.

Laziness. The operational semantics of the $\mathbf{F}_{\mathrm{D}}$ and $\mathbf{F}_{\{\}}$ calculi in this work are given through standard call-by-name semantics, in order to approximate Haskell's laziness. The system can easily be adapted to either call-by-value or call-by-need, with little impact on the proofs.

[4] `http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/`

It is important to note though, that while expressions are evaluated lazily, type class resolution itself is eager, and constructs the full dictionaries at compile time. This complicates supporting certain GHC features that rely on laziness, like cyclic and infinite dictionaries. They could be supported through loop detection and deferring the construction of dictionaries to runtime, but these would nonetheless pose a significant challenge.

3.8 Related Work

Type Classes. Jones [1993, 1994] formally proves coherence for the framework of qualified types, which generalizes from type classes to arbitrary evidence-backed type constraints. He focuses on nondeterminism in the typing derivation, and assumes that resolution is coherent.

Morris [2014] presents an alternative, denotational semantics for type classes (without superclasses) that avoids elaboration and instead interprets qualified type schemes as the set of denotations of all its monomorphic instantiations that satisfy the qualifiers. The nondeterminism of resolution does not affect these semantics.

Kahl & Scheffczyk [2001] present named type class instances that are not used during resolution, but can be explicitly passed to functions. Nevertheless, they violate the uniqueness of instances, and give rise to incoherence of the form illustrated by our `discern` function in Section 3.1.3.

Unlike most other languages with type classes (such as Haskell, Mercury or PureScript) Coq [Sozeau & Oury 2008] does not enforce the non-overlapping instances condition. Consequently, coherence does not hold for type class resolution in Coq. The reason for this alternative design choice is twofold: (a) Since Coq's type system is more complex than that found in regular programming languages, it is not always possible to decide whether two instances overlap [Lampropoulos & Pierce 2018, Chapter 2: Typeclasses]. (b) Type class members in Coq are often proofs and, unlike for expressions, users are often indifferent to the coherence of proofs (even though from a semantic point of view, Coq differentiates between them). This concept is known as "proof irrelevance" [Gilbert et al. 2019], that is, as long as at least one proof exists, the concrete choice between these proofs is irrelevant. Users can deal with this lack of coherence by either assigning priorities to overlapping instances, or by manually curating the instance database and locally removing specific instances.

Winant & Devriese [2018] introduce explicit dictionary application to the Haskell language, and prove coherence for this extended system. Their proof is

parametric in the constraint entailment judgment and thus assumes that the constraint solver produces "canonical" evidence. They proceed by introducing a disjointness condition to explicitly applied dictionaries, in order to ensure that coherence is preserved by their extension. This chapter proves their aforementioned assumption, by establishing coherence for type class resolution.

Dreyer et al. [2007] blend ML modules with Haskell type class resolution. Unlike Haskell, they feature multiple global (or outer) scopes; instances within one such global scope must not overlap. Moreover, global instances are shadowed by those given through type signatures. While their language has been formalized, no formal proof of coherence is given.

Implicits. Cochis [Schrijvers et al. 2019] is a calculus with highly expressive implicit resolution, including local instances. It achieves coherence by imposing restrictions on the implicit context and enforcing a deterministic resolution process. This allows for a much simpler coherence proof.

OCaml's modular implicits [White et al. 2015] do not enforce uniqueness of "instances" but dynamically ensure coherence by rejecting programs where there are multiple possible resolution derivations. This approach has not been formalized yet.

Other. Reynolds [1991] introduced the notion of coherence in the context of the Forsythe language's intersection types; he proved coherence directly in terms of the denotational semantics of the language.

In contrast, Bi et al. [2018, 2019] consider a setting where subtyping for intersection types is elaborated to coercions. Inspired by Biernacki & Polesiuk [2015], they use an approach based on contextual equivalence and logical relations, which has inspired us in turn. However, they do not create an intermediate language to avoid the problem of a more expressive target language. This leads to a notion of contextual equivalence that straddles two languages and complicates their proofs.

3.9 Scientific Output

The material found in this chapter is drawn from the following publication:

Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. Proc. ACM Program. Lang. 3, ICFP, Article 91 (August 2019), 28 pages. DOI: https://doi.org/10.1145/3341695

The above is multi-author work, where the design of the three calculi presented in this chapter, the structure of the coherence proof and the meta-theory related to $\boldsymbol{\lambda}_{\text{TC}}$ and $\mathbf{F}_{\text{D}}$ were mainly developed by Gert-Jan Bottu, Ningning Xie and Tom Schrijvers. My contribution is situated in the $\mathbf{F}_{\{\}}$ metatheory and the $\mathbf{F}_{\text{D}}$-to-$\mathbf{F}_{\{\}}$ elaboration. Iterations of proofreading the coherence proof were shared between Gert-Jan Bottu, Ningning Xie and myself.

Chapter 4

Cochis: Stable and Coherent Implicits

This chapter presents Cochis: the Calculus Of CoHerent ImplicitS. Cochis is an improved variant of the implicit calculus [Oliveira et al. 2012] that guarantees *coherence* and *stability*. Cochis supports local scoping, overlapping instances, first-class instances and higher-order rules. Yet, in contrast to most previous work that supports these features, the calculus is not only type-safe, but also stable and coherent. Naturally, the unrestricted calculus does not support global uniqueness of instances, since this property depends on the global scoping restriction. Nevertheless, if retaining global uniqueness is desired, that can be modeled by the subset of Cochis without local declarations. In fact, GHC has been recently extended with *quantified class constraints* [Bottu et al. 2017; GHC 2017b], based on the resolution algorithm of an early version of Cochis. Quantified class constraints enable higher-order rules in a setting with global scoping.

Ensuring coherence and stability in the presence of Cochis's overlapping and higher-order rules is particularly challenging. We introduce a logical formulation of how to resolve implicits, which is simple but ambiguous and incoherent, and a second formulation, which is less simple but unambiguous and coherent. Every resolution of the second formulation is also a resolution of the first, but not conversely. Parts of the second formulation bear a close resemblance to a standard technique for proof search in logic called *focusing* [Andreoli 1992; Liang & Miller 2009; Miller et al. 1991]. However, unlike focused proof search, which is still essentially non-deterministic, Cochis's resolution employs additional techniques to be entirely deterministic and thus obviously coherent.

In particular, unlike focused proof search, our resolution uses a stack discipline to prioritize implicits, and removes any recursive resolutions from matching decisions. Moreover, further restrictions are needed to obtain stability.

In summary, our contributions are as follows:

- We present COCHIS, a *coherent*, *stable* and *type-safe* formal model for implicit programming that supports local scoping, overlapping implicits, first-class implicits and higher-order rules.
- We significantly improve the design of resolution over the existing work on the implicit calculus by Oliveira et al. [2012]. The new design for resolution is more powerful and expressive; it is closely based on principles of logic and the idea of propositions as types [Wadler 2015]; and is related to the idea of focusing in proof search.
- COCHIS comes with a semantics in the form of a type-directed elaboration to System F.
- We provide a unification-based algorithm as an executable form of the declarative specification of resolution. A prototype implementation of this algorithm and of COCHIS is available at `https://bitbucket.org/tom_schrijvers/cochis/`.
- We establish key meta-theoretic properties of our system:
  - The elaboration is type-preserving.
  - Resolution is deterministic and thus obviously coherent.
  - Resolution is stable under type substitution, and type-directed elaboration is preserved by reduction of type application.
  - Our algorithm is sound and complete with respect to its declarative specification.

  A sketch of the proofs is available in Appendix B of this thesis. The full proofs are available in the appendix of [Schrijvers et al. 2019] and at `https://bitbucket.org/KlaraMar/cochiscoq`.

Organization Section 4.1 presents an informal overview of our calculus. Sections 4.2 and 4.3 describe a polymorphic type system that statically excludes ill-behaved programs. Section 4.4 provides the elaboration semantics of our calculus into System F and correctness results. Section 4.5 discusses several of our design choices as well as some alternatives. Section 4.6 discusses related work and Section 4.7 concludes.

4.1 Overview

This section introduces the key features of COCHIS and discusses its stability and coherence properties.

4.1.1 An Overview of Cochis

Like Haskell, our calculus COCHIS guarantees stability and coherence and, like Scala, it permits nested/overlapping declarations, and does not guarantee global uniqueness. COCHIS improves upon the implicit calculus [Oliveira et al. 2012] by having stability and a better, more expressive design for resolution. Like the implicit calculus, the primary goal of COCHIS is to model *implicit resolution* and the *scoping* of implicit values used by resolution. Next we iterate over the key constructs and features of COCHIS.

Fetching Values by Type A central construct in COCHIS is a query. Queries allow values to be fetched by type, not by name. For example, in the following function call

$$foo\ ?Int$$

the query $?Int$ looks up a value of type Int in the implicit environment, to serve as an actual argument. Note that queries in COCHIS play the same role as the keyword `implicitly` in Scala, shown in Section 2.3.

Constructing Values with Type-Directed Rules COCHIS constructs values, using programmer-defined, type-directed rules (similar to functions). A rule (or rule abstraction) defines how to compute, from implicit arguments, a value of a particular type. For example, here is a rule that given an implicit Int value, adds one to that value:

$$\lambda_? Int.?Int + 1$$

The rule abstraction syntax resembles a traditional λ expression. However, instead of having a variable as argument, a rule abstraction ($\lambda_?$) has a type as argument. The type argument denotes the availability of a value of that type (in this case Int) in the implicit environment inside the body of the rule abstraction. Thus, queries over the rule abstraction type argument inside the rule body will succeed.

The type of the rule above is the *rule type* $Int \Rightarrow Int$. This type denotes that the rule has type Int provided a value of type Int is available in the implicit

environment. The implicit environment is extended through rule application (analogous to extending the environment with function applications). Rule application is expressed as, for example:

$$(\lambda_? Int.?Int + 1) \textbf{ with } 1$$

With syntactic sugar similar to a **let**-expression, a rule abstraction-application combination is more compactly denoted as:

$$\textbf{implicit } 1 \textbf{ in } (?Int + 1)$$

Both expressions return 2.

The analog to rule abstractions in Scala are functions with arguments marked with the **implicit** keyword. However, in older versions of Scala, functions with implicit arguments were not first class and could not be abstracted over. In particular in older versions of Scala it was impossible to express the type of a function with an implicit argument. Recent versions of Scala, partly inspired by the implicit calculus, generalize the mechanism of implicits and make rule abstractions and types first class too, by what they call *implicit function types* [Odersky et al. 2017].

Higher-Order Rules COCHIS supports higher-order rules. For example, the rule

$$\lambda_? Int.\lambda_?(Int \Rightarrow Int \times Int).?(Int \times Int)$$

when applied, will compute an integer pair given an integer and a rule to compute an integer pair from an integer. This rule is higher-order because another rule (of type $Int \Rightarrow Int \times Int$) is used as an argument. The following expression returns $(3, 4)$:

$$\begin{array}{l}(\lambda_? Int.\lambda_?(Int \Rightarrow Int \times Int).?(Int \times Int)) \\ \quad \textbf{with } 3 \\ \quad \textbf{with } (\lambda_? Int.(?Int, ?Int + 1))\end{array}$$

Recursive Resolution Note that resolving the query $?(Int \times Int)$ above involves multiple implicits. The current environment does not contain the required integer pair. It does however contain the integer 3 and a rule $\lambda_? Int.(?Int, ?Int + 1)$ to compute a pair from an integer. Hence, the query is resolved with $(3, 4)$, the result of applying the pair-producing rule to 3.

Polymorphic Implicits and Queries COCHIS features explicit polymorphism. For example, the rule $\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))$ abstracts over a type using standard type abstraction and then uses a rule abstraction to provide a value of type α in the implicit environment of the rule body. This rule has type $\forall\alpha.\alpha \Rightarrow \alpha \times \alpha$ and can be instantiated to multiple rules of monomorphic types $Int \Rightarrow Int \times Int$, $Bool \Rightarrow Bool \times Bool$, ...

Multiple monomorphic queries can be resolved by the same polymorphic rule. The following expression returns $((3,3),(True, True))$:

$$\begin{array}{l}\textbf{implicit } 3 \textbf{ in} \\ \quad \textbf{implicit } True \textbf{ in} \\ \quad\quad \textbf{implicit } (\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))) \textbf{ in} \\ \quad\quad\quad (?(Int \times Int), ?(Bool \times Bool))\end{array}$$

Queries can be polymorphic too. For instance, the following example extracts the polymorphic implicit with a polymorphic query.

$$\textbf{implicit } (\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))) \textbf{ in } ?(\forall\beta.\beta \Rightarrow (\beta \times \beta)))$$

In practice, polymorphic queries are useful in combination with higher-kinded types where they occur as recursive resolvents of polymorphic rules. We cannot illustrate this with COCHIS as, to keep its definition small, it is not equipped with higher-kinded types. The interested reader can find examples in the work of Bottu et al. [2017].

Type-Directed Synthesis of Simple Programs One interesting feature of COCHIS is that it can synthesize simple programs of a given type. This feature can potentialy be useful when there is at most one possible implementation of a program. For example a variant of the polymorphic identity function could be synthesized with the following query:

$$?(\forall\alpha.\alpha \Rightarrow \alpha)$$

We have not yet explored the usefulness of this feature for practical programming in depth, but this seems to be related to programming with typed holes [Norell 2009]. Perhaps such a feature (or some extension of it) can be useful to automatically synthesize implementations for typed holes.

Combining Higher-Order and Polymorphic Rules The rule:

$$\lambda_? Int.\lambda_?(\forall\alpha.\alpha \Rightarrow \alpha \times \alpha).(?((Int \times Int) \times (Int \times Int)))$$

prescribes how to build a pair of integer pairs, inductively from an integer value, by consecutively applying the rule of type $\forall\alpha.\alpha \Rightarrow \alpha \times \alpha$ twice: first to an integer, and again to the result (an integer pair). For example, the following expression returns $((3,3),(3,3))$:

$$\textbf{implicit } 3 \textbf{ in implicit } (\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))) \textbf{ in } ?((\mathit{Int}\times\mathit{Int})\times(\mathit{Int}\times\mathit{Int}))$$

Locally and Lexically Scoped Implicits Implicits can be nested and resolution respects their lexical scope. Consider the following program:

$$\begin{array}{l}\textbf{implicit } 1 \textbf{ in}\\ \quad\textbf{implicit } \mathit{True} \textbf{ in}\\ \quad\quad\textbf{implicit } (\lambda_?\mathit{Bool}.\ \textbf{if } ?\mathit{Bool} \textbf{ then } 2 \textbf{ else } 0) \textbf{ in}\\ \quad\quad\quad ?\mathit{Int}\end{array}$$

The query $?\mathit{Int}$ is not resolved with the integer value 1. Instead the rule that returns an integer from a boolean is applied to the boolean *True*, because that rule can provide an integer value and it is nearer to the query. So, the program returns 2 and not 1.

Care with Reduction Observe that some care is required with the inlining of **let**-bindings and other refactorings that perform variable substitutions. Consider for instance the following program.

$$\begin{array}{l}\textbf{implicit } 1 \textbf{ in}\\ \quad\textbf{let } x = ?\mathit{Int} \textbf{ in}\\ \quad\quad\textbf{implicit } 2 \textbf{ in}\\ \quad\quad\quad x\end{array}$$

One might inline x and obtain the following.

$$\begin{array}{l}\textbf{implicit } 1 \textbf{ in}\\ \quad\textbf{implicit } 2 \textbf{ in}\\ \quad\quad ?\ \mathit{Int}\end{array}$$

But this would change its meaning: the first program returns 1, while the second returns 2. The solution is to never consider a term as a candidate for substitution until all of its implicits have been resolved.

This situation is not much different from that of regular **let**-bindings. For example, if we naively inline y in the following program:

$$\begin{array}{l}\textbf{let } x = 1 \textbf{ in}\\ \quad\textbf{let } y = x \textbf{ in}\end{array}$$

$$\begin{array}{l}\textbf{let } x = 2 \textbf{ in} \\ \quad y\end{array}$$

we get

$$\begin{array}{l}\textbf{let } x = 1 \textbf{ in} \\ \quad \textbf{let } x = 2 \textbf{ in} \\ \quad\quad x\end{array}$$

The second program clearly has a different meaning, since the variable x is now captured by the second binder rather than the first. While this problem can be remedied by renaming the second binder's variable, there is no analogous solution for implicits.

Encoding Simple Type Classes in Cochis

A simple form of type classes can be encoded in Cochis similarly to how type classes can be encoded in Scala. In this section we briefly (and informally) illustrate the encoding using examples. The simple encoding presented here does not deal with *superclasses*. We discuss superclasses in Section 4.5.

To help with readability, we assume a few convenient source language features not available in Cochis (which is designed as a formal core calculus rather than a full-fledged source language). In particular, Cochis has no type-inference and requires explicit rule applications and queries. The design of a source language that supports type-inference, implicit rule applications, implicit polymorphism and that translates into a Cochis-like calculus was already explored in previous work on the implicit calculus [Oliveira et al. 2012]. To better illustrate some of our examples here, we will assume such source language features.

Next, we show how to encode the `Ord` (without superclasses), `Show` and `Read` examples from Section 2.2. Firstly, we use type synonyms to allow us to give a short name to a type. Secondly, we use records. Using both of those constructs, the three type classes can be declared as:

$$\begin{array}{l}\textbf{type } \mathit{Ord}\ \alpha = \{\mathit{le} : \alpha \to \alpha \to \mathit{Bool}\} \\ \textbf{type } \mathit{Show}\ \alpha = \{\mathit{show} : \alpha \to \mathit{String}\} \\ \textbf{type } \mathit{Read}\ \alpha = \{\mathit{read} : \mathit{String} \to \alpha\}\end{array}$$

Similarly to the Scala encoding in Section 2.3, we define a *cmp* function that makes the argument of type $\mathit{Ord}\ \alpha$ implicit:

$$\textbf{let } \mathit{cmp} : \forall\alpha.\mathit{Ord}\ \alpha \Rightarrow \alpha \to \alpha \to \mathit{Bool} = (?(\mathit{Ord}\ \alpha)).\mathit{le} \textbf{ in}$$

Here the query is annotated with $Ord\ \alpha$ triggering the resolution of a value of that type. Once that value is computed, the field *le* can be extracted from it.

The "instances" of *Ord* can be defined as record values or rule types returning an *Ord* record.

$$\begin{array}{l}\textbf{let}\ ordInt : Ord\ Int = \{\, le = \lambda x.\ \lambda y.\ primIntLe\ x\ y \,\}\ \textbf{in}\\ \textbf{let}\ ordChar : Ord\ Char = \{\, le = \lambda x.\ \lambda y.\ primCharLe\ x\ y \,\}\ \textbf{in}\\ \textbf{let}\ ordPair : \forall\alpha\ \beta.\ Ord\ \alpha \Rightarrow Ord\ \beta \Rightarrow Ord\ (\alpha, \beta) = \{\, le = \lambda x.\ \lambda y.\\ \quad cmp\ (fst\ x)\ (fst\ y) \wedge ((\neg\ (cmp\ (fst\ y)\ (fst\ x))) \vee cmp\ (snd\ x)\ (snd\ y)) \,\}\\ \textbf{in}\ ...\end{array}$$

Here the first two values denote instances for the base types *Int* and *Char*. The instance for pairs (*ordPair*) has two constraints ($Ord\ \alpha$ and $Ord\ \beta$), and those constraints are implicitly used by *cmp*.

Given a *sort* function:

$$\textbf{let}\ sort : \forall\alpha.Ord\ \alpha \Rightarrow List\ \alpha \to List\ \alpha = ...$$

we can now use **implicit** to introduce the "instances" into the implicit scope and have the $Ord\ (List\ Int)$ argument of the call $sort\ [(3, \texttt{'a'}), (2, \texttt{'c'}), (3, \texttt{'b'})]$ automatically inferred:

$$\begin{array}{l}\textbf{implicit}\ ordInt\ \textbf{in}\\ \quad\textbf{implicit}\ ordChar\ \textbf{in}\\ \quad\quad\textbf{implicit}\ ordPair\ \textbf{in}\\ \quad\quad\quad sort\ [(3, \texttt{'a'}), (2, \texttt{'c'}), (3, \texttt{'b'})]\end{array}$$

4.1.2 Overlapping Implicits and Stability in Cochis

As previously shown, the lexical scope imposes a natural precedence on implicits that ensures coherence. This precedence means that the lexically nearest implicit is used to resolve a query, and not necessarily the most specific implicit. For instance, the following Cochis variation on the *trans* example from Section 2.2.1

$$\begin{array}{l}\textbf{implicit}\ (\lambda n.n + 1 : Int \to Int)\ \textbf{in}\\ \quad\textbf{implicit}\ (\lambda x.x : \forall\alpha.\alpha \to \alpha)\ \textbf{in}\\ \quad\quad ?(Int \to Int)\ 3\end{array}$$

yields the result 3 as the inner identity implicit has precedence over the more specific incrementation implicit in the outer scope. Yet, it is not always possible to statically select the nearest matching implicit. Consider the program fragment

$$
\begin{array}{l}
\textbf{let } \mathit{bad} : \forall\beta.\beta \to \beta = \\
\quad \textbf{implicit } (\lambda x.x : \forall\alpha.\alpha \to \alpha) \textbf{ in} \\
\quad\quad \textbf{implicit } (\lambda n.n + 1 : \mathit{Int} \to \mathit{Int}) \textbf{ in} \\
\quad\quad\quad ?(\beta \to \beta)
\end{array}
$$

Here we cannot statically decide whether $\mathit{Int} \to \mathit{Int}$ matches $\beta \to \beta$: it depends on whether β is instantiated to *Int* or not.

One might consider to force the matter by picking the lexically nearest implicit that matches all possible instantiations of the query, e.g., $\forall\alpha.\alpha \to \alpha$ in the example. While this poses no threat to type safety, this approach is nevertheless undesirable for two reasons. Firstly, it makes the behaviour of programs harder to predict, and, secondly, the behaviour of programs is not *stable under inlining*. Consider the call *bad Int* 3, which would yield the result 3. If instead we inline the function definition of *bad* at the call site and substitute the type argument, we obtain the specialised program

$$
\begin{array}{l}
\textbf{implicit } (\lambda x.x : \forall\alpha.\alpha \to \alpha) \textbf{ in} \\
\quad \textbf{implicit } (\lambda n.n + 1 : \mathit{Int} \to \mathit{Int}) \textbf{ in} \\
\quad\quad ?(\mathit{Int} \to \mathit{Int})\ 3
\end{array}
$$

Now $\mathit{Int} \to \mathit{Int}$ is the nearest lexical match and the program yields the result 4. Consequently, inlining definitions changes their behavior. To avoid this unpredictable behaviour, COCHIS rejects such unstable matchings. Technically speaking the key property that COCHIS guarantees is *stability of resolution* (see also Section 4.4). Essentially this property ensures that type instantiation does not affect the resolution of queries. That is, if some type variables appear free in a query that resolves, then, after instantiating any or all of those type variables, the query still resolves in the same way, i.e., using the same implicits. If it cannot be statically guaranteed that resolution behaves in the same way for *every* instantiation, then the program is rejected. The benefit of rejecting such potentially unstable programs is that the principle of substituting equals for equals is not affected by the interaction between resolution and instantiation. This makes reasoning about programs and code refactoring more predictable.

4.2 The Cochis Calculus

This section formalizes the syntax and type system of COCHIS, while Section 4.4 formalises the type-directed translation to System F. To avoid duplication and ease reading, we present the type system and type-directed translation together, using grey boxes to indicate which parts of the rules belong to the type-directed translation. These greyed parts can be ignored in this section and will be explained in the next.

Types	ρ	$::=$	$\alpha \mid \rho_1 \to \rho_2 \mid \forall\alpha.\rho \mid \rho_1 \Rightarrow \rho_2$
Monotypes	σ	$::=$	$\alpha \mid \sigma \to \sigma$
Expressions	e	$::=$	$x \mid \lambda(x:\rho).e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e\,\sigma \mid ?\rho \mid \lambda_?\rho.e \mid e_1$ **with** e_2
Type Environments	Γ	$::=$	$\epsilon \mid \Gamma, x:\rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x$

Figure 4.1: COCHIS syntax

4.2.1 Syntax

Figure 4.1 shows the syntax of the calculus. Types ρ comprise four constructs: type variables α; function types $\rho_1 \to \rho_2$; universal types $\forall\alpha.\rho$; and the novel *rule* types $\rho_1 \Rightarrow \rho_2$. In a rule type $\rho_1 \Rightarrow \rho_2$, type ρ_1 is called the *context* and type ρ_2 the *head*, following Haskell's terminology for type class instances. Observe that COCHIS types are similar to Haskell's type schemes with the exception that contexts are types rather than type class constraints. We also define a subset of types, called *monotypes* σ, which, like Haskell's monotypes, do not contain universal quantifiers or rule types.

Expressions e include three abstraction-elimination pairs. The form $\lambda(x : \rho).e$ abstracts over a value of type ρ in expression e, which can refer to it with variable x; the abstraction is eliminated by application $e_1\, e_2$. Similarly, $\Lambda\alpha.e$ abstracts over a type in expression e, which can refer to it with type variable α; the abstraction is eliminated by *predicative* type application $e\,\sigma$ (see Section 4.2.5). Finally, $\lambda_?\rho.e$ abstracts over implicit values of type ρ in expression e, which can refer to it with implicit query $?\rho$; the abstraction is eliminated by implicit application e_1 **with** e_2. For convenience we adopt the Barendregt [1985] convention, that variables in binders are distinct, throughout this chapter.

Using implicit abstraction and implicit application we can build the **implicit** sugar used in Section 2.3.

$$\textbf{implicit}\ e_1 : \rho\ \textbf{in}\ e_2 \stackrel{\mathsf{def}}{=} (\lambda_?\rho.e_2)\ \textbf{with}\ e_1$$

For brevity, we keep the COCHIS calculus small. Examples may use additional syntax such as built-in integer and boolean literals and types, and operators.

4.2.2 Type System

Figure 4.2 presents the static type system of COCHIS. Our language is based on predicative System F, which is included in our system.

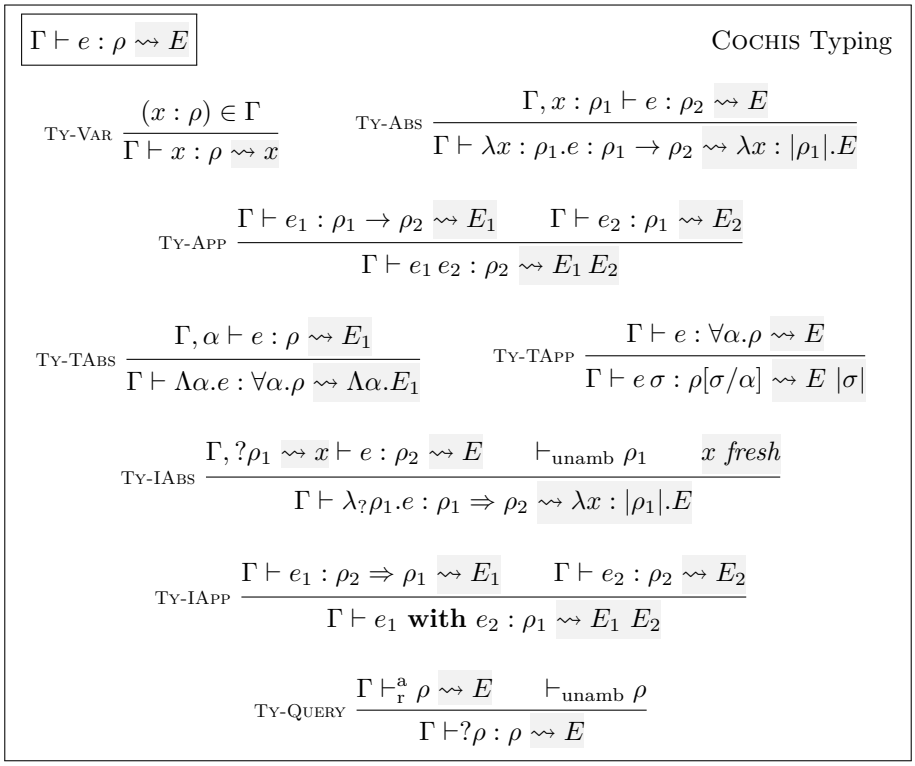


Figure 4.2: Type System and Type-directed Translation to System F

As shown in Figure 4.1, a type environment Γ records type variables α and variables x with associated types ρ that are in scope, like in predicative System F. New here is that it also records instances of implicits $?\rho$.

A typing judgment $\Gamma \vdash e : \rho$ holds if expression e has type ρ with respect to type environment Γ. The first five rules copy the corresponding predicative System F rules; only the last three deserve special attention. Firstly, rule Ty-IAbs extends the implicit environment with the type of an implicit instance. The side condition $\vdash_{\text{unamb}} \rho_1$ states that the type ρ_1 must be unambiguous; we explain this concept in Section 4.2.5. Secondly, rule Ty-IApp eliminates an implicit abstraction by supplying an instance of the required type. Finally, rule Ty-Query resolves a given type ρ against the implicit environment. Again, a side-condition states that ρ must be unambiguous. Resolution is defined in terms of the auxiliary judgment $\Gamma \vdash^{\text{a}}_{\text{r}} \rho$, which is explained next.

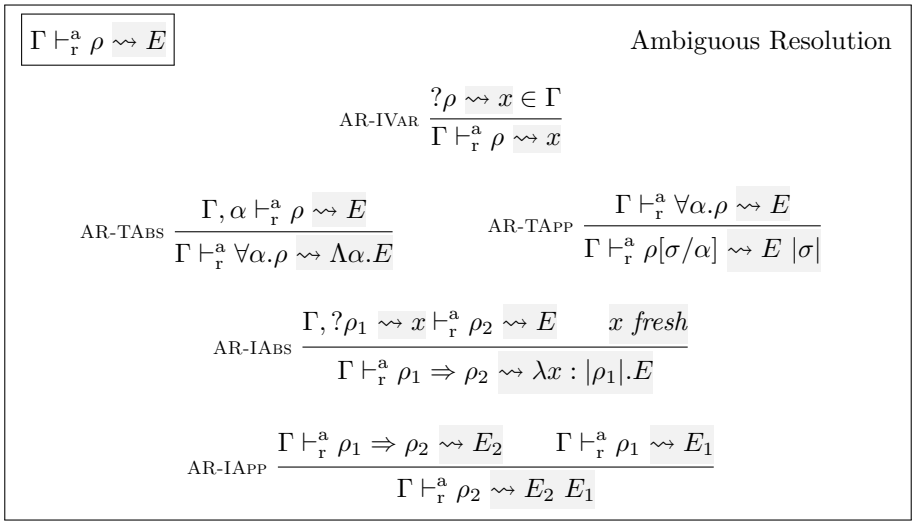


Figure 4.3: Ambiguous Resolution

4.2.3 Resolution

Figure 4.3 provides a first, ambiguous definition of the resolution judgment. Its underlying principle is resolution in logic. Intuitively, $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho$ holds if Γ entails ρ, where the types in Γ and ρ are read as propositions, r stands for resolution and a for ambiguous. Following the "Propositions as Types" correspondence [Wadler 2015], we read α as a propositional variable and $\forall\alpha.\rho$ as universal quantification. Yet, unlike in the traditional interpretation of types as propositions, we have two forms of arrows, function types $\rho_1 \to \rho_2$ and rule types $\rho_1 \Rightarrow \rho_2$, and the twist is that we choose to treat only rule types as implications, leaving function types as uninterpreted predicates.

Unfortunately, the definition in Figure 4.3 suffers from two problems. Firstly, it is *not syntax-directed*; several of the inference rules have overlapping conclusions. Hence, a deterministic resolution algorithm is non-obvious. Secondly and more importantly, the definition is *ambiguous*: a type can be derived in multiple different ways. As an example of both issues, consider that under the environment

$$\Gamma_0 = ?\mathit{Int}, ?\mathit{Bool}, ?(\mathit{Bool} \Rightarrow \mathit{Int})$$

there are two different derivations for resolving $\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}$:

$$\text{AR-IVar}\ \frac{?\mathit{Int} \in \Gamma_0}{\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}}$$

and

$$\textsc{ar-iApp}\ \frac{\textsc{ar-iVar}\ \dfrac{?(\mathit{Bool} \Rightarrow \mathit{Int}) \in \Gamma_0}{\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Bool} \Rightarrow \mathit{Int}} \qquad \textsc{ar-iVar}\ \dfrac{?\mathit{Bool} \in \Gamma_0}{\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Bool}}}{\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}}$$

This example illustrates the first issue; in particular the inference rules AR-IVar and AR-IApp overlap as both can be used to conclude $\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}$. It also shows the second issue as there are two fully different derivation trees for $\Gamma_0 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}$. While this may seem harmless at the type-level, at the value-level each derivation corresponds to a (possibly) different value. Hence, ambiguous resolution may render the meaning of a program ambiguous. In other words, if both resolutions are allowed then the semantics is not coherent.

We next address these two issues one by one. Readers who are keen to see the end result may wish to skip the gradual developments and jump straight to Section 4.2.5.

4.2.4 Type-Directed Resolution with Focusing

To obtain a type-directed formulation of resolution, we adopt a solution from proof search known as *focusing* [Andreoli 1992]. This solution makes sure that only one inference rule applies at any given point and thereby rules out gratuitous forms of nondeterminism.

As an example of such gratuitous nondeterminism consider the following two ways of resolving α given $\Gamma = \alpha, ?\alpha \rightsquigarrow x$:

$$\textsc{ar-iVar}\ \frac{?\alpha \rightsquigarrow x \in \Gamma}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow x}$$

versus

$$\textsc{ar-iApp}\ \frac{\textsc{ar-iAbs}\ \dfrac{\textsc{ar-iVar}\ \dfrac{?\alpha \rightsquigarrow y \in (\Gamma, ?\alpha \rightsquigarrow y)}{\Gamma, ?\alpha \rightsquigarrow y \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow y}}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \Rightarrow \alpha \rightsquigarrow \lambda y.y} \qquad \textsc{ar-iVar}\ \dfrac{?\alpha \rightsquigarrow x \in \Gamma}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow x}}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow (\lambda y.y)\, x}$$

While these are two different proofs, they use the information in the context Γ in essentially the same way. They are β-equivalent and infer System F expressions that are also semantically equivalent. Hence, unlike the nondeterminism in the previous example at the end of Section 4.2.3, where the context provides two ways of resolving the query, this form of nondeterminism serves no purpose.

$$
\begin{array}{lrcl}
\text{Context Types} & \rho & ::= & \forall\alpha.\rho \mid \rho_1 \Rightarrow \rho_2 \mid \tau \\
\text{Simple Types} & \tau & ::= & \alpha \mid \rho_1 \to \rho_2
\end{array}
$$

Figure 4.4: Context types and simple types

$$\Sigma ::= \epsilon \mid \rho \rightsquigarrow x, \Sigma$$

$\boxed{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E}$ Focusing

$$\textsc{FR-TAbs}\ \frac{\Gamma, \alpha \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\forall\alpha.\rho] \rightsquigarrow \Lambda\alpha.E} \qquad \textsc{FR-IAbs}\ \frac{x\ \mathit{fresh} \quad \Gamma, ?\rho_1 \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho_2] \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$\textsc{FR-Simp}\ \frac{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho'] \rightsquigarrow E' \ \ (\forall \rho' \in \bar{\rho}') \qquad ?\rho \rightsquigarrow x \in \Gamma \qquad \Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \bar{\rho}' \rightsquigarrow \bar{x}'; \tau \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\tau] \rightsquigarrow E[\bar{E}'/\bar{x}']}$$

$\boxed{\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$ Matching

$$\textsc{FM-Simp}\ \frac{}{\Gamma; [\tau] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \epsilon; \tau \rightsquigarrow E}$$

$$\textsc{FM-TApp}\ \frac{\Gamma; [\rho[\sigma/\alpha]] \rightsquigarrow E\,|\sigma| \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}{\Gamma; [\forall\alpha.\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$$

$$\textsc{FM-IApp}\ \frac{\Gamma; [\rho_2] \rightsquigarrow E\,x \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad x\ \text{fresh}}{\Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \rho_1 \rightsquigarrow x, \Sigma; \tau \rightsquigarrow E'}$$

Figure 4.5: Focusing Resolution

We will see that focusing provides a straitjacket that eliminates the gratuitous nondeterminism and allows only the first and more direct of these two proofs. In fact, it allows only derivations in β-reduced and η-expanded form.

The focusing approach refines the grammar of types to distinguish a special class of *simple* types, as shown in Figure 4.4. Observe that simple types τ are those types that do not have corresponding pairs of introduction and elimination rules in the ambiguous resolution judgment.

The definition of resolution with focusing that uses this refined grammar is given in Figure 4.5. The main *focusing* judgment $\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E$ is defined with the help of the auxiliary *matching* judgment $\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'$. Both definitions are syntax-directed on the type ρ enclosed in square brackets.

The focusing judgment $\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E$ focuses on the type ρ that is to be resolved – we call this type the "goal". There are three rules, for the three possible syntactic forms of ρ. Rules FR-IAbs and FR-TAbs decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple type τ, Rule FR-Simp selects an implicit ρ from the environment Γ to discharge it. The selected type must *match* the goal, a notion that is captured by the auxiliary judgment. Matching gives rise to a sequence Σ of new (and hopefully simpler) goals that are resolved recursively.

The matching judgment $\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'$ focuses on the selected implicit ρ and checks whether it matches the simple goal τ; informally it captures that ρ can be instantiated to $\Sigma \Rightarrow \tau$. Again, there are three rules for the three possible forms the rule can take. Rule FM-TApp handles universal quantification by instantiating the quantified variable α in a way that recursively yields a match. Rule FM-IApp handles a rule type $\rho_1 \Rightarrow \rho_2$ by recursively checking whether ρ_2 matches the goal. At the same time it yields a new goal ρ_1 which needs to be resolved in order for the rule to apply. Finally, rule FM-Simp expresses the base case where the axiom is identical to the goal and there are no new goals.

This type-directed formulation of entailment reduces the number of proofs for a given goal. For instance, for the example above there is only one proof:

$$\textsc{fr-Simp}\ \frac{?\alpha \rightsquigarrow x \in \Gamma \qquad \textsc{fm-Simp}\ \Gamma; [\alpha] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \epsilon; \alpha \rightsquigarrow x}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\alpha] \rightsquigarrow x}$$

4.2.5 Deterministic and Stable Resolution

While focusing provides a syntax-directed definition of resolution, it does not make resolution entirely deterministic. There are still two sources of nondeterminism: 1) the *ambiguous* instantiation of type variable α with a monotype σ in rule FM-TApp, and 2) nondeterministic selection of an implicit ρ from the type environment Γ in rule FR-Simp. This section eradicates those two remaining sources of nondeterminism to obtain an entirely deterministic formulation of resolution. On top of that, it imposes an additional *stability* condition to make resolution "super"-deterministic: resolution is preserved under type substitution. First, though, we point out that our choice for predicative instantiation has pre-emptively avoided a further source of nondeterminism.

Predicative Instantiation

We have restricted COCHIS to predicative instantiation, i.e., type variables can only be instantiated with monotypes σ. Impredicativity is a known source of nondeterminism in other settings like type inference for the polymorphic λ-calculus [Boehm 1985; Pfenning 1993]. It causes similar problems for COCHIS, in the rules AR-TAPP and FM-TAPP for ambiguous and focusing resolution that choose an instantiation of a type variable.

To see why the impredicative instantiation in those rules causes nondeterminism, consider the two ways of resolving $\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}$ against the environment $\Gamma_1 = ?(\forall\alpha.\alpha \Rightarrow \alpha)$:[1]

$$\textsc{AR-TApp}\ \dfrac{\textsc{AR-IVar}\ \dfrac{?(\forall\alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.\alpha \Rightarrow \alpha}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}}$$

and

$$\textsc{AR-TApp}\ \dfrac{\textsc{AR-IApp}\ \dfrac{\textsc{AR-TApp}\ \dfrac{\textsc{AR-IVar}\ \dfrac{?(\forall\alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.\alpha \Rightarrow \alpha}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} (\forall\beta.\beta \Rightarrow \beta) \Rightarrow (\forall\beta.\beta \Rightarrow \beta)} \qquad \textsc{AR-IVar}\ \dfrac{?(\forall\beta.\beta \Rightarrow \beta) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\beta.\beta \Rightarrow \beta}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\beta.\beta \Rightarrow \beta}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}}$$

The first proof only involves the instantiation of α with *Int*. Yet, the second proof contains an impredicative instantiation of α with $\forall\beta.\beta \Rightarrow \beta$.

We adopt the standard solution from the outset, which only allows predicative instantiation and thus only accepts the first of the two derivations above.

Observe that we not only forbid instantiation with universally quantified types $\forall\alpha.\rho$, but also with rule types $\rho_1 \Rightarrow \rho_2$. The latter are also a source of ambiguity. Consider for instance resolving *Int* in the environment $\Gamma_2 = ?(\forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha), ?\mathit{Bool}, ?((\mathit{Bool} \Rightarrow \mathit{Int}) \to (\mathit{Bool} \to \mathit{Int})), ?(\mathit{Int} \to \mathit{Int})$. There is one derivation that involves instantiating the first entry's α with a monotype, namely with *Int*:

$$\textsc{AR-IApp}\ \dfrac{\textsc{AR-TApp}\ \dfrac{\textsc{AR-IVar}\ \dfrac{?(\forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha) \in \Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} (\mathit{Int} \to \mathit{Int}) \Rightarrow \mathit{Int}} \qquad \textsc{AR-IVar}\ \dfrac{?(\mathit{Int} \to \mathit{Int}) \in \Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \to \mathit{Int}}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}}$$

[1] For the sake of compactness the example uses the ambiguous definition of resolution. Similarly problematic examples can be created for the focusing-based definition.

However, instantiation with the non-monotype $Bool \Rightarrow Int$ also yields a derivation; for conciseness, we have abbreviated $(Bool \Rightarrow Int)$ to BI.

$$\dfrac{\dfrac{\dfrac{\dfrac{?(\forall\alpha.(\alpha\to\alpha)\Rightarrow\alpha)\in\Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.(\alpha\to\alpha)\Rightarrow\alpha}\,\text{AR-IVar}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} (BI\to BI)\Rightarrow BI}\,\text{AR-TApp} \qquad \dfrac{?(BI\to BI)\in\Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} BI\to BI}\,\text{AR-IVar}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} Bool\Rightarrow Int}\,\text{AR-IApp} \qquad \dfrac{?Bool\in\Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} Bool}\,\text{AR-IVar}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} Int}\,\text{AR-IApp}$$

By restricting ourselves to instantiation with monotypes, we disallow the second derivation and thus avoid this source of ambiguity.

Non-Ambiguity Constraints

Rule FM-TApp does not explain how the predicative substitution $[\sigma/\alpha]$ for the type $\forall\alpha.\rho$ should be obtained. At first sight it seems that the choice of σ is free and thus a source of nondeterminism. However, in many cases the choice is not free at all, but is instead determined fully by the simple type τ that we want to match. However, the choice is not always forced by the matching. For instance, the context type $\forall\alpha.(\alpha \to \mathit{String}) \Rightarrow (\mathit{String} \to \alpha) \Rightarrow (\mathit{String} \to \mathit{String})$ encodes the Haskell type `∀a. (Show a, Read a) => String -> String` of the ambiguous expression **`read . show`** discussed in Section 2.2.1. The choice of α is ambiguous when matching against the simple type $\mathit{String} \to \mathit{String}$. Yet, the choice is critical for two reasons. Firstly, if we guess the wrong instantiation σ for α, then it may not be possible to recursively resolve $(\mathit{String} \to \alpha)[\sigma/\alpha]$ or $(\alpha \to \mathit{String})[\sigma/\alpha]$, while with a lucky guess both can be resolved. Secondly, for different choices of σ, $(\mathit{String} \to \alpha)[\sigma/\alpha]$ and $(\alpha \to \mathit{String})[\sigma/\alpha]$ can be resolved in completely different ways.

In order to avoid any problems, we conservatively forbid all ambiguous context types in the implicit environment with the $\vdash_{\mathrm{unamb}} \rho_1$ side-condition in rule Ty-IAbs of Figure 4.2.[2] This judgment is defined in Figure 4.6 in terms of the auxiliary judgment $\bar{\alpha} \vdash_{\mathrm{unamb}} \rho$ which takes an additional sequence of type variables α that is initially empty.

This auxiliary judgment expresses that all type variables $\bar{\alpha}$ are resolved when matching against ρ. Its base case, rule UA-Simp, ensures that the accumulated type variables $\bar{\alpha}$ appear free in τ. The inductive rule UA-TAbs accumulates the

[2] A more permissive design would allow quantified type variables that are not mentioned anywhere, such as α in $\forall\alpha.\mathit{Int} \Rightarrow \mathit{Int}$, and instantiate them to a dummy type, like GHC's `GHC.Prim.Any`, which is only used for this purpose. As such unused type variables serve little purpose, we have opted not to make an exception for them.

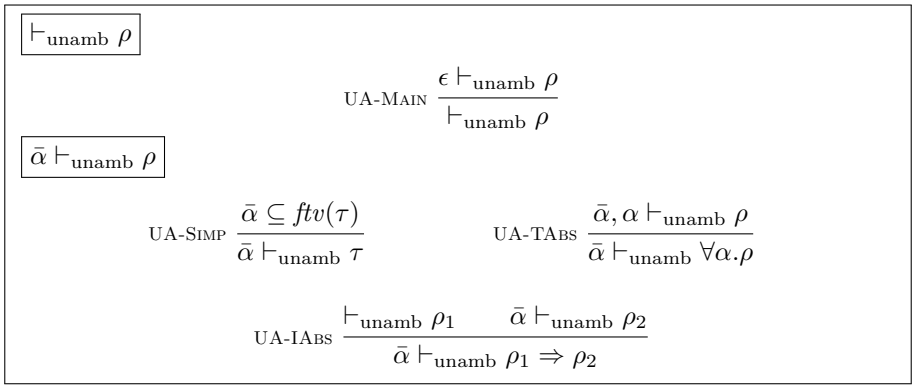


Figure 4.6: Unambiguous Context Types

bound type variables $\bar{\alpha}$ before the head. Rule UA-IAbs skips over any contexts on the way to the head, but also recursively requires that these contexts are unambiguous. The latter is necessary because rule FR-Simp resolves those contexts recursively when ρ matches the resolvent; as recursive resolvents they add their contexts to the implicit environment in rule FR-IAbs.

Finally, the unambiguity condition is imposed on the queried type ρ in rule Ty-Query because this type too may extend the implicit environment in rule FR-IAbs.

Note that the definition rules out harmless ambiguity, such as that in the type $\forall\alpha.Int$. When we match the head of this type Int with the simple type Int, the matching succeeds without actually determining how the type variable α should be instantiated. Here the ambiguity is harmless, because it does not affect the semantics. Yet, to keep the meta-theory simple, we have opted to not differentiate between harmless and harmful ambiguity.

Committed Choice

The other remaining source of nondeterminism is the nondeterministic choice $?\rho \in \Gamma$ that appears in rule FR-Simp of the focusing judgment. Consider the trivial example of resolving the goal Int against the environment $\Gamma = ?Int \rightsquigarrow x, ?Int \rightsquigarrow y$. Both implicits in the environment match the goal and yield different, i.e., incoherent, elaborations.

Our solution is to replace the nondeterministic relation $?\rho \in \Gamma$ by a deterministic one that selects the first matching implicit in the environment and commits to

it. In fact, we encapsulate all three hypotheses of rule FR-Simp in a new *lookup* judgment $\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$ which resolves τ with the first matching implicit in the environment Γ' and performs any recursive resolutions against the environment Γ. Of course, the modified rule FR-Simp' invokes this lookup judgment with two copies of the same environment, i.e., Γ and Γ' are identical.

$$\textsc{fr-Simp'}\ \frac{\Gamma; [\Gamma] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\tau] \rightsquigarrow E}$$

The (still preliminary) definition of the judgment itself is syntax-directed with respect to the type environment Γ':

$\boxed{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$ Lookup (Preliminary)

$$\textsc{dl-Match}\ \frac{\Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E \qquad \Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho'] \rightsquigarrow E' \quad (\forall \rho' \in \overline{\rho'})}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E[\bar{E}'/\bar{x}]}$$

$$\textsc{dl-NoMatch}\ \frac{\nexists E, \Sigma: \ \Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E \qquad \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

$$\textsc{dl-Var}\ \frac{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma; [\Gamma', x : \rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \textsc{dl-TyVar}\ \frac{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma; [\Gamma', \alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$$

Rule DL-Match concerns the case where the first entry in the environment matches the goal. Its behavior is the same as in the original definition of rule FR-Simp.

Rule DL-NoMatch is mutually exclusive with the above rule: it skips the first entry in the environment iff it does not match, to look for a matching implicit deeper in the environment. This implements the committed choice semantics: the first matching implicit is committed to and further implicits are not considered.

Finally, rules DL-Var and DL-TyVar skip the irrelevant non-implicit entries in the type environment.

It is not difficult to see that with the above definition there is only one way to resolve the goal *Int* against the environment $\Gamma = ?Int \rightsquigarrow x, ?Int \rightsquigarrow y$. The first matching entry, which elaborates to y, is committed to and the second entry is not considered.

Stability

While the above committed-choice formulation of resolution is deterministic, it is a rather fragile, or *unstable*, notion of resolution. Consider for example resolving the goal Int against the environment $\Gamma = ?Int \rightsquigarrow x, \alpha, ?\alpha \rightsquigarrow y$. This scenario arises for instance when type checking the expression $e = \lambda_? Int.(\Lambda\alpha.\lambda_?\alpha.?Int)\ Int$. Our definition of resolution skips the first implicit entry in the environment, because α does not match Int, commits to the second implicit entry, because Int trivially matches Int, and elaborates to x.

However, this resolution is not stable. Consider what happens when we apply a seemingly innocuous refactoring to the expression e by β-reducing the type application. This yields the new, and supposedly equivalent, expression $e' = \lambda_? Int.\lambda_? Int.?Int$. The direct impact of this refactoring on the resolution problem is to substitute Int for the type variable α. As a consequence the resolution commits now to the first entry and elaborates to y instead of x. Hence, more generally, the above definition of resolution is not stable under type substitution. This defies the common expectation that simple refactorings, like the reduction of type application above, do not change a program's behavior.

To avoid this problem and obtain stability under type substitution, we tighten the requirement of rule DL-NoMatch: an implicit in the environment can only be skipped iff it does not match under any possible substitution of type variables. With this tightened requirement the scenario above simply does not resolve: unstable resolutions are invalid.

$$\text{DL-NoMatch'}\ \frac{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau) \qquad \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

where a first stab at a formalisation of the stability condition is:

$\boxed{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau)}$ Stability condition (Ppreliminary)

$$\text{Stable'}\ \frac{\nexists \theta, E, \Sigma:\ \ \theta(\Gamma); [\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma; \theta(\tau) \rightsquigarrow E}{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau)}$$

The above formulation of the condition is a bit too lax; we have to be more precise about the domain and range of the substitution θ. Indeed, substitution does not make sense for every type variable in the environment. Consider for example resolving the type $\forall\beta.\beta \to \beta$ against the environment $\Gamma_0 = ?(\forall\gamma.\gamma \to \gamma) \rightsquigarrow x, \alpha, ?(\alpha \to \alpha) \rightsquigarrow y$. We would like this resolution of $\forall\beta.\beta \to \beta$ to succeed against $?(\forall\gamma.\gamma \to \gamma)$.

Unfortunately, the above formulation of stability unnecessarily throws a spanner in the works. Consider what happens: Using Rule FR-TAbs, we would recursively resolve $\beta \to \beta$ against the extended environment $\Gamma_1 = \Gamma_0, \beta$. Next we get stuck as neither rule DL-Match nor rule DL-NoMatch' applies. The former does not apply because $\alpha \to \alpha$ does not match $\beta \to \beta$. Also the latter does not apply because there are two substitutions such that $\theta(\alpha \to \alpha)$ matches $\theta(\beta \to \beta)$ and hence skipping $\alpha \to \alpha$ is deemed unstable.

However, if we look more closely at these substitutions, we see that none of them make sense. Essentially, there are two groups of substitutions:

- Those substitutions that instantiate β, of which $\theta = [\alpha/\beta]$ is a prominent example. These substitutions do not make sense because code inlining cannot result in β being instantiated to α or to any other type, because β is not in scope at the point in the code where the query happens (i.e., β does not appear in Γ_0). Hence, considering substitutions of β does not make sense.

  Figure 4.8, which puts all the measures together to obtain a type-directed, deterministic and stable resolution, addresses the issue as follows. It introduces a top-level *main* judgment $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$ to handle a query that delegates to the focusing-based judgments we have described above. The only contribution of the main judgment, which is defined by the single rule R-Main, is to gather the type variables $\bar{\alpha}$ that appear in the environment at the point of the query by means of the function $\mathsf{tyvars}(\Gamma)$, and to pass them on through the auxiliary judgments to the point where the stability check is performed. Hence, the auxiliary judgments $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$ (focusing), $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$ (lookup) and $\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$ now all feature an additional argument $\bar{\alpha}$ of type variables that can be substituted.

- The substitution $\theta' = [\beta/\alpha]$ also generates a match. However, this substitution does not make sense either because code inlining can only result in substitutions of α by types that are well-scoped in the prefix of the environment before α. In the case of the example this means that we can only consider substitutions $[\sigma/\alpha]$ where $?(\forall\gamma.\gamma \to \gamma) \rightsquigarrow x \vdash \sigma$. In other words, σ cannot have any free type variables. There is no such σ that matches β.

In summary, Figure 4.7 formalises our notion of valid substitutions with the judgment $\bar{\alpha}; \Gamma \vdash \theta$. Rule S-Empty covers the base case and states that the empty substitution ϵ is trivially valid. Rule S-Cons covers the inductive case $[\sigma/\alpha] \cdot \theta$. It says that the single variable substitution $[\sigma/\alpha]$ is valid if α appears in the sequence of substitutable type variables, expressed by the structural pattern $\bar{\alpha}, \alpha, \bar{\alpha}'$. Moreover, α must appear in the type environment, expressed by a

$$\theta ::= \epsilon \mid [\sigma/\alpha] \cdot \theta$$

$\boxed{\bar{\alpha}; \Gamma \vdash \theta}$ Valid Substitutions

$$\textsc{S-Empty}\ \frac{}{\bar{\alpha}; \Gamma \vdash \epsilon} \qquad \textsc{S-Cons}\ \frac{\Gamma \vdash \sigma \qquad \bar{\alpha}, \bar{\alpha}'; \Gamma, \theta(\Gamma') \vdash \theta}{\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma' \vdash [\sigma/\alpha] \cdot \theta}$$

Figure 4.7: Valid Substitutions

similar structural pattern Γ, α, Γ'. Lastly, the type σ must be well-scoped with respect to the environment prefix Γ. In addition, the remainder θ must be valid with respect to the remaining type variables $\bar{\alpha}, \bar{\alpha}'$ and the type environment after substitution of α.

Summary

Figure 4.8 puts all the above measures together in our unambiguous, deterministic and stable definition of resolution.

The *main judgment* $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$ resolves the query ρ against the type environment Γ. It is defined by a single rule, R-Main, which uses the $\mathsf{tyvars}(\Gamma)$ function to retrieve the type variables $\bar{\alpha}$ that are recorded in the type environment, and delegates the task to the auxiliary *focusing judgment* $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$. The definition of $\mathsf{tyvars}(\cdot)$ is shown here:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{tyvars}(\epsilon) & = & \epsilon & \mathsf{tyvars}(\Gamma, \alpha) & = & \mathsf{tyvars}(\Gamma), \alpha \\ \mathsf{tyvars}(\Gamma, x : \rho) & = & \mathsf{tyvars}(\Gamma) & \mathsf{tyvars}(\Gamma, ?\rho \rightsquigarrow x) & = & \mathsf{tyvars}(\Gamma) \end{array}$$

The focusing judgment has one more index than the main judgment for tracking the type variables $\bar{\alpha}$ in the type environment when focusing has first been triggered. Three rules define this judgment. The first two, R-IAbs and R-TAbs, strip the query type ρ until only a simple type τ remains, which is handled by rule R-Simp. Rule R-IAbs strips the context ρ_1 from a rule type $\rho_1 \Rightarrow \rho_2$, adds it to the type environment as a new implicit and then recursively processes the head ρ_2. Rule R-TAbs strips the quantifier from a universally quantified type $\forall \alpha.\rho$ and adds the type variable α to the type environment in which ρ is processed. Finally, rule R-Simp delegates the job of processing the simple type τ to the auxiliary *lookup judgment* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$.

The lookup judgment takes an additional index, the type environment Γ', which is initialised to Γ in rule R-Simp. It pops entries from Γ' until it finds an implicit

$$\Sigma ::= \epsilon \mid \rho \rightsquigarrow x, \Sigma$$

$\boxed{\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E}$ Main

$$\textsc{R-Main}\ \frac{\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E}{\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E}$$

$\boxed{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E}$ Focusing

$$\textsc{R-IAbs}\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x \vdash_{\mathrm{r}} [\rho_2] \rightsquigarrow E \qquad x\ \mathit{fresh}}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$\textsc{R-TAbs}\ \frac{\bar{\alpha}; \Gamma, \alpha \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\forall \alpha.\rho] \rightsquigarrow \Lambda\alpha.E} \qquad \textsc{R-Simp}\ \frac{\bar{\alpha}; \Gamma; [\Gamma] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\tau] \rightsquigarrow E}$$

$\boxed{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$ Lookup

$$\textsc{L-Match}\ \frac{\Gamma; [\rho] \rightsquigarrow x \vdash_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E \qquad \bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho'] \rightsquigarrow E' \quad (\forall \rho' \in \overline{\rho'})}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E[\bar{E}'/\bar{x}]}$$

$$\textsc{L-NoMatch}\ \frac{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau) \qquad \bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

$$\textsc{L-Var}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', x : \rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \textsc{L-TyVar}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', \alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$$

$\boxed{\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$ Matching

$$\textsc{M-Simp}\ \frac{}{\Gamma; [\tau] \rightsquigarrow E \vdash_{\mathrm{r}} \epsilon; \tau \rightsquigarrow E}$$

$$\textsc{M-IApp}\ \frac{\Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\, x \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad x\ \mathit{fresh}}{\Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash_{\mathrm{r}} \rho_1 \rightsquigarrow x, \Sigma; \tau \rightsquigarrow E'}$$

$$\textsc{M-TApp}\ \frac{\Gamma; [\rho[\sigma/\alpha]] \rightsquigarrow E\, |\sigma| \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad \Gamma \vdash \sigma}{\Gamma; [\forall \alpha.\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$$

$\boxed{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)}$ Stability

$$\textsc{Stable}\ \frac{\nexists \theta, E, \Sigma : \ \bar{\alpha}; \Gamma \vdash \theta \qquad \theta(\Gamma); [\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma; \theta(\tau) \rightsquigarrow E}{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)}$$

Figure 4.8: Deterministic Resolution and Translation to System F

that matches the simple query type τ. Rule L-Match first uses the auxiliary *matching judgment* $\Gamma; [\rho] \rightsquigarrow x \vdash_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E$ to establish that implicit ρ at the top of Γ' matches the query type τ and to receive new queries ρ', which it resolves recursively against the type environment Γ. Rule L-NoMatch skips the implicit at the top of the environment when it is stable to do so according to the auxiliary *stability judgment* $\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$. Rules L-Var and L-TyVar skip term and type variable entries.

Three rules define the matching judgment. In the first one, M-Simp, the implicit is a simple type τ that is identical to query type; there are no remaining queries. When the implicit is a rule type $\rho_1 \Rightarrow \rho_2$, rule M-IApp defers querying ρ_1 and first checks whether ρ_2 matches the query τ. When the implicit is a universally quantified type $\forall\alpha.\rho$, rule M-TApp instantiates it appropriately to match the query τ.

Finally, the stability judgment is defined by a single rule, Stable, which makes sure that there is no substitution θ of the type variables $\bar{\alpha}$ for which $\theta(\rho)$ matches $\theta(\tau)$.

4.3 Resolution Algorithm

This section presents an algorithm that implements the deterministic resolution rules of Figure 4.8. It differs from the latter in two important ways: firstly, it computes rather than guesses type substitutions in rule M-TApp; and secondly, it replaces explicit quantification over all substitutions θ in rule Stable with a tractable approach to stability checking.

The definition of the algorithm, in Figure 4.9, is structured in the same way as the declarative specification: with one main judgment and three auxiliary ones that have similar roles (focusing, lookup, and matching). In fact, the main and focusing judgments are identical, since the differences are not situated there.

4.3.1 Deferred Variable Instantiation

The first difference is in the matching judgment $\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'$. While its declarative counterpart immediately instantiates the quantified type variable in rule M-TApp, this algorithmic formulation defers the instantiation to the point where a deterministic choice can be made. As long as the type variables $\bar{\alpha}$ have not been instantiated, the judgment keeps track of them in its first argument. The actual instantiation happens in the base case, rule Alg-M-Simp. This last rule performs the deferred instantiation of type variables

$\boxed{\Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E}$ Main

$$\textsc{Alg-R-Main}\ \frac{\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E}{\Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E}$$

$\boxed{\bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E}$ Focusing

$$\textsc{Alg-R-IAbs}\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x \vdash_{\mathrm{g}} \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda(x : |\rho_1|).E}$$

$$\textsc{Alg-R-TAbs}\ \frac{\bar{\alpha}; \Gamma, \alpha \vdash_{\mathrm{g}} \rho \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E} \qquad \textsc{Alg-R-Simp}\ \frac{\bar{\alpha}; \Gamma; [\Gamma] \vdash_{\mathrm{g}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \tau \rightsquigarrow E}$$

$\boxed{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{g}} \tau \rightsquigarrow E}$ Lookup

$$\textsc{Alg-L-Match}\ \frac{\begin{array}{c}\epsilon; \Gamma; [\rho] \rightsquigarrow x; \epsilon \vdash_{\mathrm{g}} \bar{\rho}' \rightsquigarrow \bar{x}'; \tau \rightsquigarrow E \\ \bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \rho' \rightsquigarrow E' \ \ (\forall \rho' \in \bar{\rho}')\end{array}}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{g}} \tau \rightsquigarrow E[\bar{E}'/\bar{x}']}$$

$$\textsc{Alg-L-NoMatch}\ \frac{\bar{\alpha}; \Gamma; \rho \not\vdash_{\mathrm{sta}} \tau \qquad \bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{g}} \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{g}} \tau \rightsquigarrow E'}$$

$$\textsc{Alg-L-Var}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{g}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', x : \rho] \vdash_{\mathrm{g}} \tau \rightsquigarrow E} \qquad \textsc{Alg-L-TyVar}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{g}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', \alpha] \vdash_{\mathrm{g}} \tau \rightsquigarrow E}$$

$\boxed{\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'}$ Matching

$$\textsc{Alg-M-Simp}\ \frac{\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')}{\bar{\alpha}; \Gamma; [\tau'] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \theta(\Sigma); \tau \rightsquigarrow |\theta|(E)}$$

$$\textsc{Alg-M-IApp}\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\, x; \Sigma, \rho_1 \rightsquigarrow x \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E' \qquad x\ \mathit{fresh}}{\bar{\alpha}; \Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'}$$

$$\textsc{Alg-M-TApp}\ \frac{\bar{\alpha}, \alpha; \Gamma, \alpha; [\rho] \rightsquigarrow E\, \alpha; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\forall\alpha.\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'}$$

Figure 4.9: Resolution Algorithm

$\bar{\alpha}$ by computing the *most general unifier* $\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau', \tau)$. The unification algorithm, which we present below, computes a substitution θ that is valid (i.e., $\bar{\alpha}; \Gamma \vdash \theta$) and that equates the two types (i.e., $\theta(\tau) = \theta(\tau')$).

In order to subject the recursive goals to this substitution, the algorithmic judgment makes use of an accumulating parameter Σ. This accumulator Σ represents all the goals collected so far in which type variables have not been substituted yet. In contrast, Σ' denotes all obligations with type variables already substituted.

Finally, observe that rule Alg-L-Match invokes the algorithmic judgment with an empty set of not-yet-instantiated type variables and an empty accumulator Σ.

The following example illustrates the differences between the declarative judgment:

$$\dfrac{\dfrac{\dfrac{}{\Gamma; [Int] \rightsquigarrow x\ Int\ y \vdash_{\mathrm{r}} \epsilon; Int \rightsquigarrow x\ Int\ y}\ \text{M-Simp}}{\Gamma; [Int \Rightarrow Int] \rightsquigarrow x\ Int \vdash_{\mathrm{r}} Int \rightsquigarrow y; Int \rightsquigarrow x\ Int\ y}\ \text{M-IApp}}{\Gamma; [\forall \alpha.\alpha \Rightarrow \alpha] \rightsquigarrow x \vdash_{\mathrm{r}} Int \rightsquigarrow y; Int \rightsquigarrow x\ Int\ y}\ \text{M-TApp}$$

and its algorithmic counterpart:

$$\dfrac{\dfrac{\dfrac{[Int/\alpha] = \mathsf{unify}_{\Gamma;\alpha}(\alpha, Int)}{\alpha; \Gamma; [\alpha] \rightsquigarrow x\ \alpha\ y; \alpha \rightsquigarrow y \vdash_{\mathrm{g}} \epsilon; Int \rightsquigarrow x\ Int\ y}\ \text{Alg-M-Simp}}{\alpha; \Gamma; [\alpha \Rightarrow \alpha] \rightsquigarrow x\ \alpha; \epsilon \vdash_{\mathrm{g}} Int \rightsquigarrow y; Int \rightsquigarrow x\ Int\ y}\ \text{Alg-M-IApp}}{\epsilon; \Gamma; [\forall \alpha.\alpha \Rightarrow \alpha] \rightsquigarrow x; \epsilon \vdash_{\mathrm{g}} Int \rightsquigarrow y; Int \rightsquigarrow x\ Int\ y}\ \text{Alg-M-TApp}$$

4.3.2 Algorithmic Stability Check

The second difference can be found in rule Alg-L-NoMatch of the lookup judgment. Instead of using the $\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$ judgment, which quantifies over all valid substitutions, this rule uses the algorithmic judgment $\bar{\alpha}; \Gamma; \rho \vdash_{\mathrm{sta}} \tau$, defined in Figure 4.10. This auxiliary judgment checks algorithmically whether type ρ matches τ under any possible instantiation of the type variables $\bar{\alpha}$. We use the same deferred-instantiation technique as with the first difference: instead of applying a substitution first and then checking whether the implicit matches the goal, we defer the instantiation to the end where we can deterministically pick one instantiation instead of considering all valid instantiations.

As a consequence of the similarity, in purpose and strategy, between the matching judgment, $\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'$, and the algorithmic

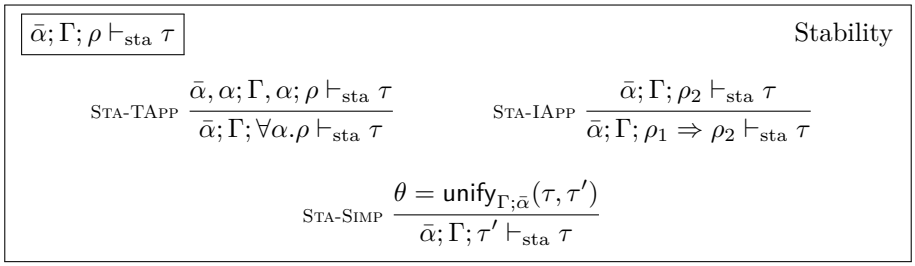


Figure 4.10: Algorithmic Stability Check

stability, $\bar{\alpha}; \Gamma; \rho \vdash_{\text{sta}} \tau$, the latter is a variation of the former, with two differences. Firstly, since the stability judgment is only concerned with matchability, no recursive resolvents Σ are collected nor are any elaborations tracked. Secondly, since the stability check considers the substitution of the type variables $\bar{\alpha}$ that occur in the environment at the point of the query, rule Alg-L-NoMatch pre-populates the substitutable variables of the $\vdash_{\text{sta}}$ judgment with them. Contrast this with the matching judgment where only the implicit's quantified variables are instantiated, as witnessed by rules Alg-M-TApp and Alg-M-Simp.

4.3.3 Scope-Aware Unification

The unification algorithm $\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)$ is a key component of the two algorithmic changes explained above.

Figure 4.11 provides its definition, which is a hybrid between standard first-order unification [Martelli & Montanari 1982] and polymorphic type instantiation [Dunfield & Krishnaswami 2013] that is a restricted form of Miller's [1992] mixed-prefix unification. The domain restriction $\bar{\alpha}$ denotes which type variables are to be treated as unification variables; all other type variables are to be treated as constants. The returned substitution is a unifier of ρ_1 and ρ_2, i.e., $\theta(\rho_1) = \theta(\rho_2)$.

Validity The differences with standard first-order unification arise because, like polymorphic type instantiation, the algorithm has to account for the scope of type variables. Indeed, as we have already explained in Section 4.2.5, we expect that the returned substitution is valid, i.e., $\bar{\alpha}; \Gamma \vdash \theta$. For instance, using standard first-order unification for $\mathsf{unify}_{\Gamma;\beta}(\forall\alpha.\alpha \to \beta, \forall\alpha.\alpha \to \alpha)$ would yield the substitution $[\beta/\alpha]$, which is invalid because α is not in scope in Γ.

Most General Unifier Secondly, traditional unification computes the most general unifier, i.e., any other unifier can be expressed as its composition with another substitution. Yet, the most general unifier may not be a valid substitution, while more specific unifiers may be valid. Consider for instance unifying α with $\beta \to \beta$ where $\Gamma = \alpha, \beta$ and both α and β are unification variables. The most general unifier is $[\beta \to \beta/\alpha]$. However this unifier is not valid, as α appears before β in the environment. In contrast, there are infinitely many more specific unifiers that are valid, all of the form $[\rho \to \rho/\alpha, \rho/\beta]$ where ρ is a closed type.

Fortunately, by a stroke of luck, the above is not a problem for either of our two use cases:

- The first use case is that in rule Alg-M-Simp where this is not a problem because the scenario never arises. In $\mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau', \tau)$ only τ' contains unification variables and hence the range of the substitution never contains any unification variables. As a consequence the above example and others like it cannot occur.
- The second use case, in rule Sta-Simp, is only interested in the existence of a valid substitution. We neither care which one it is nor whether it is the most general one. Moreover, as illustrated above, whenever there is a most general substitution that is invalid due to the relative position of unification variables in the environment, we can always construct a more specific valid substitution by substituting the remaining unification variables by closed types.

Definition With the above issues in mind we can consider the actual definition in Figure 4.11. The main unification judgment $\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)$ is defined by rule U-Main. This rule computes the unifier in terms of the auxiliary judgment $\theta = \mathsf{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)$, which is essentially standard unification, and then checks the above validity concerns. Indeed, for any type variable β that appears in the image of a type variable α, either β must appear before α in the environment Γ (regular validity), or β must itself be a unification variable (the exceptional case). The relative position of variables is checked with the auxiliary judgment $\beta >_\Gamma \alpha$ whose one rule verifies that β appears before α in the environment Γ;[3] a similar check on relative positions can be found in Dunfield & Krishnaswami's [2013] algorithm.

The auxiliary judgment $\mathsf{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)$ computes the actual unifier. Rule U-Var is the standard reflexivity rule for type variables. Rules U-InstL and U-InstR

[3]If type variables are represented by de Bruijn indices, this can be done by checking whether one index is greater than the other.

$\boxed{\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)}$

$$\textsc{U-Main}\ \frac{\theta = \mathsf{unify}'_{\bar{\alpha}}(\rho_1, \rho_2) \qquad \beta \in \bar{\alpha} \ \vee\ \beta >_\Gamma \alpha \quad (\forall [\sigma/\alpha] \in \theta, \forall \beta \in \mathit{ftv}(\sigma))}{\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)}$$

$\boxed{\theta = \mathsf{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)}$

$$\textsc{U-Var}\ \frac{}{\epsilon = \mathsf{unify}'_{\bar{\alpha}}(\beta, \beta)} \qquad \textsc{U-Univ}\ \frac{\theta = \mathsf{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)}{\theta = \mathsf{unify}'_{\bar{\alpha}}(\forall \beta.\rho_1, \forall \beta.\rho_2)}$$

$$\textsc{U-InstL}\ \frac{\alpha \in \bar{\alpha} \qquad \alpha \notin \mathit{ftv}(\sigma)}{[\sigma/\alpha] = \mathsf{unify}'_{\bar{\alpha}}(\alpha, \sigma)} \qquad \textsc{U-InstR}\ \frac{\alpha \in \bar{\alpha} \qquad \alpha \notin \mathit{ftv}(\sigma)}{[\sigma/\alpha] = \mathsf{unify}'_{\bar{\alpha}}(\sigma, \alpha)}$$

$$\textsc{U-Fun}\ \frac{\theta_1 = \mathsf{unify}'_{\bar{\alpha}}(\rho_{11}, \rho_{21}) \qquad \theta_2 = \mathsf{unify}'_{\bar{\alpha}}(\theta_1(\rho_{12}), \theta_1(\rho_{22}))}{\theta_2 \cdot \theta_1 = \mathsf{unify}'_{\bar{\alpha}}(\rho_{11} \to \rho_{12}, \rho_{21} \to \rho_{22})}$$

$$\textsc{U-Rul}\ \frac{\theta_1 = \mathsf{unify}'_{\bar{\alpha}}(\rho_{11}, \rho_{21}) \qquad \theta_2 = \mathsf{unify}'_{\bar{\alpha}}(\theta_1(\rho_{12}), \theta_1(\rho_{22}))}{\theta_2 \cdot \theta_1 = \mathsf{unify}'_{\bar{\alpha}}(\rho_{11} \Rightarrow \rho_{12}, \rho_{21} \Rightarrow \rho_{22})}$$

$\boxed{\beta >_\Gamma \alpha}$

$$\frac{}{\beta >_{\Gamma_1,\beta,\Gamma_2,\alpha,\Gamma_3} \alpha}$$

Figure 4.11: Unification Algorithm

are two symmetric base cases; they only create a substitution $[\sigma/\alpha]$ if α is one of the unification variables and if α does not occur in σ, which is the well-known occurs-check. Rules U-Fun, U-Rul and U-Univ are standard congruence rules that combine the unification problems derived for their subterms.

4.3.4 Termination of Resolution

If we are not careful about which implicits are added to the environment, then the resolution process may not terminate. This section describes how to impose a set of modular syntactic restrictions that prevents non-termination. As an example of non-termination consider

$$?(\mathit{Char} \Rightarrow \mathit{Int}), ?(\mathit{Int} \Rightarrow \mathit{Char}) \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}$$

which loops, using alternatively the first and second implicit in the environment. The source of this non-termination is the recursive nature of resolution: a simple

type can be resolved in terms of an implicit type whose head it matches, but this requires further resolution of the implicit type's context.

The problem of non-termination has been widely studied in the context of Haskell's type classes, and a set of modular syntactic restrictions has been imposed on type class instances to avoid non-termination [Sulzmann et al. 2007]. We adapt those restrictions to the setting of COCHIS.

Overall Approach We show termination by characterising the resolution process as a (resolution) tree with goals in the nodes. The initial goal sits at the root of the tree. A multi-edge from a parent node to its children represents a rule type from the environment that matches the parent node's goal; the node's children are the recursive goals.

Resolution terminates if the tree is finite. Hence, if it does not terminate, there is an infinite path from the root in the tree, that denotes an infinite sequence of matching rule type applications. To show that there cannot be such an infinite path, we use a norm $\|\cdot\|$ (defined at the bottom of Figure 4.12) that maps the head of every goal ρ to a natural number, its size. While types like *Int* and *Bool* are not formally part of our calculus, we assume that their size is 1. Similarly, the size of type constructor applications like $List\ \rho$ is $1 + \|\rho\|$.

By showing that this size strictly decreases from any parent goal to its children, we can conclude that, because the order on the natural numbers is well-founded, on any path from the root there is eventually a goal that has no children.

Termination Condition It is trivial to show that the size strictly decreases, if we require that every implicit binding in the environment makes it so. This requirement is formalised as the termination condition $\vdash_{\text{term}} \rho$ in Figure 4.12. This condition should be imposed on every type added to the environment, namely to ρ_1 in rule TY-IABS of Figure 4.2 and to ρ_1 in rule R-IABS of Figure 4.8. However, because the latter concerns only a part of a resolved type, we feel that it is easier to follow for the programmer if we impose the condition instead on the whole resolved type in rule TY-QUERY of Figure 4.2.

The judgment is defined by case analysis on the type ρ. Rule T-SIMP states that simple types trivially satisfy the condition. Next, rule T-FORALL is the congruence rule for universally quantified types. Finally, rule T-RULE enforces the actual condition on rule types $\rho_1 \Rightarrow \rho_2$, which requires that the head τ_1 of ρ_1 is strictly smaller than the head τ_2 of ρ_2.

To account for polymorphism and the fact that the type variables in rule types can be instantiated, rule T-RULE ensures that the $\|\tau_1\| < \|\tau_2\|$ property is

$\boxed{\vdash_{\text{term}} \rho}$

$$\textsc{T-Simp}\ \frac{}{\vdash_{\text{term}} \tau} \qquad \textsc{T-Forall}\ \frac{\vdash_{\text{term}} \rho}{\vdash_{\text{term}} \forall\alpha.\rho}$$

$$\textsc{T-Rule}\ \frac{\begin{array}{c}\vdash_{\text{term}} \rho_1 \qquad \vdash_{\text{term}} \rho_2 \qquad \tau_1 = \text{hd}(\rho_1) \qquad \tau_2 = \text{hd}(\rho_2) \\ \|\tau_1\| < \|\tau_2\| \qquad \forall\alpha \in ftv(\rho_1) \cup ftv(\rho_2) :\ \text{occ}_\alpha(\tau_1) \leqslant \text{occ}_\alpha(\tau_2)\end{array}}{\vdash_{\text{term}} \rho_1 \Rightarrow \rho_2}$$

$$\begin{array}{c}\text{hd}(\tau) = \tau \\ \text{hd}(\forall\alpha.\rho) = \text{hd}(\rho) \\ \text{hd}(\rho_1 \Rightarrow \rho_2) = \text{hd}(\rho_2)\end{array} \qquad \left| \qquad \begin{array}{c}\text{occ}_\alpha(\beta) = \begin{cases}1 & \text{if } (\alpha = \beta) \\ 0 & \text{if } (\alpha \neq \beta)\end{cases} \\ \text{occ}_\alpha(\forall\beta.\rho) = \text{occ}_\alpha(\rho) \quad (\alpha \neq \beta) \\ \text{occ}_\alpha(\rho_1 \to \rho_2) = \text{occ}_\alpha(\rho_1) + \text{occ}_\alpha(\rho_2) \\ \text{occ}_\alpha(\rho_1 \Rightarrow \rho_2) = \text{occ}_\alpha(\rho_1) + \text{occ}_\alpha(\rho_2)\end{array} \right.$$

$$\begin{array}{c}\|\alpha\| = 1 \\ \|\rho_1 \to \rho_2\| = 1 + \|\rho_1\| + \|\rho_2\|\end{array} \qquad \begin{array}{c}\|\forall\alpha.\rho\| = \|\rho\| \\ \|\rho_1 \Rightarrow \rho_2\| = 1 + \|\rho_1\| + \|\rho_2\|\end{array}$$

Figure 4.12: Termination Condition

stable under substitution. Declaratively, we can formulate this stability under substitution as:

$$\forall\theta.dom(\theta) \subseteq ftv(\tau_1) \cup (ftv)(\tau_2) :\ \ \|\theta(\tau_1)\| < \|\theta(\tau_2)\|$$

Consider for instance the type $\forall a.(a \to a) \Rightarrow (a \to \mathit{Int} \to \mathit{Int})$. In this example, the head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate α to $(\mathit{Int} \to \mathit{Int} \to \mathit{Int})$, then the head's size becomes 9 while the context constraint's size becomes 11.

The declarative formulation above is not suitable in an algorithm because it enumerates all possible substitutions. Rule T-Rule uses instead an equivalent algorithmic formulation which states that, in addition to $\|\tau_1\| < \|\tau_2\|$, the number of occurrences of any free type variable α may not be larger in τ_1 than in τ_2. The first condition expresses that for the empty substitution, the size strictly decreases, say from $\|\tau_2\| = n$ to $\|\tau_1\| = m$. If we instantiate a type variable α to a type σ of size k, then the sizes change to $\|[\sigma/\alpha]\tau_1\| = m + k \times \text{occ}_\alpha(\tau_1)$ and $\|[\sigma/\alpha]\tau_2\| = n + k \times \text{occ}_\alpha(\tau_2)$ where the auxiliary function $\text{occ}_\alpha(\rho)$ determines the number of occurrences of α in ρ. The second condition guarantees that

$k \times \mathrm{occ}_\alpha(\tau_1) \leqslant k \times \mathrm{occ}_\alpha(\tau_2)$ and thus that the strict decrease is preserved under substitution. In our example above, we do have that $\|\alpha \to \alpha\| = 3 < 5 = \|\alpha \to \mathit{Int} \to \mathit{Int}\|$, the second condition is not satisfied, i.e., $\mathrm{occ}_\alpha(\alpha \to \alpha) = 2 \not\leqslant 1 = \mathrm{occ}_\alpha(\alpha \to \mathit{Int} \to \mathit{Int})$.

Finally, as the types have a recursive structure whereby their components are themselves added to the environment, rule T-Rule also enforces the termination condition recursively on the components.

Discussion Above we have adapted termination conditions for Haskell's type class resolution to Cochis. While our adapted conditions are sufficient for termination, they are not necessary. In fact, they can be rather restrictive. For instance, $\not\vdash_{\mathrm{term}} \mathit{Int} \Rightarrow \mathit{Bool}$ because $\|\mathit{Int}\| \not< \|\mathit{Bool}\|$. Indeed, resolving *Bool* in the context $\Gamma_1 = ?(\mathit{Bool} \Rightarrow \mathit{Int}), ?(\mathit{Int} \Rightarrow \mathit{Bool})$ is problematic. Yet, it is not in the context $\Gamma_2 = ?(\mathit{Int}), ?(\mathit{Int} \Rightarrow \mathit{Bool})$. The problem is that the conditions are not context sensitive. We leave exploring more permissive, context-sensitive conditions to future work.

4.4 Type-Directed Translation to System F

In this section we explain the dynamic semantics of Cochis in terms of System F's dynamic semantics, by means of a type-directed translation. This translation turns implicit contexts into explicit parameters and statically resolves all queries, much like Wadler & Blott's [1989] dictionary passing translation for type classes. The advantage of this approach is that we simultaneously provide a meaning to well-typed Cochis programs and an effective implementation that resolves all queries statically.

The translation follows the type system presented in Section 4.2. The additional machinery that is necessary (on top of the type system) corresponds to the grayed parts of Figures 4.2, 4.3 and 4.8.

4.4.1 Type-Directed Translation

Figure 4.2 presents the translation rules that convert Cochis expressions into System F expressions. The syntax and typing rules of System F are presented in Appendix A.1 of this thesis.

The gray parts of the figure extend the type system with the necessary information for the translation. In addition, the gray extension to the syntax of

type environments, Γ, annotates every implicit's type with explicit System F evidence in the form of a term variable x.

Translation of Types The function $|\cdot|$ takes COCHIS types ρ to System F types T:

$$|\alpha| = \alpha \qquad |\forall\alpha.\rho| = \forall\alpha.|\rho|$$
$$|\rho_1 \to \rho_2| = |\rho_1| \to |\rho_2| \qquad |\rho_1 \Rightarrow \rho_2| = |\rho_1| \to |\rho_2|$$

It reveals that implicit COCHIS arrows are translated to explicit System F function arrows.

Translation of Terms The type-directed translation judgment, which extends the typing judgment, is

$$\Gamma \vdash e : \rho \rightsquigarrow E$$

This judgment states that the translation of COCHIS expression e with type ρ is System F expression E, with respect to type environment Γ.

Variables, lambda abstractions and applications are translated straightforwardly. Perhaps the only noteworthy rule is TY-IABS. This rule associates the type ρ_1 with the fresh variable x in the type environment. This creates the necessary evidence that can be used by resolutions in the body of the rule abstraction to construct System F terms of type $|\rho_1|$.

Resolution The more interesting part of the translation happens when resolving queries. Queries are translated by rule TY-QUERY using the auxiliary resolution judgment $\vdash_{\mathrm{r}}$:

$$\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$$

which is shown, in deterministic form, in Figure 4.8. The translation builds a System F term as evidence for the resolution.

The evidence building mechanism dualizes the process of peeling off abstractions and universal quantifiers: Rule R-IABS wraps a lambda binder with a fresh variable x around a System F expression, E, which is generated from the resolution for the head of the rule (ρ_2). Similarly, rule R-TABS wraps a type binder around the System F expression resulting from the resolution of ρ.

For simple types τ rule R-SIMP delegates the work of building evidence, when a matching implicit ρ is found in the environment, to rule L-MATCH. The evidence consists of two parts: E is the evidence of matching τ against ρ. This match contains placeholders $\bar{x}$ for the contexts whose resolution is postponed by rule

M-IApp. It falls to rule L-Match to perform these postponed resolutions, obtain their evidence $\bar{E}$ and fill in the placeholders.

Meta-Theory The type-directed translation of Cochis to System F exhibits a number of desirable properties.

Theorem 4.1 (Type-Preserving Translation)**.**
If $\Gamma \vdash e : \rho \rightsquigarrow E$*, then* $|\Gamma| \vdash_{\text{F}} E : |\rho|$*, where* e *is a* Cochis *expression,* ρ *a type,* Γ *a type environment and* E *a System F expression.*

We define the translation of the type environment from Cochis to System F as:

$$|\epsilon| = \epsilon \qquad |\Gamma, \alpha| = |\Gamma|, \alpha$$
$$|\Gamma, x : \rho| = |\Gamma|, x : |\rho| \qquad |\Gamma, ?\rho \rightsquigarrow x| = |\Gamma|, x : |\rho|$$

An important lemma in the theorem's proof is the type preservation of resolution.

Lemma 4.1 (Type-Preserving Resolution)**.**
If $\Gamma \vdash^{\text{a}}_{\text{r}} \rho \rightsquigarrow E$*, then* $|\Gamma| \vdash_{\text{F}} E : |\rho|$*, where* Γ *is a type environment,* ρ *a type and* E *a System F expression.*

The full-proof Appendix points to the mechanized proofs of Theorem 4.1 and Lemma 4.1 stated above.

In addition, we can express three key properties of Figure 4.8's definition of resolution in terms of the generated evidence. Firstly, the deterministic version of resolution is a sound variation on the original ambiguous resolution.

Theorem 4.2 (Soundness)**.**
Figure 4.8's deterministic definition of resolution is sound (but incomplete) with respect to Figure 4.3's ambiguous definition.

$$\forall \Gamma, \rho, E : \qquad \Gamma \vdash_{\text{r}} \rho \rightsquigarrow E \quad \Rightarrow \quad \Gamma \vdash^{\text{a}}_{\text{r}} \rho \rightsquigarrow E$$

Section B.3 of the Appendix delineates the proof of the above theorem. As a corollary (by use of this result with Lemma 4.1), the deterministic resolution process is also type-preserving. Secondly, the deterministic resolution is effectively deterministic:

Theorem 4.3 (Determinacy)**.**
Let Γ *be a type environment,* ρ *be a type and* E_1*,* E_2 *be system F expressions. Assume that* $\vdash_{\text{unamb}} \rho$ *and* $\forall ?\rho_i \in \Gamma, \vdash_{\text{unamb}} \rho_i$*. Then*

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad E_1 = E_2$$

A proof sketch of Theorem 4.3 can be found in Section B.4 of the Appendix.

It follows immediately that deterministic resolution is also coherent, which for our elaboration-based setting is formulated, following Biernacki & Polesiuk [2015], in terms of *contextual equivalence* [Morris Jr 1969].

Corollary 4.1 (Coherence)**.**
Let Γ *be a type environment,* ρ *be a type and* E_1*,* E_2 *be system F expressions. Assume that* $\vdash_{\text{unamb}} \rho$ *and* $\forall ?\rho_i \in \Gamma$*,* $\vdash_{\text{unamb}} \rho_i$*. Then*

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad |\Gamma| \vdash_{\text{F}} E_1 \simeq_{\text{ctx}} E_2 : |\rho|$$

Here, the well-studied [Chong 2017] contextual equivalence judgment

$$\Gamma \vdash_{\text{F}} E_1 \simeq_{\text{ctx}} E_2 : T$$

captures that E_1 and E_2 behave the same in any well-typed program context. We do not give the formal definition here as it is not required to prove the corollary; we only need to know that it is an equivalence relation, and, more specifically, that it is reflexive, i.e., any well-typed System F term is contextually equivalent to itself.

Thirdly, on top of the immediate coherence of deterministic resolution, an additional stability property holds.

Lemma 4.2 (Stability of Resolution)**.**
Resolution is stable under monotype substitution.

$$\begin{aligned}&\forall \Gamma, \alpha, \Gamma', \sigma, \rho, E : \\ &\quad \Gamma, \alpha, \Gamma' \vdash_{\text{r}} \rho \rightsquigarrow E \ \wedge \ \Gamma \vdash \sigma \ \Rightarrow \ \Gamma, \Gamma'[\sigma/\alpha] \vdash_{\text{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]\end{aligned}$$

This is a key lemma to establish that static reduction of type application preserves typing and elaboration.

Lemma 4.3 (Stability of Typing under Type Application Reduction)**.**
Static reduction of type application preserves typing.

$$\forall \Gamma, \alpha, \sigma, \rho, E : \quad \Gamma \vdash (\Lambda\alpha.e)\,\sigma : \rho \rightsquigarrow (\Lambda\alpha.E)\,|\sigma| \ \Rightarrow \ \Gamma \vdash e[\sigma/\alpha] : \rho \rightsquigarrow E[|\sigma|/\alpha]$$

This, together with another property of System F's contextual equivalence:

$$\forall \Gamma, \alpha, E, T, T' : \quad \Gamma \vdash_{\text{F}} (\Lambda\alpha.E)\,T : T' \ \Rightarrow \ \Gamma \vdash_{\text{F}} (\Lambda\alpha.E)\,T \simeq_{\text{ctx}} E[T/\alpha] : T'$$

allows us to conclude the correctness of static reduction of type application.

Theorem 4.4 (Correctness of Type Application Reduction)**.**
If a type application and its reduced form elaborate to two System F terms, those terms are contextually equivalent. Formally,

$$\forall \Gamma, \alpha, e, \sigma, E1, E2 :$$
$$\Gamma \vdash (\Lambda\alpha.e)\,\sigma : \rho \rightsquigarrow E_1 \ \wedge\ \Gamma \vdash e[\sigma/\alpha] : \rho \rightsquigarrow E_2 \ \Rightarrow\ |\Gamma| \vdash_{\mathrm{F}} E_1 \simeq_{\mathrm{ctx}} E_2 : |\rho|$$

The proofs for the stability Lemmas 4.2 and 4.3 are sketched in Section B.5 of the Appendix.

4.4.2 Evidence Generation in the Algorithm

The evidence generation in Figure 4.9 is largely similar to that in the deterministic specification of resolution in Figure 4.8. With the evidence we can state the correctness of the algorithm.

Theorem 4.5 (Partial Correctness)**.**
Let Γ be a type environment, ρ be a type and E be a System F expression. Assume that $\vdash_{\mathrm{unamb}} \rho$ and also $\vdash_{\mathrm{unamb}} \rho_i$ for all $?\rho_i \in \Gamma$. Then $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$ if and only if $\Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E$.

The proof is split in two parts, one for each "direction" of the theorem. Section B.7 of the Appendix sketches the proof for the soundness of the algorithm with respect to deterministic resolution, while Section B.8 sketches the proof of partial completeness. The latter correctness property is partial because it does not hold without the additional termination conditions for the queried type and the type environment, Γ.

4.4.3 Dynamic Semantics

Finally, we define the dynamic semantics of Cochis as the composition of the type-directed translation and System F's dynamic semantics. Following Siek & Lumsdaine's [2005] notation, this dynamic semantics is:

$$\mathit{eval}(e) = V \qquad \textit{where } \epsilon \vdash e : \rho \rightsquigarrow E \textit{ and } E \rightarrow^* V$$

with $\rightarrow^*$ the reflexive, transitive closure of System F's standard single-step call-by-value reduction relation (see Chapter 23 of Pierce 2002).

Now we can state the conventional type safety theorem for Cochis:

Theorem 4.6 (Type Safety)**.**
If $\epsilon \vdash e : \rho$, then $\mathit{eval}(e) = V$, for some System F value V.

The proof follows from Theorem 4.1 and System F's well-known normalization property.

4.5 Discussion

In this section we discuss and justify several of the design decisions made during the creation of Cochis. Mostly, these choices are motivated by the design of Haskell type classes or Scala implicits.

4.5.1 Predicative Instantiation

System F is an impredicative calculus, allowing the instantiation of type variables with arbitrary (polymorphic) types. In contrast Cochis is predicative: instantiation of type variables only allows *monotypes*. Our reasons for departing from System F are threefold:

- **Impredicative instantiation in resolution leads to additional ambiguity.** As discussed in Section 4.2, impredicative instantiations of type variables during resolution can lead to ambiguity. The restriction to predicative instantiation removes this ambiguity, and we see no way that preserves impredicativity to achieve the same.
- **Impredicativity is complex for implicit instantiation.** While System F (where all type instantiations are explicit) is simple, matters become much more complicated when some implicit instantiation is allowed. Indeed, the design of System F-like calculi with implicit instantiation and/or some form of type-inference [Le Botlan & Rémy 2003; Leijen 2008; Odersky & Läufer 1996; P. Jones et al. 2007; Vytiniotis et al. 2008] is much more divided in terms of design choices regarding (im)predicativity. Notably, Rouvoet [2016] has shown that the ambiguous resolution from the implicit calculus [Oliveira et al. 2012], which is the impredicative variant of our Figure 4.3, is undecidable. His proof proceeds by showing that the problem is equivalent to the System F type inhabitation problem, which is known to be undecidable [Barendregt et al. 2013].
- **Predicative instantiation is not a big restriction in practice.** Due to the above complications brought by impredicativity, many practical languages with type-inference only allow predicative instantiation. For example, the key algorithm for type-inference currently employed by the

GHC Haskell compiler is predicative [P. Jones et al. 2007; Vytiniotis et al. 2011]. Worth noting is that the original Hindley-Milner (HM) type system [Hindley 1969; Milner 1978] is where the predicativity restriction on polymorphic type systems with type-inference was first put into place. Since COCHIS is intended as a target for languages with implicit polymorphism and type-inference, which have predicativity restrictions, restricting the core language to allow only predicative instantiation does not pose any problems.

Alternative Design Choices One alternative design choice worth mentioning for COCHIS would be to allow impredicative instantiation in explicit type applications, but still retain the predicativity restriction during resolution. This design would be less restrictive than the design of COCHIS, and we believe that it is a reasonable design. We decided against it for two reasons. Firstly, as already mentioned, COCHIS is aimed to be a target for source languages with type-inference. As these source languages have predicative restrictions anyway, there is little to be gained from having impredicative instantiation in the core. Secondly, and more importantly, some of the meta-theory would be more involved if impredicative instantiation on type applications were allowed. In particular, Lemmas 4.2, 4.3 and Theorem 4.4 would need to be generalized to allow any types to be used in the substitution, rather than just monotypes. This could be problematic since the impredicative instantiations of type variables could bring back the ambiguity issues discussed in Section 4.2. We expect that additional restrictions would be needed at type applications to prevent instantiations with problematic polymorphic types that would lead to ambiguity.

Allowing full impredicativity (both in type applications and resolution) seems more complicated. We expect that such a design is possible, but certainly more complicated if ambiguity and undecidability are to be avoided. We expect that the work on impredicative type-inference [Le Botlan & Rémy 2003; Leijen 2008; Vytiniotis et al. 2008] is relevant, as some of the design choices in those works may be helpful in the design of a system with full impredicativity.

4.5.2 Committed Choice

COCHIS commits to the first implicit whose head matches the query type. It has inherited this committed choice approach from Haskell. Consider for instance the following Haskell program with two overlapping instances:

```
class C a where
  m :: a -> a
```

```
instance Eq a   => C [a] where ...
instance Ord a  => C [a] where ...

f :: StablePtr Int -> [StablePtr Int]
f sp = m [sp]
```

This code declares a type class `C a` and defines two instances. The first instance requires `Eq a`, whereas the second instance requires `Ord a`. The function `f` takes a stable pointer (`StablePtr`) and returns a list of stable pointers. Unlike many other types, `StablePtr` only supports equality, and not ordering. That is, there is an instance `Eq (StablePtr a)` but not one for `Ord (StablePtr a)`.

Should the above code type-check or not? In GHC Haskell the answer is no. Even though there is no ambiguity in this program— resolution only succeeds with the first type class instance— the program is nevertheless rejected. The reason is that Haskell's resolution only checks whether the instance heads match. As there are two equality specific matching heads `C [a]`, the program is rejected.

Although this Haskell design choice is not very well documented in the research literature, the reason for not allowing backtracking is folklore among Haskell programmers and can be found in various informal discussions [Bottu & simonpj 2018; oleg 2006; rHaskell 2006]. In essence there are two arguments for not allowing backtracking during resolution:

- **Reasoning:** When reasoning about Haskell code that involves type classes, programmers have to understand which instance is used. This involves performing the resolution algorithm mentally. The fact that only instance heads are needed to determine whether an instance is committed to, makes this much easier than performing a full backtracking process.
- **Performance:** If backtracking is allowed, type-checking times of programs could grow exponentially due to backtracking. Thus, by disallowing backtracking, GHC eliminates a potential source of significant performance degradation in type-checking.

No Backtracking in Cochis Like in Haskell, our committed choice design for Cochis also looks only at the heads of rule types, whereas a design with backtracking would need to look at their contexts as well. In other words we commit to a matching rule type, even if its recursive goals do not resolve. For instance, when resolving $Char$ against the environment $\Gamma = ?Bool, ?(Bool \Rightarrow Char), ?(Int \Rightarrow Char)$, we commit to $?(Int \Rightarrow Char)$ even though its recursive goal Int cannot be resolved and thus the resolution of $Char$ also fails. A more

permissive approach would be to backtrack when a recursive resolution fails and try the next alternative matching implicit. That would allow *Char* to resolve.

In the design of COCHIS, we have followed Haskell's pragmatic reasons for committed choice. Considering that Haskell's 30 years of experience have shown that this works well in practice, we believe that it is a reasonable choice.

Alternative Designs While there are advantages to committed choice, backtracking also has its appeal and several systems have adopted it [Sozeau & Oury 2008; White et al. 2015]. In particular, backtracking accepts more queries, and would allow us to have a sound and complete algorithm for COCHIS (instead of just a sound one) with respect to Figure 4.5. Yet, the specification in Figure 4.5 does not guarantee stability or coherence. So additional restrictions would be needed to have both properties.

OCaml's modular implicits [White et al. 2015] suggest another alternative. When a query is resolved, it exhaustively searches the implicit context for all possible solutions. If more than one solution is found, then the program is rejected due to ambiguity. In this way it is possible to have highly overlapping implicits in the type environment, that could result in some queries being ambiguous. One advantage of this design is its flexibility, since contexts can be more liberal and all queries that would be resolved in unambiguous contexts with backtracking can, in principle, also be resolved with OCaml's modular implicits. However the modular implicits approach is not formalized yet, and the fact that contexts have to be searched exhaustively raises practical questions regarding performance and ease of reasoning that have dictated the committed choice approach taken by Haskell type classes.

Finally, in the context of theorem provers like Coq [Sozeau & Oury 2008] where proof irrelevance typically holds, backtracking seems to be the better choice. If type classes are supplying proofs and it does not matter which proof is found, coherence is not relevant, and the objection about the difficulty of reasoning is also not relevant. Moreover, in theorem proving the expressiveness of search is often more important than having a very fast search method, and thus worse performance is more tolerable.

4.5.3 Superclasses

Like Scala's implicits design,[4] COCHIS does not directly support superclasses. While superclasses can be encoded in COCHIS to a certain extent, there are

[4]Superclasses are often simulated in Scala with OO Subtyping and class hierarchies, although there is no one-to-one correspondence between both.

several problems.

A First Attempt at Encoding Superclasses Recall the definition of the `Ord` type class as a subclass of `Eq` and the example function `p`, from Section 2.2:

```
class Eq a => Ord a where ...

p :: Ord a => a -> a -> Bool
p = (==)
```

Assuming also the `Eq Int` and `Ord Int` instances defined in Section 2.2, we can try to encode the `p` function in the COCHIS environment $\Gamma = ?(\forall\alpha.\mathit{Ord}\ \alpha \Rightarrow \mathit{Eq}\ \alpha) \rightsquigarrow \mathit{super}, ?(\mathit{Eq}\ \mathit{Int}) \rightsquigarrow x, ?(\mathit{Ord}\ \mathit{Int}) \rightsquigarrow y$, whose implicit entries capture the superclass relation—*super* is a function that extracts the *Eq* dictionary out of an *Ord* dictionary— and the two instances[5]. With respect to COCHIS, the query $?(\mathit{Eq}\ \mathit{Int})$ resolves deterministically by picking the second entry in Γ. Hence, COCHIS's explicit ordering of implicits avoids Haskell's non-determinism.

While the ordering is beneficial for determinism, fewer queries may succeed due to an unsuitable order of implicits. For example, suppose that we have instead the environment $\Gamma = ?(\mathit{Eq}\ \mathit{Int}) \rightsquigarrow x, ?(\forall\alpha.\mathit{Ord}\ \alpha \Rightarrow \mathit{Eq}\ \alpha) \rightsquigarrow \mathit{super}$. Then resolving $?(\mathit{Eq}\ \mathit{Int})$ fails because the first entry matches and its requirement, $\mathit{Ord}\ \mathit{Int}$, cannot be satisfied. In this case the committed choice semantics prevents reaching the second entry, which would resolve $\mathit{Eq}\ \mathit{Int}$ directly.

Superclasses with Committed Choice The situation we just saw also arises in Haskell. Treating superclass relations in the same way as type class instances does not work well with a committed choice strategy. That is why GHC treats superclass relations differently. In essence, whenever GHC adds a given type class constraint (e.g., from a programmer-supplied type signature) to the type environment, it uses the superclass relation to also add all of its ancestors. We believe that a similar strategy would be possible for COCHIS. For instance, in the last example, the type environment would be $\Gamma = ?\mathit{Eq}\ \mathit{Int} \rightsquigarrow x$, not contain the superclass relation and be able to resolve the query $?(\mathit{Eq}\ \mathit{Int})$ with x. When adding an $\mathit{Ord}\ \mathit{Int}$ entry, we would also add its $\mathit{Eq}\ \mathit{Int}$ superclass, yielding the modified environment $\Gamma = ?\mathit{Eq}\ \mathit{Int} \rightsquigarrow x, ?\mathit{Ord}\ \mathit{Int} \rightsquigarrow y, ?\mathit{Eq}\ \mathit{Int} \rightsquigarrow \mathit{super}\ y$. Now, the query $?(\mathit{Eq}\ \mathit{Int})$ is resolved with $\mathit{super}\ y$, which is the superclass value of the new $\mathit{Ord}\ \mathit{Int}$ entry.

[5]Note that the proposed encoding does not satisfy our termination conditions, but we ignore this aspect since there are other more pressing issues with the encoding.

Superclasses with Global Scoping and Cochis-style Resolution in GHC Haskell As we have explained above, standard Haskell gets away with committed choice in the presence of superclasses without unnecessarily rejecting queries. Yet, this is no longer the case for the recent extension with *quantified class constraints* [Bottu et al. 2017]. Like COCHIS, quantified class constraints allow nested implications in type class instance contexts and as superclasses. The committed choice approach does not work with the latter. GHC has decided to retain its committed choice approach and to reject queries that would require backtracking to explore the superclass relations.

4.5.4 Coherence

There are several ways to enforce coherence in a language design. For example, Haskell guarantees coherence by ensuring that there is a unique instance of a type class per type. In this way whenever code accesses a type class dictionary, it always returns the same (equal) dictionary value. This is a very strict way to enforce coherence.

COCHIS's way to achieve coherence is more relaxed than Haskell's. COCHIS enforces that the elaboration and resolution are deterministic but, under different scopes the same queries can resolve to different values (unlike Haskell).

While determinism is sufficient to ensure coherence, it is still a fairly strict way to ensure coherence. A more relaxed and general notion of coherence is to allow elaboration and resolution to have multiple different (but observationally equivalent) terms for the same expression. Our Corollary 4.1 provides a formal statement of coherence that is based on contextual equivalence of two expressions:

$$\forall \Gamma, \rho, E_1, E_2 : \qquad \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad |\Gamma| \vdash_{\mathrm{f}} E_1 \simeq_{\mathrm{ctx}} E_2 : |\rho|$$

This statement is close to the usual definition of coherence in the literature [Breazu-Tannen et al. 1991; Jones 1992; Reynolds 1991]. That is, E_1 and E_2 are not required to be syntactically equivalent, but they must be semantically equivalent. Many language designs that are coherent are often not necessarily deterministic (unlike COCHIS).

Alternative Designs We use determinism to establish coherence in COCHIS, but a more relaxed notion of coherence would also be possible. For example if, instead of committed choice, we decided to allow for a more powerful resolution strategy (for example with backtracking), then a more relaxed notion

of coherence would be helpful. This could be useful to deal with some situations that appear in superclasses. For example, consider again the context $\Gamma = ?(Eq\ Int), ?(\forall\alpha.Ord\ \alpha \Rightarrow Eq\ \alpha), ?(Ord\ Int)$. In this case the query $?(Eq\ Int)$ can be resolved in two possible ways: either going via the superclass instance *Ord Int*; or by directly using the instance *Eq Int*. Even thought the two elaborations are syntactically different, their semantics is the same.

4.6 Related Work

The most closely related work can be divided into three strands: IP mechanisms that support local scoping with coherence and stability, but forbid overlapping implicits and lack other types of flexibility; IP mechanisms that have global scoping and preserve coherence and stability; and IP mechanisms that are unstable and sometimes incoherent but offer greater flexibility in terms of local scoping and/or overlapping implicits. Cochis is unique in offering flexibility (local scoping with overlapping implicits, first-class implicits and higher-order rules), while preserving coherence and stability.

A fourth subsection is added to discuss stability in other contexts.

4.6.1 Implicit Programming with Local Scoping, but no Overlapping Rules

Our work allows a very flexible model of implicits with first-class implicits, higher-order rules and nested scoping with overlapping implicits while guaranteeing coherence and stability. Closest to our work in terms of combining additional flexibility with desirable properties (such as coherence and stability) are *modular type classes* [Dreyer et al. 2007] and System F_G [Siek & Lumsdaine 2005]. Both works preserve coherence and stability in the presence of local scoping, but (unlike Cochis) the local scopes *forbid overlapping implicits*. The restriction of no overlapping implicits is an essential part of guaranteeing coherence and stability. Cochis also has several other fundamental differences to both modular type classes and System F_G. *Modular type classes* [Dreyer et al. 2007] are a language design that uses ML-modules to model type classes. The main novelty of this design is that, in addition to explicit instantiation of modules, implicit instantiation is also supported. System F_G [Siek & Lumsdaine 2005] also offers an implicit parameter passing mechanism with local scoping, which is used for concept-based generic programming [Siek 2011]. Both mechanisms are strongly influenced by type classes, and they preserve some of the characteristics of type classes such as only allowing modules or concepts to be implicitly passed.

Moreover neither of those mechanisms support higher-order rules. In contrast COCHIS follows the Scala implicits philosophy and allows values of any type to be implicit, and additionally higher-order rules are supported.

Implicit parameters [Lewis et al. 2000] are a proposal for a name-based implicit parameter passing mechanism with local scoping. Implicit parameters allow *named* arguments to be passed implicitly, and these arguments can be of any type. However, implicit parameters do not support recursive resolution, so for most use-cases of type-classes, including the `Ord` and `Eq` instances for pairs in Sections 2.2 and 1.2.1, implicit parameters would be very cumbersome. They would require manual composition of rules instead of providing automatic recursive resolution. This is in stark contrast with most other IP mechanisms, including COCHIS, where recursive resolution and the ability to compose rules automatically is a key feature and source of convenience.

4.6.2 Implicit Programming with Local Scoping and Overlapping Implicits

The implicit calculus [Oliveira et al. 2012] is the main inspiration for the design of COCHIS. There are two major differences between COCHIS and the implicit calculus. The first difference is that the implicit calculus, like Scala, does not enforce stability. Programs similar to that in Figure 2.2 can be written in the implicit calculus and there is no way to detect instability. The second difference is in the design of resolution. Rules in the implicit calculus have n-ary arguments, whereas in COCHIS rules have single arguments and n-ary arguments are simulated via multiple single argument rules. The resolution process with n-ary arguments in the implicit calculus is simple, but quite ad-hoc and forbids certain types of resolution that are allowed in COCHIS. For example, the query:

$$?(\mathit{Char} \Rightarrow \mathit{Bool}), ?(\mathit{Bool} \Rightarrow \mathit{Int}) \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Char} \Rightarrow \mathit{Int}$$

does not resolve under the deterministic resolution rules of the implicit calculus, but it resolves in COCHIS. Essentially resolving such query requires adding the rule type's context to the implicit environment in the course of the resolution process. But in the implicit calculus the implicit environment never changes during resolution, which significantly weakens the power of resolution.

Rouvoet [2016] presents $\lambda^{S}_{\Rightarrow}$, which is a variation on the implicit calculus. The key feature of his calculus is the focusing resolution of Figure 4.5, although Rouvoet does not make the connection with focusing in proof search. As we have already explained in Section 4.2.5 this approach is both nondeterministic and unstable.

Scala implicits [Odersky 2010; Odersky et al. 2008] were themselves the inspiration for the implicit calculus and, therefore, share various similarities with COCHIS. In Scala, implicit arguments can be of any type and local scoping (including overlapping implicits) is supported. However the original model of Scala implicits did not allow higher-order rules. Recently, following the implicit calculus and a preliminary version of COCHIS, Odersky et al. [2017] presented the SI calculus as a new basis for the Scala language's treatment of implicits. Prominently, SI features implicit function types $T_1 ?\!\!\rightarrow T_2$, which are akin to rule types $T_1 \Rightarrow T_2$ in COCHIS, and implicit queries $?$, which are akin to $?_T$ in COCHIS. There are two main differences with COCHIS. Firstly, like the Hindley-Milner calculus SI is aimed at type inference and, e.g., does not feature explicit abstraction over implicits $\lambda_? T.e$ or type application $e\,T$ at the term level. In contrast, COCHIS is more similar to System F in this sense, making all abstractions and applications explicit.

Secondly, while COCHIS aims to formalise and investigate the meta-theory of resolution, the priority of Odersky et al. is not so much the SI calculus itself as the derived implementation of the Scala compiler. As a consequence, SI features a simplified type system that is incoherent and unstable and a resolution algorithm that supports only monomorphic types, while the compiler's much more complex enforcement of coherence and support for polymorphism are only discussed informally.

An interesting design of implicits has also been created in OCaml [White et al. 2015], where the implicit entities are OCaml modules. Like COCHIS, these implicits have local scope, but, unlike COCHIS, coherence is obtained by performing a backtracking search over all possible ways to resolve an implicit module signature, and fail if there is not exactly one way. Hence, while introducing overlapping implicits is allowed, they are only usable if only one leads to a full resolution. As far as we know, a partial prototype exists but the approach has not been formalised yet.

4.6.3 Implicit Programming with Global Scoping

Several core calculi and refinements have been proposed in the context of type classes. As already discussed in detail in Section 2.4, there are a number of design choices that (Haskell-style) type classes take that are different from COCHIS. Most prominently, type classes make a strong differentiation between types and type classes, and they use global scoping instead of local scoping for instances/implicits. The design choice for global scoping can be traced back to Wadler & Blott's [1989] original paper on type classes. They wanted to extend Hindley-Milner type-inference [Damas & Milner 1982; Hindley 1969; Milner

1978] and discovered that local instances resulted in the loss of principal types. For Haskell-like languages the preservation of principal types is very important, so local instances were discarded. However, there are many languages with IP mechanisms (including Scala, Coq, Agda, Idris or Isabelle) that have more modest goals in terms of type-inference. In these languages there are usually enough type annotations to avoid ambiguity introduced by local instances.

There have been some proposals for addressing the limitations that arise from global scoping [Dijkstra & Swierstra 2005; Kahl & Scheffczyk 2001] in the context of Haskell type classes. Both *named instances* [Kahl & Scheffczyk 2001] and *Explicit Haskell* [Dijkstra & Swierstra 2005] preserve most design choices taken in type classes (including global scoping), but allow instances that do not participate in the automatic resolution process to be named. This enables the possibility of overriding the compiler's default resolution result with a user-defined choice.

Jones's [1995] work on *qualified types* provides a particularly elegant framework that captures type classes and other forms of predicates on types. Like type classes, qualified types strongly distinguish between types and predicates over types, and scoping is global. Jones [1994] discusses the coherence of qualified types. The formal statement of determinacy in COCHIS essentially guarantees a strong form of coherence similar to the one used in qualified types.

The GHC Haskell compiler supports overlapping instances [P. Jones et al. 1997] that live in the same global scope. This allows some relief for the lack of local scoping. Yet, it still does not allow different instances for the same type to coexist in different scopes of a program. Moreover, GHC only commits to an instance if it is the only one whose head matches, and does not backtrack among multiple matching instances.

COCHIS has a different approach to overlapping compared to *instance chains* [Morris & Jones 2010]. With instance chains the programmer imposes an order on a set of overlapping type class instances. All instance chains for a type class have a global scope and are expected not to overlap. This makes the scope of overlapping closed within a chain. Instance chains can explicitly declare cases when resolution should fail, and these are dealt with by backtracking. In our calculus, we make our local scope closed, thus overlap only happens within one nested scope. More recently, there has been a proposal for *closed type families with overlapping equations* [Eisenberg et al. 2014]. This proposal allows the declaration of a type family and a (closed) set of instances. After this declaration no more instances can be added. In contrast our notion of scoping is closed at a particular resolution point, but the scopes can still be extended at other resolution points.

Inspired by the focusing approach of Cochis, Bottu et al. [2017] have extended Haskell's type class inference with *quantified class constraints*. This generalizes the syntax of Haskell's type class constraints to feature arbitrarily nested uses of universal quantification and implication. Their work differs from Cochis in that it does not support local instances. Moreover, they achieve coherence through requiring non-overlapping instances. Their algorithm performs a backtracking search among these instances and any local assumptions (which themselves can ultimately only be satisfied by combinations of global instances), rather than a linear committed-choice traversal of the environment.

IP Mechanisms in Dependently Typed Programming A number of dependently typed languages also have IP mechanisms inspired by type classes. Coq's type classes [Sozeau & Oury 2008] and canonical structures [Gonthier et al. 2011], Agda's instance arguments [Devriese & Piessens 2011] and Idris type classes [Brady 2015] all allow multiple and/or highly overlapping implicits/instances that can be incoherent. The Coq theorem prover has two mechanisms that allow modeling type-class like structures: *canonical structures* [Gonthier et al. 2011] and *type classes* [Sozeau & Oury 2008]. The two mechanisms have quite a bit of overlap in terms of functionality. In both mechanisms the idea is to use dependent records to model type-class-like structures, and pass instances of such records implicitly, but they still follow Haskell's global scoping approach. Nevertheless highly overlapping instances, which can be incoherent, are allowed and resolution performs a backtracking search. Like implicits, the design of Idris type classes, known as *interfaces*, allows for any type of value to be implicit. Thus type classes in Idris are first-class and can be manipulated like any other values. The language distinguishes unnamed instances, which are used for resolution, and named instances which have to be applied explicitly. The former cannot be overlapping, while there can be multiple (incoherent) named instances of the same type. The implicit resolution follows the committed choice strategy of Haskell, and ignores the fact that named instances can distinguish between alternative derivations. *Instance arguments* [Devriese & Piessens 2011] are an Agda extension that is closely related to implicits. Like Cochis, instance arguments use a special arrow for introducing implicit arguments. However, unlike most other mechanisms, implicits are not declared explicitly, but are drawn directly from the regular type environment, and any previously defined declaration can be used as an implicit. The original design of instance arguments severely restricted the power of resolution by forbidding recursive resolution. Since then, recursive resolution has been enabled in Agda. Like Coq's and Idris's type classes, instance arguments allow multiple incoherent implicits. Agda computes all possible resolutions and uses one of them only if all are equal.

4.6.4 Global Uniqueness and Same Instance Guarantee

Haskell type classes not only ensure coherence but also *global uniqueness* [E. Zhang 2014] (due to global scoping), as discussed in Subsection 2.2.3. Unrestricted COCHIS programs ensure coherence only, as multiple implicits that match the same type can coexist in the same program. We agree that for programs such as the *Set* example, it is highly desirable to ensure that the same ordering instance is used consistently. COCHIS is a core calculus, meant to enable the design of source languages that utilize its power. An example are Bottu et al.'s [2017] quantified class constraints for Haskell, which forbid local scoping constructs and, instead, make all declared implicits visible in a single global environment. This retains several of the benefits of COCHIS (such as first-class implicits, higher-order rules, and coherent overlapping implicits), while providing a form of global uniqueness. However this design is still essentially non-modular, which is a key motivation for many alternatives to type classes to provide local scoping instead.

Global uniqueness of instances is just a sufficient property to ensure consistent uses of the same instances for examples like *Set*. However, the important point is not to have global uniqueness, but to consistently use the same instance. COCHIS admittedly does not provide a solution to enforce such consistency, but there is existing work with an alternative solution to deal with the problem. Genus [Y. Zhang et al. 2015] tracks the types of instances to enforce their consistent use. For instance, in Genus two sets that use different orderings have different types that reflect which `Ord` instance they use. As a consequence, taking the union of those two sets is not possible. In contrast to COCHIS Genus is focused on providing a robust source language implementation for generic programming. Although the Genus authors have proved some meta-theoretic results, neither type-safety nor coherence have been proved for Genus. In dependently typed languages such as Agda and Idris, it is possible to parametrize types by the instances they use [Brady 2015]. This achieves a similar outcome to Genus's approach to consistent usage of instances. Investigating the applicability of a similar approach to COCHIS is an interesting line of future work.

4.6.5 Focused Proof Search

Part of the syntax-directedness of our deterministic resolution is very similar to that obtained by *focusing* in proof search [Andreoli 1992; Liang & Miller 2009; Miller et al. 1991]. Both approaches alternate a phase that is syntax directed on a "query" formula (our focusing judgment), with a phase that is syntax directed on a given formula (our matching judgment). This is as far as

the correspondence goes though, as the choice of given formula to focus on is typically not deterministic in focused proof search.

4.6.6 Stability

Bottu & Eisenberg [2021] study stability in systems with parametric polymorphism that feature implicit as well as explicit type instantiation, but not a resolution mechanism. They identify a number of code transformations that may seem innocuous, but can actually change a program's semantics depending on the design choices regarding type variable instantiation. Their work is complementary to ours, since in our work we study stability in the context of implicit programming, where we assume an explicitly typed language, while Bottu & Eisenberg study stability in the context of type inference and do not take implicit programming into account. Furthermore, in our system instability can be eliminated by prohibiting unstable code, while Bottu & Eisenberg conclude that there is no concrete design choice to entirely prevent instability. In their scenario, the best solution is to choose for a design that preserves semantics under most (but not all) of the identified code transformations.

4.7 Scientific Output

The material found in this chapter is drawn from the following publication:

> Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. Journal of Functional Programming, 29, E3. DOI: `https://doi.org/10.1017/S0956796818000242`

The above is multi-author work, where my contribution is situated mainly in the meta-theory of COCHIS. An earlier version of this work was using paper proofs (i.e. proofs written in paper). I mechanized the calculus and proved part of its correctness properties in the Coq language. In addition, I identified and corrected the paper proofs of some lemmas that contained errors. The mechanization of COCHIS is available at `https://bitbucket.org/KlaraMar/cochiscoq`.

Chapter 5

Resolution as Intersection Subtyping via Modus Ponens

5.1 Introduction

Subtyping is a well-known programming language feature. One particular form of subtyping arises from *intersection types* [Barendregt et al. 1983; Reynolds 1991]. Intersection types support—among others—a form of *finitary overloading* [Castagna et al. 1992]. For example, in various systems with intersection types, it is possible to build overloaded functions such as:

```
f : (Int -> Int) & (Char -> Char) = succ ,, toUpper
```

The *merge operator* (`,,`) allows the function `f` to behave as both the `succ` and the `toUpper` functions. Thus, `f` can be applied both to an integer and to a character. When applied to an integer, `f` behaves like the successor function, whereas when applied to a character `f` capitalizes its input. The automatic selection of the appropriate behavior (from the overloaded definition `f`) is based on the type of the input, which triggers an upcast induced by the subtyping relation. For instance, in the case of the application `f 3`, the input of the function must be `Int` and therefore, the following upcast is triggered in such an application:

```
(Int -> Int) & (Char -> Char) <: Int -> Int
```

Resolution is another programming language mechanism that can also be used to define overloaded functions. Resolution is used in Haskell type classes [Wadler & Blott 1989], Scala implicits [Oliveira et al. 2010] and in similar mechanisms

of many other programming languages [Devriese & Piessens 2011; Sozeau & Oury 2008; White et al. 2015]. For example, in Haskell, we may define:

```
class Over a where f :: a -> a
instance Over Int where f = succ
instance Over Char where f = toUpper
```

This achieves an overloaded behavior similar to the intersection types version of `f`. Indeed, the Haskell version of `f` can also be applied both to an integer and to a character with similar results. However, in Haskell, the selection of the right function in the application `f 3` uses resolution.

The function `f` illustrates the similarities and overlapping use cases for subtyping and resolution. However, there are also various use cases that are not in common. On the one hand, with subtyping we can, for instance, have `Int -> Int <: Int -> Top`. Such a conversion is typically not allowed by resolution mechanisms documented in the literature [Hall et al. 1996; Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019; Wadler & Blott 1989]. On the other hand, a key feature of resolution is its ability to compose instances (or implicits). For example, with type classes we can define the following instance to create an overloaded version of `f` for lists of integers:

```
instance Over Int => Over [Int] where
  f []       = []
  f (x : xs) = f x : f xs
```

Note that here the context `Over Int` is not necessary in Haskell, but for illustration purposes, we add the redundant context. With this instance, if we call `f [1,2,3]`, then the result is `[2,3,4]`. For this to work, resolution needs to find an instance for `Over [Int]`. While no instance directly matches the required type, resolution composes the instance for lists with the instance for integers to build an instance of the appropriate type. The ability to perform such compositions is lacking in traditional mechanisms with subtyping and intersection types.

This chapter shows that, with a small extension, subtyping with intersection types can in fact subsume forms of resolution. The extension that is necessary to recover the power of resolution via subtyping is essentially the logical rule for *modus ponens*:

$$\frac{A \vdash B \to C \qquad A \vdash B}{A \vdash C}\ \textsc{Modus Ponens}$$

Modus ponens states that if from A one can conclude both $B \to C$ and B then one can also conclude C. The key idea in this chapter is that we can adapt modus ponens to subtyping, and thus enable its power (and the essential power

of resolution). To accomplish this, we simply reinterpret entailment ($A \vdash B$) as subtyping ($A <: B$). Modus ponens captures the essence of composition that is necessary to enable the following derivation:

```
(Int -> Int) & ((Int -> Int) -> [Int] -> [Int]) <: [Int] -> [Int]
```

which could be used in a language with intersection types and subtyping, to find an overloaded implementation of `f` in a call such as `f [1,2,3]` (just like the version with type classes).

While modus ponens is easy to add in a declarative version of subtyping, retaining desirable properties, such as transitivity and decidability of subtyping, need significant care. Our work builds and extends upon the NeColus calculus [Bi et al. 2018]—or λ_i^+—which has both transitivity and decidability of subtyping, as well as coherence of the elaboration semantics. However, NeColus does not support modus ponens. Extending the proofs of transitivity, decidability and coherence of NeColus to support modus ponens is highly non-trivial. The key difficulty is that modus ponens makes the context in which a type appears relevant for resolution (which is not the case in NeColus).

Adding modus ponens to subtyping, and viewing resolution as a special case of subtyping, comes with important advantages. Firstly, resolution does not need to be implemented as a completely separate mechanism. This can make implementations as well as metatheory more compact, because the two mechanisms are generalized into a single relation. Secondly, the interaction between resolution and subtyping becomes apparent, which contrasts with the current state-of-the-art where it is ignored.[1] Indeed, various foundational calculi related to Scala formalize either subtyping [Amin et al. 2016, 2012] or implicits [Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019], but usually not both. The only exception is the recent work by Jeffery [2019], which studies a calculus with both features (but where resolution and subtyping are modeled independently). Finally, the integration of resolution into subtyping enables *first-class (implicit) environments*. That is, we can create (implicit) environments (encoded as intersection types), use them as arguments, or return them from functions. This is impossible in existing approaches for type classes and calculi with implicits, as there is no explicit notion of a type for environments.

To materialize these ideas we develop λ_i^{MP}: a calculus that extends the NeColus calculus (a calculus with disjoint intersection types [Oliveira et al. 2016]) and develop its metatheory in the Coq theorem prover. In summary, the contributions of this work are:

[1] For example, GHC issue ticket 17295 is a closely related problem, caused by an unexpected interaction between resolution and overlapping "intersections".

- **Resolution as subtyping:** We extend subtyping of intersection types with *modus ponens*. This enables composing components from intersections to solve subtyping problems, and (together with other standard intersection rules) subsumes forms of resolution.
- **The λ_i^{MP} calculus:** We develop a concrete calculus that illustrates the idea of resolution as subtyping. The calculus is an extension of NeColus [Bi et al. 2018]. We prove several basic properties, including type-safety, which is proved via an elaboration into the target language λ_c^{MP}, a simply typed lambda calculus extended with coercions.
- **Algorithmic subtyping for λ_i^{MP}:** We incorporate modus ponens in the NeColus subtyping algorithm by (1) restructuring it in two mutually recursive judgments—one of which builds coercions by accumulating them in a continuation-passing form—and (2) adding a loop detector for termination. Our Coq proofs show that our algorithm subsumes that of NeColus, and is *sound*, *complete* and *terminating*; this establishes that the subtyping of λ_i^{MP} is decidable.

  The proofs of termination and transitivity, which is used to show completeness, are both highly non-trivial. The former uses different arguments for three different kinds of loops, including reasoning about the effectiveness of loop detection. The latter consists of 7 mutually recursive lemmas with complex interdependencies. To establish the well-foundness of their proof we employ the AProVE tool [Giesl et al. 2017].
- **Conjectured Coherence for λ_i^{MP}:** Establishing coherence of λ_i^{MP} is also a challenging technical goal. The addition of modus ponens makes it a lot harder due to the ability to compose information. We formulate a non-trivial extension of the canonicity relation of NeColus, based on which we sketch a partial coherence proof and mechanize it in the Coq theorem prover. The proof's partiality comes from two conjectures that are necessary to complete it.

The supplementary material contains a Coq mechanization of the definitions (4k lines) and theorems (5.6k lines) in this chapter, as well as a prototype implementation in Haskell (~820 lines). It is available online at `https://zenodo.org/record/4043646`.

An extended version of this work, containing appendices, can be found online, by following this link: `https://arxiv.org/abs/2010.06216`. Throught this chapter, references to the Appendix point to appendix sections of the extended version of this work.

5.2 Background

This section introduces background on resolution, coercive subtyping, intersection types and their use in programming languages.

5.2.1 Coercive Subtyping, Resolution and Their Computational Interpretation

This work brings together two different programming language features: (implicit) *resolution* as used in Haskell type classes and Scala implicits on the one hand, and *type-directed coercive subtyping* on the other hand. While these two features are very different on the surface, they are in fact both based on similar computational interpretations of logical inference. The different rules of logical inference they use are compatible. We show how to extend this compatibility to the language design level and how to ensure properties that, while typically of no concern for logical inference, are key for programming languages, like coherence and decidable type checking.

In the sequent notation, the judgment $A \vdash B$ asserts that proposition B can be inferred from proposition A. An inference rule provides a way to construct a valid (or true) judgment. For instance, the inference rule

$$\frac{}{A \vdash A}$$

states that the judgment $A \vdash A$ (from A follows A) holds for any proposition A.

The $A \vdash B$ judgment can be given a computational interpretation following the "propositions as types" approach [Wadler 2015]. This means that we take the types of a programming language as the possible propositions. The judgment $A \vdash B$ then means that there exists a computable function that turns values of type A into values of type B. We call this function the (computational) evidence for the judgment. The evidence for a judgment follows from the inference rules that have been used to establish that judgment. For instance, the evidence for the trivial $A \vdash A$ judgment above is the identity function $\lambda x.x$. A more advanced inference rule is that for transitivity

$$\frac{A \vdash B \qquad B \vdash C}{A \vdash C}$$

which allows to conclude C from A provided that B can be concluded from A and C from B. Given evidence f for $B \vdash C$ and evidence g for $A \vdash B$, the evidence for $A \vdash C$ is $\lambda x.g\,(f\,x)$.

Coercive Subtyping A first programming language application of this mechanism is coercive subtyping [Luo 1999]. In this case we write $A <: B$ instead of $A \vdash B$ and call A and B subtype and supertype respectively, rather than premise and conclusion. Also we call the associated evidence of the judgment a coercion. Unlike other forms of subtyping, coercive subtyping changes the runtime representation of a value of type A when upcasting to type B. This change is accomplished by the coercion. Even though the coercion is executed at runtime, it is already statically known at what point in the program it needs to be performed, namely at the point in the static type system where an appeal is made to subtyping. In order to introduce the coercion, coercive subtyping can therefore make use of a static program transformation approach called (type-directed) elaboration.

Elaboration uses the information of static typing to transform a program with implicit appeals to subtyping into a program with explicit applications of coercions. For example, given a value v of type A and a function f that expects a value of type B, where $A <: B$, the function call $f\,v$ has a type mismatch that the type system bridges by appealing to subtyping. Elaboration then transforms this function call into $f\,(c\,v)$ where c is the coercion for $A <: B$. The explicit coercion application replaces the implicit appeal to subtyping.

Resolution A rather different application of logic inference in programming languages are implicits in Scala. In their basic form, they involve an environment of premises that the programmer can populate and extend with particular types and values of those types as evidence. Functions can take implicit parameters which the type system fills in by searching values of the appropriate type in the implicit environment. Elaboration turns those implicit parameters into explicit ones. This behavior is extended with the modus ponens inference rule, to make it more powerful.

The computational evidence associated to this rule, shown in Section 5.1, is $\lambda x.(f\,x)\,(g\,x)$ where f is the evidence for $A \vdash B \to C$ and g for $A \vdash B$. The modus ponens rule can be applied recursively, e.g. to establish B from A by modus ponens via $D \to B$ and D, and we call the general reasoning principle *recursive resolution*.

Resolution in Scala As a concrete example of resolution in programming languages, consider the judgment

$$P \to Q, Q \to U, P \vdash U$$

In this statement, P implies Q, Q implies U and P holds as a fact. Thus, we can conclude that U holds by straightforward logical reasoning.

```
1  object Test {
2    class P                                          // proposition P
3    class Q                                          // proposition Q
4    class U                                          // proposition U
5
6    implicit def f(implicit o : P) : Q = new Q()     // P -> Q
7    implicit def g(implicit o : Q) : U = new U()     // Q -> U
8    implicit val p : P = new P()                     // P
9
10   val u : U = implicitly[U]            // triggers resolution from
11                                        // the implicit environment
12 }
```

Example 5.1: Encoding of the judgment $P \to Q, Q \to U, P \vdash U$ as a Scala program

Example 5.1 illustrates how to encode the judgment in Scala. In this Scala program, we encode propositions as classes, and we define a few implicit rules that are added to the "implicit environment" of Scala. The definition of `f` encodes $P \to Q$, the definition of `g` encodes $Q \to U$ and the definition of `p` encodes that P holds. In the definition of the value `u`, we trigger resolution (via the `implicitly` keyword), which essentially searches the implicit environment and attempts to compose the rules and facts in the implicit environment, to find a value of type `U`. This value will be given eventually by the synthesized expression `g(f(p))`.

Essentially, in logic, resolution can be seen as a mechanism for proof search. In programming languages, resolution is a mechanism that searches for values of a given type by using the rules and facts of the implicit environment. It can be viewed as a type-directed form of program synthesis.

The connection to our work is that we interpret the resolution relation as subtyping, and the conjunction of propositions in the environment as intersections. Thus, the above judgment corresponds to:

$$(P \to Q) \,\&\, (Q \to U) \,\&\, P <: U$$

using subtyping and intersection types.

Haskell type classes also employ resolution, but there are several superficial differences to Scala implicits. The name spaces of implicit values—called type classes—is separate from that of other types and has global scope. Moreover type classes are used as a mechanism for overloading function names. Yet, in essence,

just like Scala implicits, type classes are based on the modus ponens inference rule for constructing wanted evidence for types in terms of given evidence of other types. We will look at type classes in more detail in Section 5.3.

5.2.2 Different Interpretations for Intersection Types in the Literature

There are several different interpretations of intersection types in the literature. Intersection types date back to work from Coppo et al. [1981] and Pottinger [1980] and were first used to characterize the strongly normalizable terms of the λ-calculus.

Following Forsythe [Reynolds 1988], intersection types were used in various other programming languages, including CDuce [Benzaken et al. 2003] and Object Oriented languages like TypeScript[2], Flow [Raskin 1974], Ceylon[3] and Scala [Odersky 2010], enabling multiple inheritance [Pierce 1991]. They were also studied as part of the syntax of several calculi, to enable various forms of polymorphism such as finitary function overloading [Castagna et al. 1992], bounded polymorphism [Pierce 1997] and evaluation-order polymorphism [Dunfield 2015]. Refinement types is yet another feature where intersection types has been studied [Davies & Pfenning 2000; Dunfield 2007; T. Freeman & Pfenning 1991]. However, in those systems the type Int & Bool is invalid, since Int and Bool are not refinements of each other.

Note that intersection types alone only increase the expressive power of types. While languages with intersection types do not necessarily have a merge operator, the latter adds expressive power to the terms. This term construct has been used in various, but usually limited, forms like in Forsythe or in the work of Castagna et al. [1992], discussed in Section 5.7.

The use of intersection types in this work follows the interpretation of Dunfield [2012] who has proposed a particular flavor of intersection types with an unrestricted merge operator, allowing the construction of values that inhabit intersection types such as Int & Bool. In her work, Dunfield uses coercive subtyping, based on the inference rules for the logical conjunction operator. In particular, conjunction elimination is expressed in two rules:

$$\frac{}{A \wedge B \vdash A} \qquad \frac{}{A \wedge B \vdash B}$$

Here, we write $A\&B$, instead of $A \wedge B$, to denote an intersection type. The runtime representation of values of type $A\&B$ is a tuple (v, w) where v has type

[2] https://www.typescriptlang.org/
[3] https://ceylon-lang.org/

A and w has type B. The coercions associated with the two inference rules are respectively π_1 and π_2, the projection on respectively the first and second component of the tuple. Dunfield observes that her formulation of the calculus does not eliminate ambiguous expressions. However, recent work by Oliveira et al. [2016] remedies this issue by introducing a notion of disjoint intersection types.

Our proposed design fits well with this interpretation, i.e. intersection types as a mechanism to implicitly provide coercions on pairs. Our source language provides a layer of convenience on top of an ordinary target language with pairs, by generating various types of coercions involving pairs. In particular, the modus ponens rule is just one of the possible coercions involving pairs.

5.3 Overview

This section gives an overview of our work. We are mainly interested in the *type-based overloading* and *resolution* capability at the core of type classes or implicits. In what follows, we use Haskell type classes for illustration, but we could have used Scala implicits or other similar mechanisms just as well. We first establish the relationship between type-based overloading, using type classes and/or implicits, and disjoint intersection types. Then, we explain the key idea of our work to add modus ponens to the subtyping relation for a calculus with (disjoint) intersection types to recover resolution. Finally, we discuss several technical challenges that need to be addressed for this addition to work.

5.3.1 The Merge Operator and Disjoint Intersection Types

The merge operator, originally proposed by Reynolds [1988], adds expressive power to intersection types. In most calculi with intersection types, a type such as `Int & Char` is not inhabited, since these two types are *disjoint* (i.e. they have no common values), so their intersection is empty. With the merge operator, however, it is possible to build values that inhabit such an intersection type. For example:

```
v : Int & Char = 1 ,, 'c'
```

In the definition of `v`, the merge operator (`,,`), is used to build a value of type `Int & Char` from an integer and a character. In calculi with a merge operator, intersection types are similar to conventional pair types. The merge operator is used to build values of the intersection types, similarly to how we build values for pairs. The main difference to pairs and product types is that while for

pairs the eliminations (i.e. projections such as `fst` and `snd`) are explicit, for intersection types they are implicit and driven by the type system.

Disjointness and ambiguity The merge operator is powerful, but without any restrictions it is easy to have ambiguous programs. For instance, consider the following value:

```
n : Int & Int = 1 ,, 2
```

If we use `n` in an expression such as `n + 1` there could be two possible results (`2` and `3`) depending on what integer gets selected from the merge.

Disjoint intersection types [Oliveira et al. 2016] were proposed to solve ambiguity problems in calculi with intersection types and a merge operator. The key idea is to introduce a restriction on merges: a valid merge can only be built from two values with disjoint types. The definition of disjoint types is based on that of top-like types. Both definitions are taken from previous work [Oliveira et al. 2016] and are repeated here.

Definition 5.1 (Disjoint Types)**.**
Two types are disjoint if their only common supertypes are top-like.

Definition 5.2 (Top-like Types)**.**
Top-like types, denoted as $⌉A⌈$*, are specified by the following inductive rules.*

$$\textit{TL-TOP}\ \frac{}{⌉\top⌈} \qquad \textit{TL-AND}\ \frac{⌉A⌈ \qquad ⌉B⌈}{⌉A \,\&\, B⌈} \qquad \textit{TL-ARR}\ \frac{⌉B⌈}{⌉A \to B⌈}$$

The disjointness restriction prevents the above merged value `n` from type-checking, while accepting `v`. Furthermore, for previous calculi with disjoint intersection types it has been proven that the disjointness restriction is sufficient to guarantee coherence [Reynolds 1991].

5.3.2 Disjoint Merges, Non-Overlapping Instances and Resolution

Disjointness checks for merges and non-overlap checks for type class instances are closely related. The non-overlap check for type classes can be viewed as a simple form of disjointness checking. If we ignore polymorphic types, then the

non-overlap check is simply testing whether two types are distinct[4], which is subsumed by a disjointness check for merges.

Type class instances as merges The correspondence between disjointness checking and non-overlapping checking is crucial for expressing code analogous to type class instances as merges. Consider a variant of `f` in Section 5.1:

```
f : (Int -> Int) & (Char -> Char) & ((Int -> Int) -> [Int] -> [Int])
  = succ ,, toUpper ,, listInstance
```

where `listInstance` is analogous to the type class instance `Over [Int]` from Section 5.1, with the type `((Int -> Int) -> [Int] -> [Int])`. Here, the merge creates an overloaded function and checks that the three implementations have disjoint types. This is similar to checking whether `Over Int`, `Over Char` and `Over [Int]` overlap. Therefore, we can essentially express type class instances through merges, whose components define the methods of the instances for overloading.

Resolution via Modus Ponens While we can express some type class instances with disjoint intersection types, an important "class" of instances cannot be expressed because previous work on disjoint intersection types lacks resolution. That is, in previous work a program such as:

```
p : [Int] = f [1,2,3]
```

would not type check because `[Int] -> [Int]`, which is the type that `f` needs to have, *is not* a supertype of the type of `f`. However the addition of modus ponens changes this and `[Int] -> [Int]` becomes a supertype of the type of `f`, thus enabling the above use of `f`.

5.3.3 First-Class Environments

Viewing instances/implicits as merges opens up the possibility of new and interesting features for programming with implicit values and resolution. In particular, unlike type class instances (which are second class), merges are *first-class*. In other words, we can pass merged values as arguments or return them as results. This is not possible with type class instances. An instance declaration is top-level, but we cannot create instances on the fly[5].

[4] With polymorphism we also need to account for instantiation. For example the type `[Int]` is not equal to `[a]`, but it does instantiate `[a]`. So two instances `C [a]` and `C [Int]` (for some type class `C`) overlap.

[5] Though recent work by Winant & Devriese [2018] aims to rectify this.

To illustrate the use of first-class environments, consider a program that processes data from a sensor measuring temperatures. The temperatures are collected at fixed time intervals, and the sensor may fail to read temperatures (for example due to temperatures being too high, low or to some unknown error). The sensor data is collected in a text file using a sequence of numbers or strings (for errors) separated by commas. For example,

```
32,-9,"LOW",10,"ERROR"
```

With Haskell type classes one may be tempted to use the `Read` type class:

```
class Read a where
   read :: String -> a
```

to process the input string using the existing instances of the `Read` class:

```
instance Read Int where ...
instance Read String where ...
instance Read a => Read [a] where ...
instance (Read a, Read b) => Read (Either a b) where ...
```

The programmer's hope here would be that we could interpret the type of sensor values as `[Either String Int]`, and then simply employ resolution to compose the above instances:

```
read "32,-9,\"LOW\",10,\"ERROR\"" :: [Either String Int]
```

However this does not work because the input string is not in the correct format to be parsed by the `Read` instances. In Haskell the `Read` instances for types such as `Either a b` and `[a]` use the Haskell syntax[6]. The `read` method provided by `Read [Either String Int]` would require the string:

```
"[Right 32, Right -9, Left \"LOW\", Right 10, Left \"ERROR\"]"
```

Because Haskell does not allow more than one instance of a type class for the same type, it is not possible to use alternative instances for `Either a b` and `[a]`. There are various workarounds for this issue, but none of these is ideal.

A solution using first-class environments An alternative approach to solve this problem is to use merges to encode first-class environments for instances of `Read`. Example 5.2 presents such a solution using the λ_i^{MP} calculus, where we assume a few convenient source language features such as type aliases and built-in types for lists and sum types (i.e. modeling, respectively, Haskell's

[6]The Haskell format requires that strings to be read as `Either a b` values start with a `Left` or `Right` substring. For list values, the string should start and end with opening and closing brackets.

```
1  type Read a = String -> a
2
3  readI    : Read Int = ...
4  readS    : Read String = ...
5  readESI  : Read String -> Read Int -> Read (Either String Int) = ...
6  readLESI : Read (Either String Int) -> Read [Either String Int] = ...
7
8  type Sensor = Read Int & Read String
9              & (Read String -> Read Int -> Read (Either String Int))
10             & (Read (Either String Int) -> Read [(Either String Int)])
11
12 env : Sensor = readI ,, readS ,, readESI ,, readLESI
13
14 parseSensor (read : Sensor) (s: String) : [Either String Int] = read s
```

Example 5.2: A solution using first-class environments

`List a` and `Either a b`). Line 1 defines the `Read` type alias. Lines 3 - 6 model the four Haskell instances for `Read`. Here, `readI` and `readS` behave just like the corresponding Haskell instances, while `readESI` and `readLESI` are custom instances that read values in the sensor format. Line 12 uses the merge operator to define a sensor environment, `env`, as a collection of instanes that read sensor data. The type `Sensor` of this environment, define in lines 8-10, is the intersection of its instances' types. Finally, the `parseSensor` function, in line 14, parametrizes over a suitable sensor environment.

Calling `parseSensor` with `env` can parse strings in the expected format for sensor data. Moreover, an additional advantage of the `parseSensor` function is that it is *decoupled* from concrete implementations of `Read`. For instance, if later we want to express sensor data using Haskell's own format for `Read` instances, we just have to provide an alternative environment:

```
haskEnv : Sensor = readI ,, readS ,, readEither ,, readList
```

where `readEither` and `readList` behave just like the Haskell instances for `Read (Either a b)` and `Read [a]`. By calling `parseSensor` with `haskEnv` as its first argument, the system will infer an instance of the same type, but taken from the newly provided environment encoding, `haskEnv`.

As the code in Example 5.3 shows, without the modus ponens rule in subtyping, the same functionality would require a more verbose program with both the provided environment and the parser function parametrized over the alternative instances for `Either String Int` and `[Either String Int]`, and with explicit application of instances.

```
type Sensor' = Read Int & Read String & Read (Either String Int)
             & Read [Either String Int]
type GSensor =  (Read Int -> Read String -> Read (Either String Int)))
             -> (Read (Either String Int) -> Read [Either String Int])
             -> Sensor'

env' (rESI  : Read Int -> Read String -> Read (Either String Int))
     (rLESI : Read (Either String Int) -> Read [Either String Int])
 : Sensor'
 = readI ,, readS ,, (rESI readI readS) ,, (rLESI (rESI readI readS))

 parseSensor' (rESI  : Read Int -> Read String
                       -> Read (Either String Int))
              (rLESI : Read (Either String Int)
                       -> Read [Either String Int])
              (read : GSensor) (s : String)
  : [Either String Int]
  = read rESI rLESI s
```

Example 5.3: A solution without first-class environments

```
class A {}
class B extends A {}

implicit val v : B = new B() {};  // adds v to the implicit environment
val u : A = implicitly            // resolves to v
```

Example 5.4: A simple Scala program using both subtyping and resolution

In summary, modus ponens captures the essence of resolution and brings convenience to programmers by automatically composing instances. This is also the reason why resolution is heavily used by Haskell and Scala programmers. In addition, the integration of modus ponens into a system with intersection subtyping adds expressive power that is not available in Haskell or Scala. In contrast to Haskell, implicit environments are first-class. With respect to Scala, viewing resolution as subtyping makes obvious how the two features interact, which we discuss in more detail next.

5.3.4 Interaction Between Subtyping and Resolution

In languages like Scala, which supports both subtyping and resolution, we expect that the two mechanisms interact. Consider, for instance, the simple Scala program in Example 5.4 which requires both mechanisms. This program defines two classes `A` and `B`. The value `v` is marked with the `implicit` keyword, thus adding `v` to the implicit environment, so that it can be used by resolution later. In the definition of `u`, `implicitly` triggers resolution to search for a value of type `A`. The search succeeds and resolves to `v` which has type `B`, a subtype of `A`. Clearly this only works if resolution accounts for subtyping.

The Scala compiler accepts this program and its implementation of resolution does indeed account for subtyping. However, there is very little work formalizing the (non-trivial) interaction between resolution and subtyping. Most foundational calculi for Scala formalize subtyping, but not resolution [Amin et al. 2016, 2012]. Similarly, works that formalize resolution typically do not account for subtyping [Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019].

One advantage of making resolution a part of subtyping is that the interaction between the two features becomes trivial. For instance, the following program:

```
a : Int                  = 1
b : Int & Char           = a ,, 'c'
u (env : Int&Char) : Int = env
```

is roughly analogous to the Scala program in Example 5.4 (with classes `A` and `B` corresponding to types `Int` and `Int&Char` of the values `a` and `b`, respectively), except that we use a first-class environment (the argument `env` of function `u`) instead of an implicit environment. In this case, resolution triggers a search for an integer in the body of `u`, which is successful because `Int & Char <: Int`. Other more complex interactions between subtyping and resolution, such as in the following judgment, work as well.

```
(Int -> Top -> Bool) & Int <: String -> Bool
```

Here, the two types in the intersection need to be composed with modus ponens to obtain `Top -> Bool` and `Top -> Bool <: String -> Bool` via subtyping.

5.3.5 Technical Challenges

To formalize our integration of resolution and subtyping, we create the λ_i^{MP} calculus by extending the NeColus calculus with the modus ponens rule. While the extension is seemingly a small change, it has far-reaching consequences

Types	A, B	$::=$	$\mathsf{Int} \mid \top \mid A \to B \mid A \,\&\, B$
Terms	E	$::=$	$x \mid \top \mid i \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,, E_2 \mid E : A$
Typing Contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$

Figure 5.1: λ_i^{MP} Syntax

with respect to the metatheory. Conceptually, this is due to the fact that modus ponens invalidates much of the reasoning about types in the NeColus metatheory. In NeColus it is possible to consider the syntactic components of types in isolation. This is no longer possible in λ_i^{MP} because one component may interact with another through modus ponens.

This affects coherence: the NeColus disjointness constraint is insufficient for λ_i^{MP} and needs to be tightened. Also the coherence proof requires a considerably more complex canonicity relation that features a form of context dependency to capture the possible interaction between type components.

Algorithmic subtyping is also affected by this. The algorithm no longer terminates naturally and requires a loop detector. This makes the termination proof a lot harder. Moreover, the size-based induction used to prove transitivity for the algorithmic subtyping of NeColus breaks. Instead, our transitivity proof involves 7 mutually recursive lemmas and relies on the AProVE automatic termination checker to establish their well-foundedness.

The following sections set up λ_i^{MP} and explain these aspects in detail.

5.4 The λ_i^{MP} Calculus

Syntax Figure 5.1 shows the syntax of λ_i^{MP}. It is the same as that of NeColus, but—for the sake of conciseness—without records. Types A, B, C include naturals Int, the top type $\top$, function types $A \to B$ and intersection types $A \,\&\, B$. Terms E include variables x, natural numbers i, the top value $\top$, function abstractions $\lambda x.\, E$, function applications $E_1\, E_2$, merges $E_1 \,,, E_2$ and (type-)annotated terms $E : A$.

Bidirectional Typing The type system of λ_i^{MP}, shown in Figure 5.2, is bidirectional and thus has two modes: the *synthesis* mode ($\Rightarrow$) and the *checking* mode ($\Leftarrow$). The synthesis judgment $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ means that we can synthesize type A from expression E in context Γ, while the checking judgment $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ checks whether E has given type A. As usual, type annotations

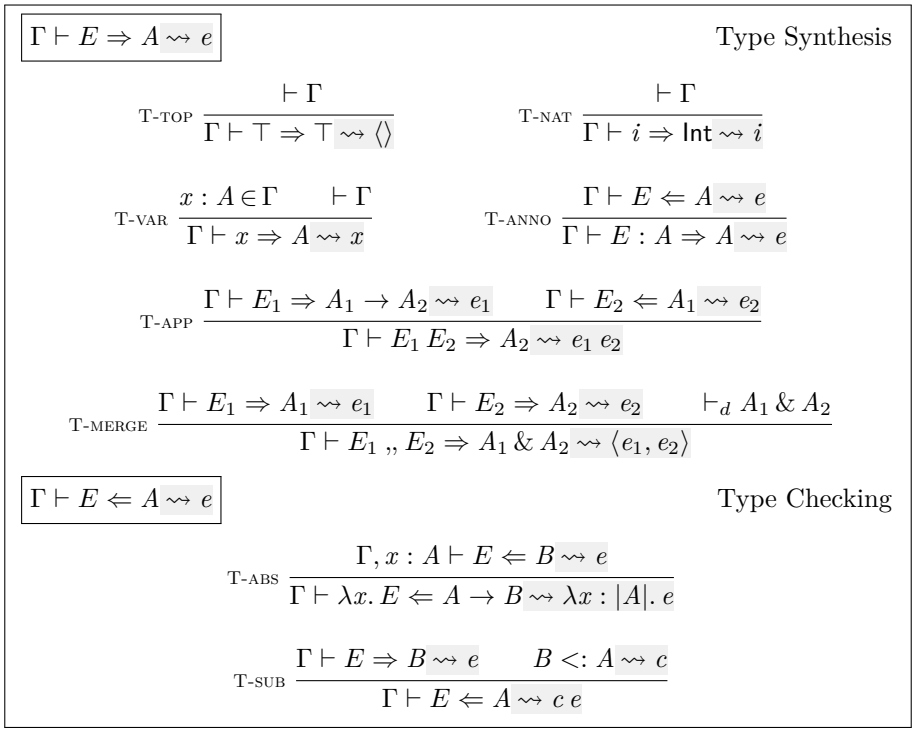


Figure 5.2: Bidirectional Typing

switch from synthesis to checking mode (rule T-ANNO). We reserve further discussion over the parts highlighted in gray for Section 5.4.2.

The rules are identical to those of NeColus with one notable difference. Rule T-MERGE not only requires the types A_1 and A_2 of the two subterms to be disjoint, $A_1 * A_2$, but also requires them to be *internally* disjoint, $\vdash_d A_1$ and $\vdash_d A_2$, all summarised in the internal disjointness premise $\vdash_d A_1 \,\&\, A_2$ of the rule. The two disjointness conditions are discussed in Section 5.4.4.

The judgment $\vdash \Gamma$ used in the premise of rules T-TOP, T-NAT and T-VAR expresses that the typing context Γ is well-formed, i.e. it holds no duplicate variables. Its definition can be found in Appendix C.3.

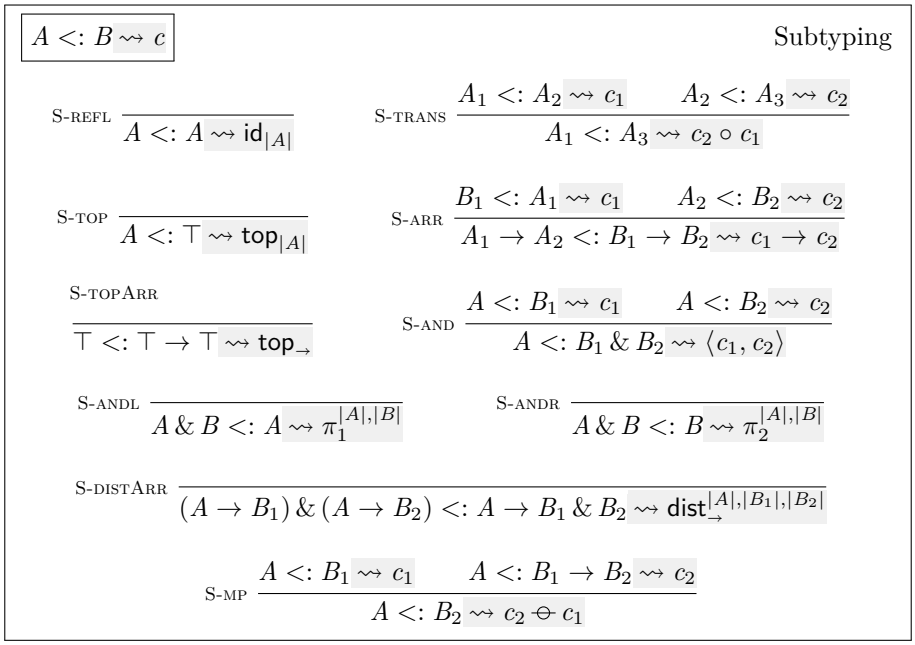


Figure 5.3: Declarative Specification of Subtyping

5.4.1 Declarative Subtyping

Figure 5.3 declaratively specifies the subtyping relation of λ_i^{MP}. The judgment $A <: B \rightsquigarrow c$ states that A is a subtype of B and produces coercion c. Intuitively, c captures the way to convert values of type A into values of type B.

BCD-Style Subtyping All but the last subtyping rule are the same as those of NeColus and originate from the BCD type system [Barendregt et al. 1983]. Rules S-REFL, S-TRANS and S-TOP ensure that subtyping is a partial order with $\top$ as greatest element. Rule S-ARR is the standard subtyping rule for function types, while rules S-ANDL, S-ANDR and S-AND are standard for intersection types. Rules S-DISTARR and S-TOPARR are standard BCD subtyping rules; the former enables *nested composition* [Bi et al. 2018], while the latter provides a more unified treatment of top-like types in the meta-theory of the language: without that rule, we cannot have $A \to B <: \top \to \top$, if A is not top-like itself.

Types	τ	::=	$\mathsf{Int} \mid \mathsf{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$
Terms	e	::=	$x \mid \langle\rangle \mid i \mid \lambda x : \tau.\, e \mid e_1\, e_2 \mid \langle e_1, e_2 \rangle \mid c\, e$
Coercions	c	::=	$\mathsf{id}_\tau \mid c_1 \circ c_2 \mid \mathsf{top}_\tau \mid \mathsf{top}_\to \mid c_1 \to c_2 \mid \mathsf{dist}_\to^{\tau_1,\tau_2,\tau_3} \mid \pi_1^{\tau_1,\tau_2} \mid \pi_2^{\tau_1,\tau_2} \mid \langle c_1, c_2 \rangle \mid c_1 \multimap c_2$
Typing contexts	Δ	::=	$\bullet \mid \Delta, x : \tau$
Values	v	::=	$\langle\rangle \mid i \mid \lambda x : \tau.\, e \mid \langle v_1, v_2 \rangle \mid \mathsf{top}_\to\, v \mid (c_1 \to c_2)\, v \mid (\mathsf{dist}_\to^{\tau_1,\tau_2,\tau_3})\, v$

Figure 5.4: λ_c^{MP} Syntax

Modus Ponens Our calculus extends the subtyping relation of NeColus with one new rule, S-MP, which corresponds to the modus ponens inference rule of propositional logic. It states that if A is a subtype of both $B_1 \to B_2$ and of B_1, then it is also a subtype of B_2. While this extension of subtyping is conceptually small, it has far-reaching consequences for the meta-theory of the language.

5.4.2 Elaboration Semantics

Following NeColus we assign a semantics to λ_i^{MP} by means of an elaboration. The target of the elaboration is λ_c^{MP}, which is the simply typed lambda-calculus extended with products and coercions. Pairs are the elaboration target of merges (rule T-MERGE), while coercions explicitly witness the implicit use of subtyping in λ_i^{MP} (rule T-SUB).

The λ_c^{MP} Calculus Figure 5.4 shows the syntax of λ_c^{MP}. This syntax is nearly identical to that of NeColus' target language. There are only two differences, both situated in the coercions c. These coercions express the conversion of a term from one type to another. The reason why we use a separate syntactic sort of coercions for this—rather than encoding the conversions as regular function terms—is that it facilitates the coherence proof. Each coercion form corresponds to a particular subtyping rule (Figure 5.3). The first and main difference to NeColus' target calculus is that, because we have a new modus ponens subtyping rule in λ_i^{MP}, there is also a new corresponding coercion form $c_1 \multimap c_2$ in λ_c^{MP} to witness it. The second and minor difference is that we have annotated several existing coercion forms with type information in order to facilitate type checking.

Figure 5.5 lists the new inference rules in the type system and small-step operational semantics of λ_c^{MP} to support the new coercion form. The full definitions of these relations can be found in Appendix C.1. The coercion typing

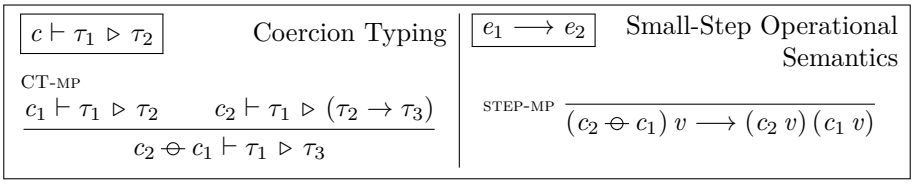


Figure 5.5: λ_c^{MP} Coercion Typing and Operational Semantics (new rules)

rule CT-MP essentially replicates the subtyping rule S-MP of Figure 5.3, but now with λ_c^{MP} types. The new small-step rule, STEP-MP, splits up an application of a modus ponens coercion $(c_2 \multimap c_1)\, v$ into separate coercion applications $c_2\, v$ and $c_1\, v$ where the former should yield a function that is applied to the latter.

How the features of λ_i^{MP} are mapped to λ_c^{MP} is given in the grey parts of Figures 5.3 and 5.2. In those figures the meta-function $|\cdot|$ transforms λ_i^{MP} types to λ_c^{MP} types, and extends naturally to typing contexts. Its definition is in Appendix C.3.

5.4.3 Target and Elaboration Metatheory

We have formally investigated the metatheory of our calculus, λ_i^{MP}, and of its target language, λ_c^{MP}, and mechanised it in Coq.

Target Language Firstly, we have proven that the key metatheoretical properties of λ_c^{MP} are not invalidated by the new coercion form. Indeed, λ_c^{MP} is type safe:

Theorem 5.1 (Preservation). *If* $\bullet \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\bullet \vdash e' : \tau$*.*

Theorem 5.2 (Progress). *If* $\bullet \vdash e : \tau$*, then either* e *is a value or there is an* e' *such that* $e \longrightarrow e'$*.*

Moreover, every well-typed λ_c^{MP} term is normalising. Here, $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$.

Theorem 5.3 (Normalisation). *If* $\bullet \vdash e : \tau$ *then there exists a value* v *such that* $e \longrightarrow^* v$*.*

Elaboration Also, the elaboration of λ_i^{MP} to λ_c^{MP} preserves well-typing.

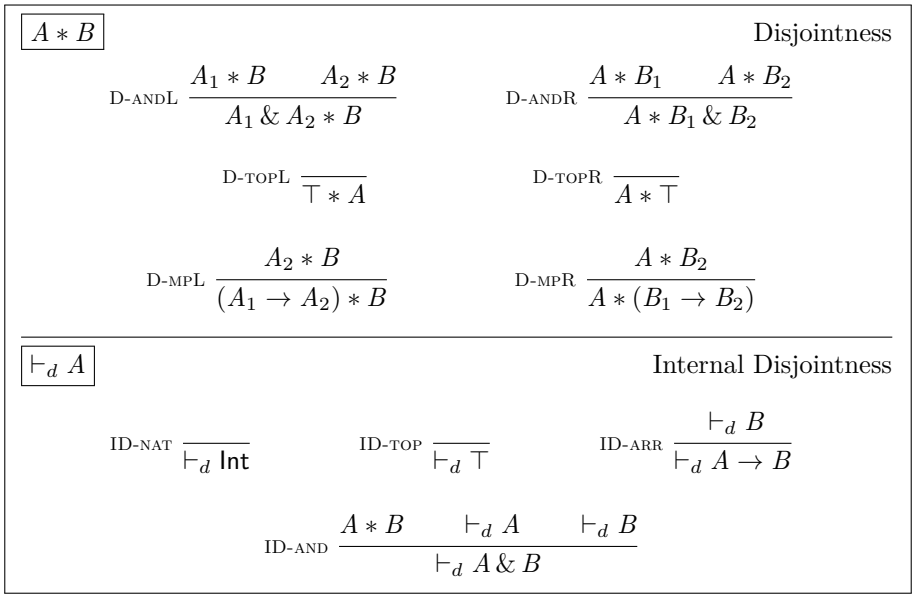


Figure 5.6: Disjointness conditions

Theorem 5.4 (Coercions Preserve Types). *If* $A <: B \rightsquigarrow c$, *then* $c \vdash |A| \triangleright |B|$.

Theorem 5.5 (Elaboration Preserves Types).

- *If* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$, *then* $|\Gamma| \vdash e : |A|$.
- *If* $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$, *then* $|\Gamma| \vdash e : |A|$.

5.4.4 Disjointness Conditions

Figure 5.6 shows the rules for disjointness (top) and internal disjointness (bottom).

Disjointness The purpose of the disjointness condition is to avoid ambiguous expressions like $1,,2$. Judgment $A * B$ conservatively captures the notion that the common supertypes of A and B are all top-like. The first four rules of Figure 5.6, for the top type and intersection types, are the same as those of NeColus. We turn our focus to the new rules D-mpL and D-mpR, added to accommodate the new modus ponens subtyping rule. While they can be combined to recover

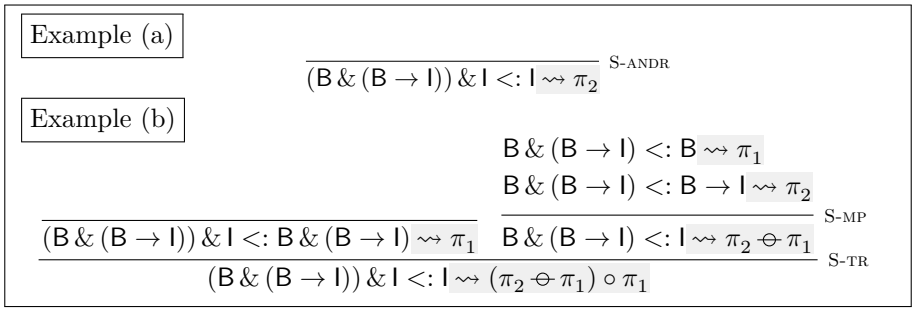


Figure 5.7: Example subtyping derivations

NeColus' D-ARR rule[7],

$$\text{D-ARR}\ \frac{B_1 * B_2}{(A_1 \to B_1) * (A_2 \to B_2)}$$

they cannot recover the NeColus axioms $\mathsf{Int} * (A \to B)$ and $(A \to B) * \mathsf{Int}$. On the contrary, λ_i^{MP} does not consider the types Int and Bool → Int to be disjoint. Although merging any expressions of those two types does not lead to ambiguity (there is only one way to generate an Int value), it does not prevent ambiguity either. Consider the example $((2 \,,, (\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Int}) \,,, \mathsf{true}) : \mathsf{Int}$. The elaborated form of the merge expression on the left of the outer type annotation is the tuple $\langle\langle 2, \lambda x : \mathsf{Bool}.\, 1\rangle, \mathsf{true}\rangle$. The outer type annotation will trigger the subtyping mechanism, first via rule T-ANNO and then through rule T-SUB, in order to produce a value of type Int. This example is ambiguous, because it can produce—through subtyping—two different values of this type: 2 by projecting on the left component of the inner tuple, and 1 by projecting on the right component of the inner tuple and using modus ponens to apply $(\lambda x : \mathsf{Bool}.\, 1)$ to true. Thus, for the sake of conciseness and compositionality, the disjointness definition is conservative. For similar reasons, expressions $(\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Int} \,,, (2 \,,, \mathsf{true})$ and $(\mathsf{true} \,,, (\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Int}) \,,, 2$ are also ambiguous. In all three cases, the component types of a merge will not be disjoint. For the latter case, Figure 5.7 shows the two subtyping derivations that generate the distinct coercions. In those derivations, types Bool and Int are abbreviated as B and I, respectively, and type annotations are omitted in the projection coercions.

The two new rules D-MPL and D-MPR bring the notion of disjointness between two types closer to the notion of no-overlaps between two (or more) instances.

[7]This rule states that two arrow types are disjoint if their codomains are disjoint.

Types B and $A \to B$ are not disjoint, similarly to how two Haskell instances `C Int` and `D Int => C Int` overlap.

Harmless overlaps and (non-)ambiguity Note that not all forms of overlap introduce semantic ambiguity. In particular, we identify two such *harmless* cases. Firstly, in the presence of an overlap, it can still be the case that there is only one way to resolve a value of a given (overlapping) type. For example, in the term $2 \,,,\, (\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Int}$ of type $\mathsf{Int} \,\&\, (\mathsf{Bool} \to \mathsf{Int})$, the types Int and $\mathsf{Bool} \to \mathsf{Int}$ overlap, but there is only one applicable way to resolve a value of type Int, because there is no Bool value to feed to the function $(\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Int}$. Secondly, expressions of an overlapping type are harmless when all possible resolution paths for a wanted type result in the same inferred value. For example, the term $(1 \,,,\, 1)$, while containing an overlap, is unambiguous.

Because of the disjointness condition imposed over merges, the source calculus conservatively rejects expressions that fall under either of the two harmless forms of overlap, when they are explicitly written by the programmer. However, they are still possible implicitly, through a type annotation, which invokes subtyping. For example, we can encode the rejected term $(1 \,,,\, 1)$ as $1 : \mathsf{Int} \,\&\, \mathsf{Int}$, which is accepted and semantically same. Its full typing derivation is in Appendix C.4.

Internal Disjointness The internal disjointness predicate, $\vdash_d A$, requires that the two components of any intersection type that appear in A are disjoint. Intuitively, $\vdash_d A$ ensures that if $A <: B$, then B can only contain harmless overlaps.

The internal disointness condition is imposed on the subterms of a merge in rule T-MERGE. Again this is a conservative measure to avoid ambiguity, this time of a more convoluted nature that involves nested merges interleaved with subtyping. Here is a minimal ambiguous example to illustrate the problem:

$$((\mathsf{true} \,,,\, \mathsf{b2n}) : \mathsf{Int} \,\&\, (\mathsf{Bool} \to \mathsf{Int}) \,,,\, \mathsf{false}) : \mathsf{Int}$$

This example features $\mathsf{b2n}$, which we assume to be a built-in function of type $\mathsf{Bool} \to \mathsf{Int}$ that maps true to 1 and false to 0. The inner merge has type $\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{Int})$, but is converted to type $\mathsf{Int} \,\&\, (\mathsf{Bool} \to \mathsf{Int})$ by means of the type annotation, which triggers the subtyping mechanism. The latter produces a coercion that is then applied on (the elaborated form of) the subterm $\mathsf{true} \,,,\, \mathsf{b2n}$. The corresponding value is equivalent to $1 \,,,\, \mathsf{b2n}$, which could not be written explicitly because Int and $\mathsf{Bool} \to \mathsf{Int}$ are not considered disjoint. Yet, because this value is not explicitly written by the programmer, but implicitly produced through subtyping, we know that the overlap is harmless. Indeed, while the

expression 1 ,, b2n potentially produces two different values of type Int, one of those requires a Bool, which is not available at this point.

However, the outer merge with false provides the missing Bool value, making the outer Int type annotation ambiguously yield both 0 and 1 as possible results. The disjointness check rejects none of the two merges in the example. Here, the internal disjointness constraint comes in. It catches the ambiguity problem at the outer merge where it identifies 1 ,, b2n as unfit to merge with.

Internal disjointness inherits the conservative behavior of the (binary) disjointness condition, as witnessed in rule ID-ARR. This rule regards the case of a function type $A \to B$ that potentially is a supertype of some internally disjoint type, say A' (formally, $A' <: A \to B$). If A' is not a subtype of A (thus, $A' <: A$ does not hold), then rule S-MP does not apply. In fact, it becomes impossible to implicitly produce B from A' and, therefore, unnecessary to demand internal disjointness of B.

The indifference of rule ID-ARR around domains of arrow types is justified by the intuition that if a supertype of an internally disjoint type contains overlaps, then those are harmless. A function of type $A \to B$ is unambiguous if it outputs unambiguous values of type B, given unambiguous input values of type A. This implies the requirement that A is also internally disjoint. However, at this point, only values produced implicitly from A' are considered, and these can only contain harmless ambiguity. Indeed, explicit values can be added only through the merge operator, thus falling under rule ID-AND. Thus, unambiguity is ensured for A.

5.5 Algorithmic Subtyping

In this section, we present an algorithm that decides the subtyping relation of Figure 5.3.

In the declarative specification of the subtyping relation (Figure 5.3), rules S-TRANS and S-MP hinder the decidability of subtyping: they both require guessing an appropriate intermediate type (A_2 in the former, and B_1 in the latter). We address this by employing a technique originating from proof search known as *focusing* [Andreoli 1992]. The general strategy for checking a constraint of the form $A <: B$ is to structurally analyze (focus on) B, and when it is stripped down to a base type do the same for A. This task is performed by two mutually-recursive judgments, the first focusing on the *right* type B, and the second focusing on the *left* type A. This technique has already been employed by Bi et al. [2018]—and, before that, by Pierce [1989]—though with a single

all-in-one judgment. Our algorithm separates the two phases for clarity (and ease of reasoning) and also deals with the additional complexity introduced by rule S-MP. Unfortunately, the support for modus ponens greatly complicates the algorithm's proofs of transitivity and termination. For the former, we had to come up with an approach external to Coq, involving the AProVE termination checker [Giesl et al. 2017], to establish well-foundedness. In the remainder of the section we describe the subtyping algorithm in detail (Subection 5.5.1), as well as its most important meta-theoretical properties (Subection 5.5.2). We follow the earlier convention to highlight the parts of the rules that concern coercion generation.

5.5.1 The Subtyping Algorithm

In Figure 5.8, we present the algorithmic subtyping judgment $A \prec: B \rightsquigarrow c$; the algorithmic counterpart of the declarative judgment $A <: B \rightsquigarrow c$ of Subsection 5.4.1. The main judgment is defined in terms of two auxiliary, mutually-recursive judgments $\mathcal{L} \vdash_{\mathrm{R}} A \prec: B \rightsquigarrow c$ and $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$, each focusing on the right or the left component of a type inequality, respectively. We explain the former in Subsection 5.5.1 and the latter in Subsection 5.5.1.

Preliminaries Meta-variables $\mathcal{L}$ and $\mathcal{M}$ stand for queues of types. The algorithm uses them to capture the left-hand sides of arrow types. Operation $\mathcal{L} \twoheadrightarrow B$ converts a queue $\mathcal{L}$ back into a type, given the right-hand side type B. It is inductively defined as follows:

$$[] \twoheadrightarrow B = B \qquad\qquad (A, \mathcal{L}) \twoheadrightarrow B = A \to (\mathcal{L} \twoheadrightarrow B)$$

Meta-variable $\mathcal{X}$ denotes *algorithmic contexts*. An algorithmic context is merely the Cayley representation of a coercion: a function from coercions to coercions. Given a coercion c, we can convert an algorithmic context $\mathcal{X}$ into a coercion by function application, denoted as $\mathcal{X}[c]$. The role of algorithmic contexts is explained in Subsection 5.5.1 where we discuss the left-focusing rules[8].

[8]In both our mechanization and implementation of the algorithm we use a defunctionalized [Reynolds 1972] representation of algorithmic contexts, which we then interpret as actual functions between coercions.

$\boxed{A \prec: B \rightsquigarrow c}$ Algorithmic Subtyping

$$\text{A-MAIN}\ \frac{[] \vdash_{\mathrm{R}} A \prec: B \rightsquigarrow c}{A \prec: B \rightsquigarrow c}$$

$\boxed{\mathcal{L} \vdash_{\mathrm{R}} A \prec: B \rightsquigarrow c}$ Right Focusing

$$\text{AR-AND}\ \frac{\mathcal{L} \vdash_{\mathrm{R}} A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash_{\mathrm{R}} A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash_{\mathrm{R}} A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&}^{B_1, B_2} \circ \langle c_1, c_2 \rangle}$$

$$\text{AR-ARR}\ \frac{\mathcal{L}, B_1 \vdash_{\mathrm{R}} A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash_{\mathrm{R}} A \prec: B_1 \to B_2 \rightsquigarrow c} \qquad \text{AR-TOP}\ \frac{}{\mathcal{L} \vdash_{\mathrm{R}} A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}_{|A|}}$$

$$\text{AR-INT}\ \frac{\mathcal{L}; []; A; \ (\lambda c.c) \vdash_{\mathrm{L}} A \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}}{\mathcal{L} \vdash_{\mathrm{R}} A \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}[\mathsf{id}_{|\mathsf{Int}|}]}$$

$\boxed{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'}$ Left Focusing

$$\text{AL-INT}\ \frac{}{[]; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} \mathsf{Int} \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}}$$

$$\text{AL-AND1}\ \frac{\mathcal{L}; \mathcal{M}; A_0; \ (\lambda c.\mathcal{X}[c \circ \pi_1^{|A_1|,|A_2|}]) \vdash_{\mathrm{L}} A_1 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \,\&\, A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}$$

$$\text{AL-AND2}\ \frac{\mathcal{L}; \mathcal{M}; A_0; \ (\lambda c.\mathcal{X}[c \circ \pi_2^{|A_1|,|A_2|}]) \vdash_{\mathrm{L}} A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \,\&\, A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}$$

$$\text{AL-ARR}\ \frac{[] \vdash_{\mathrm{R}} B_1 \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L}; \mathcal{M}, B_1; A_0; \ (\lambda c.\mathcal{X}[c_1 \to c]) \vdash_{\mathrm{L}} A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}{B_1, \mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \to A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'}$$

$$\text{AL-MP}\ \frac{[] \vdash_{\mathrm{R}} A_0 \prec: (\mathcal{M} \twoheadrightarrow A_1) \rightsquigarrow c_2 \qquad \mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X}' \vdash_{\mathrm{L}} A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}'' \qquad \mathcal{X}' = (\lambda c.[\![\mathcal{M}]\!]_{\&}^{(c \circ (\pi_1^{|A_1 \to A_2|,|A_1|} \mathbin{\ominus} \pi_2^{|A_1 \to A_2|,|A_1|})), A_1 \to A_2, A_1} \circ \langle \mathcal{X}[\mathsf{id}_{|A_1 \to A_2|}], c_2 \rangle)}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \to A_2 \prec: \mathsf{Int} \rightsquigarrow \mathcal{X}''}$$

Figure 5.8: Algorithmic Subtyping

Right Focusing

As mentioned earlier, the first judgment ($\mathcal{L} \vdash_{\mathrm{R}} A \prec: B \rightsquigarrow c$) structurally analyzes B, until it is stripped down to a base type (Int or $\top$). The accumulating parameter $\mathcal{L}$ captures the domains of arrows encountered in B, such that the following holds:

Lemma 5.1 (Soundness of Right Focusing)**.**
If $\mathcal{L} \vdash_R A \prec: B \rightsquigarrow c$*, then* $A <: (\mathcal{L} \twoheadrightarrow B) \rightsquigarrow c$.

This judgment is mostly borrowed from NeColus; we explain it only briefly here. Ignoring coercion generation for now, the judgment works as follows: Rule AR-AND decomposes an intersection type, similar to its declarative counterpart (rule S-AND). Rule AR-ARR deals with arrow types: the left sides of arrow types are inserted in the rear of type queue $\mathcal{L}$ and removed again later, left-to-right, as arrows are encountered in A (see rule AL-ARR below). There are two base cases: either $B = \top$ or $B = \mathsf{Int}$. Rule AR-TOP handles the first case, where subtyping directly produces a coercion with the use of the meta-function $[\![\mathcal{L}]\!]_\top$, discussed in the next paragraph. If $B = \mathsf{Int}$, then we perform structural analysis on A, using judgment $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$, which we elaborate on in Subsection 5.5.1.

Coercion Generation The elaboration side is a bit more involved. Coercion generation can be understood via Lemma 5.1 and works as follows. Rule AR-AND deals with intersection types. By induction we have that

$$\left.\begin{array}{l} A <: (\mathcal{L} \twoheadrightarrow B_1) \rightsquigarrow c_1 \\ A <: (\mathcal{L} \twoheadrightarrow B_2) \rightsquigarrow c_2 \end{array}\right\} \xRightarrow{\text{S-AND}} A <: (\mathcal{L} \twoheadrightarrow B_1) \,\&\, (\mathcal{L} \twoheadrightarrow B_2) \rightsquigarrow \langle c_1, c_2 \rangle$$

We finish the transition to $\mathcal{L} \twoheadrightarrow (B_1 \,\&\, B_2)$ using meta-function $[\![\mathcal{L}]\!]_{\&}^{B_1, B_2}$, whose meaning is given by the following lemma[9]:

Lemma 5.2.
$(\mathcal{L} \twoheadrightarrow B_1) \,\&\, (\mathcal{L} \twoheadrightarrow B_2) <: \mathcal{L} \twoheadrightarrow (B_1 \,\&\, B_2) \rightsquigarrow [\![\mathcal{L}]\!]_{\&}^{B_1, B_2}$.

Rule AR-TOP deals with the same problem that rule AR-AND does: according to Lemma 5.1, the coercion needs to witness $A <: (\mathcal{L} \twoheadrightarrow \top)$, and not simply $A <: \top$. As $(\mathcal{L} \twoheadrightarrow \top)$ is *top-like* [Alpuim et al. 2017] we can always cast $\top$ to it, which is what $[\![\mathcal{L}]\!]_\top$ witnesses[9]:

Lemma 5.3. $\top <: (\mathcal{L} \twoheadrightarrow \top) \rightsquigarrow [\![\mathcal{L}]\!]_\top$*, for any* $\mathcal{L}$.

[9]Its definition is straightforward and thus omitted; it can be found in Appendix C.3.

$$
\begin{array}{llcll}
\textsc{ar-int} & \Longrightarrow & \mathcal{L}_0; \mathcal{M}_0; A_0; \mathcal{X}_0 \vdash_L A_0 \prec: \mathsf{Int} & \rightsquigarrow & \mathcal{X}_n \quad (\mathcal{M}_0 = [], \mathcal{X}_0 = \square) \\
 & & \downarrow & & \uparrow \\
 & & \mathcal{L}_1; \mathcal{M}_1; A_0; \mathcal{X}_1 \vdash_L A_1 \prec: \mathsf{Int} & \rightsquigarrow & \mathcal{X}_n \\
 & & \downarrow & & \uparrow \\
 & & \cdots & & \cdots \\
 & & \downarrow & & \uparrow \\
\textsc{al-int} & \Longrightarrow & \mathcal{L}_n; \mathcal{M}_n; A_0; \mathcal{X}_n \vdash_L A_n \prec: \mathsf{Int} & \rightsquigarrow & \mathcal{X}_n \quad (A_n = \mathsf{Int}, \mathcal{L}_n = [])
\end{array}
$$

Figure 5.9: Visual Representation of the Execution of $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$

Rule AR-ARR is straightforward. Finally, rule AR-INT handles the base case where B is Int. In this case, we resort to the auxiliary judgment $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$, which we discuss next.

Left Focusing

The intuition behind $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathrm{L}} A \prec: B \rightsquigarrow \mathcal{X}'$ is that the (input) algorithmic context $\mathcal{X}$ *witnesses* the transition from A_0 to $(\mathcal{M} \twoheadrightarrow A)$, the component of A_0 that we are currently focusing on. The (output) algorithmic context $\mathcal{X}'$ is meant to capture the complete transition from A_0 to $(\mathcal{M} \to \to (\mathcal{L} \twoheadrightarrow B))$. Essentially, starting with rule AR-INT as the entry point, the judgment continuously decomposes A (via rules AL-AND1, AL-AND2, AL-ARR, and AL-MP) until it equals B (rule AL-INT); then the accumulated context $\mathcal{X}$ witnesses the complete transition from A_0 to $(\mathcal{M} \twoheadrightarrow (\mathcal{L} \twoheadrightarrow B))$ and the job is done. Figure 5.9 visualizes this process.

Ignoring coercion generation, the judgment works as follows: Rule AL-INT constitutes the base case where subtyping holds trivially. Rules AL-AND1 and AL-AND2 non-deterministically focus on the left or the right component of an intersection, similarly to their declarative counterparts (rules S-ANDL and S-ANDR, respectively). Rule AL-ARR is the algorithmic version of rule S-ARR: we dequeue B_1 from $\mathcal{L}$ in a first-in-first-out manner, in order to match the order in which arrows are encountered in A. After we prove subtyping for the domains of the functions, we record that we are going under a binder of type B_1 in the queue $\mathcal{M}$ and proceed recursively with the function images. Queue $\mathcal{M}$ thus captures all the arrow domains in B that have been removed in lockstep with arrow domains in A using rule S-ARR.[10] This context becomes useful when we

[10] In Figure 5.9, it is easy to see that $\mathcal{M}_i +\!+ \mathcal{L}_i = \mathcal{L}_0$, where $+\!+$ denotes list concatenation. At the end, we have $\mathcal{M}_n = \mathcal{L}_0$.

need to use modus ponens under a binder, e.g. to prove

$$\mathsf{Int} \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Int} \to \mathsf{String}$$

(or even $\top \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Int} \to \mathsf{String}$). Lastly, rule AL-MP is the algorithmic version of the S-MP rule. The first premise reveals the synergistic relationship between this rule and rule AL-ARR: when we need to synthesize an argument for modus ponens, we restart from the original type A_0—thus capturing the complete implicit context of $A_1 \to A_2$—and use the contextualized version of A_1, $(\mathcal{M} \twoheadrightarrow A_1)$. For the above example, this amounts to proving

$$\mathsf{Int} \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Int} \to \mathsf{Bool}$$

The second premise is straightforward.

Note that rules AL-ARR and AL-MP overlap and, to ensure completeness with respect to the declarative specification of subtyping, backtracking is required. However, it is never the case that both rules can apply for resolving a subtyping judgment. That is, backtracking will fail in at least one of the two choices.

Coercion Generation The elaboration side, again, requires a bit more explanation. The complication arises in rule AL-MP, where the first premise restarts from the original type A_0. This is essential for recovering the unfocused context needed for modus ponens, but it goes against the inductive, top-down approach to coercion generation that we follow in the right-focusing rules. Coercions produced by the recursive calls can no longer be directly combined. This is why the left focusing rules use algorithmic contexts instead of coercions. The essence of the algorithmic contexts is captured in the following invariant, which is preserved by each step of the left focusing algorithm:

Invariant 1. $\forall c\ A', A_i <: A' \rightsquigarrow c \implies A_0 <: (\mathcal{M}_i \twoheadrightarrow A') \rightsquigarrow \mathcal{X}_i[c]$

Metavariables A_i, A_0, $\mathcal{M}_i$, and $\mathcal{X}_i$ refer to the corresponding parameters at the i-th step in Figure 5.9. Informally, $\mathcal{X}_i$ captures the transition from A_0 to A_i: $\mathcal{X}_i[c]$ witnesses the complete transition from A_0 to *any* type A', if we have a way to perform the remaining transition from A_i to A'. Hence, every rule of the judgment strips a layer off A and extends the "input" algorithmic context accordingly, so that Invariant 1 is preserved.

Rule AL-MP is by far the most interesting rule. It commits to using the S-MP rule: the left premise produces a coercion c_2 that provides the argument (of type $\mathcal{M} \twoheadrightarrow A_1$), while the appropriate instantiation of the current context $\mathcal{X}$ gives us the function (of type $\mathcal{M} \twoheadrightarrow (A_1 \to A_2)$):

$$A_0 <: \mathcal{M} \twoheadrightarrow A_1 \rightsquigarrow c_2$$
$$A_0 <: \mathcal{M} \twoheadrightarrow (A_1 \to A_2) \rightsquigarrow \mathcal{X}[\mathsf{id}_{|A_1 \to A_2|}]$$

Yet, the above coercions cannot be directly combined, due to the type queue $\mathcal{M}$. We remedy this using the meta-function $[\![\mathcal{M}]\!]_{\&}^{c,B,B'}$, the meaning of which is given by the following lemma[11]:

Lemma 5.4.
If $B \,\&\, B' <: A \rightsquigarrow c$*, then* $(\mathcal{M} \twoheadrightarrow B) \,\&\, (\mathcal{M} \twoheadrightarrow B') <: \mathcal{M} \twoheadrightarrow A \rightsquigarrow [\![\mathcal{M}]\!]_{\&}^{c,B,B'}$*, for any* $\mathcal{M}$*.*

The idea is that if $A_0 <: \mathcal{M} \twoheadrightarrow A_1 \rightsquigarrow c_2$ and $A_0 <: \mathcal{M} \twoheadrightarrow (A_1 \to A_2) \rightsquigarrow c_1$, then

$$A_0 <: \mathcal{M} \twoheadrightarrow A_2 \rightsquigarrow (\lambda c.[\![\mathcal{M}]\!]_{\&}^{(c \circ d),A_1 \to A_2,A_1} \circ \langle c_1, c_2 \rangle)\ [\mathsf{id}_{|A_2|}]$$

where $d = \pi_1^{|A_1 \to A_2|,|A_1|} \mathbin{\&} \pi_2^{|A_1 \to A_2|,|A_1|}$. Returning to rule AL-MP, the second premise uses the above algorithmic context (named $\mathcal{X}'$ for convenience) to further focus on A_2.

5.5.2 Algorithm Metatheory

Subsumption of NeColus' Algorithmic Subtyping First, we have proven that the subtyping algorithm of Figure 5.8 subsumes the subtyping algorithm of NeColus [Fig. 12, Bi et al. 2018].

Theorem 5.6 (Subsumption of NeColus' Algorithm)**.**
If $[] \vdash A \ll B \rightsquigarrow c$*, then* $A \prec: B \rightsquigarrow c$*.*

Here, $\mathcal{L} \vdash A \ll B \rightsquigarrow c$ denotes the subtyping algorithm of NeColus.

Transivity and Modus Ponens A major difference between algorithmic and declarative subtyping is that the latter has an explicit transitivity rule while the former does not. Hence, transitivity of algorithmic subtyping requires a separate proof. Unfortunately, this proof is considerably more involved than in NeColus. Instead of one inductive core lemma, it consists of 7 mutually recursive lemmas. The well-foundedness of the induction used in the proofs of those 7 lemmas is no longer a straightforward size-based induction that can be easily understood by Coq. To overcome this problem we have used a new approach, external to Coq, to establish well-foundedness. We have encoded the proof—which is essentially an algorithm that transforms derivation trees—as a term rewriting system. Then we have used the AProVE tool [Giesl et al. 2017] to verify its termination automatically. AProVE reports success after 40 steps of transformation, decomposition and basic termination arguments.

[11] Its definition is straightforward and thus omitted; it can be found in Appendix C.3.

Theorem 5.7 (Transitivity of Algorithmic Subtyping)**.**
If $A_1 \prec: A_2 \rightsquigarrow c_1$ *and* $A_2 \prec: A_3 \rightsquigarrow c_2$*, then* $A_1 \prec: A_3 \rightsquigarrow c_3$ *for some* c_3*.*

The modus ponens property for algorithmic subtyping is closely tied to the transitivity proof and is another consequence of the 7 mutually recursive lemmas.

Corollary 5.1 (Modus ponens of Algorithmic Subtyping)**.**
If $A \prec: B_1 \to B_2 \rightsquigarrow c_1$ *and* $A \prec: B_1 \rightsquigarrow c_2$*, then* $A \prec: B_2 \rightsquigarrow c_3$ *for some* c_3*.*

Soundness and Completeness We have proven that the subtyping algorithm (Figure 5.8) is sound and complete with respect to its declarative specification (Figure 5.3). The soundness of the overall algorithm is a direct corollary of the soundness of right focusing (Lemma 5.1):

Theorem 5.8 (Soundness)**.** *If* $A \prec: B \rightsquigarrow c$*, then* $A <: B \rightsquigarrow c$*.*

The completeness proof largely follows the same strategy as for NeColus. The main challenge was the new transitivity proof discussed above.

Theorem 5.9 (Completeness)**.**
If $A <: B \rightsquigarrow c$*, then there is a coercion* c' *such that* $A \prec: B \rightsquigarrow c'$*.*

Termination The final key property of the algorithm is termination. Actually, the algorithm presented in Figure 5.8 is not terminating. Consider for example the trace of $\mathsf{Int} \to \mathsf{Int} \prec: \mathsf{Int}$:

$$
\begin{array}{l}
([] \vdash_{\mathrm{R}} \mathsf{Int} \to \mathsf{Int} \prec: \mathsf{Int}) \xrightarrow{\text{AR-INT}} ([];[];\mathsf{Int} \to \mathsf{Int} \vdash_{\mathrm{L}} \mathsf{Int} \to \mathsf{Int} \prec: \mathsf{Int}) \xrightarrow{\text{AL-MP}} \\
([] \vdash_{\mathrm{R}} \mathsf{Int} \to \mathsf{Int} \prec: \mathsf{Int}) \xrightarrow{\text{AR-INT}} ([];[];\mathsf{Int} \to \mathsf{Int} \vdash_{\mathrm{L}} \mathsf{Int} \to \mathsf{Int} \prec: \mathsf{Int}) \xrightarrow{\text{AL-MP}} \\
\ldots
\end{array}
$$

Such a subtyping judgment can result from an erroneous type annotation by the user. Nonetheless, the language does not statically detect annotations that may cause non-termination in the subtyping algorithm. Another example, perhaps more plausible as a programming error, is an annotation that leads to the judgment $(\mathsf{Int} \to \mathsf{Bool}) \,\&\, (\mathsf{Bool} \to \mathsf{Int}) \prec: \mathsf{Int}$. One simple ingredient needs to be added to the algorithm to make it terminating: a loop detection on the $[] \vdash_{\mathrm{R}} A \prec: B$ call in rule AL-MP. This loop detection should reject any calls that repeat one of their ancestor calls.

We can show that (1) any infinite derivation involves an infinite number of these nested calls and that (2) detecting repeated calls is effective at stopping the non-termination.

A priori there are multiple possible ways in which the algorithm can diverge. It might involve an infinite sequence of recursive calls within one of the two judgments. However, following the focusing approach, the two judgments structurally decompose their focal argument. Hence, on their own they are clearly terminating. Any divergence must thus involve an infinite number of mutually recursive calls between the two judgments. There are two "gatekeepers" of the mutual recursion, calling back from the left-focusing into the right-focusing judgment. These two are the problematic one in rule AL-MP and the one in rule AL-ARR. Yet, we can show that an infinite number of recursion steps through rule AL-ARR is impossible and thus we can eliminate it from consideration. Indeed, the *left-arrow depth* $\|\cdot\|$ defined below decreases strictly in every AL-ARR loop step, and non-strictly in every other step. Moreover, a value of 0 inhibits rule AL-ARR. Thus, after a finite number of AL-ARR steps, the rule can no longer be used.

$$\begin{array}{rclcrcl} \|\mathsf{Int}\| & = & 0 & & \|A \to B\| & = & \max(\|A\| + 1, \|B\|) \\ \|\top\| & = & 0 & & \|A \,\&\, B\| & = & \max(\|A\|, \|B\|) \end{array}$$
$$\|[] \vdash_{\mathrm{R}} A \prec: B\| = \max(\|A\|, \|B\|)$$

That leaves only the problematic mutually recursive call in rule AL-MP, which we have already demonstrated to be a source of non-termination. Fortunately, we can show that the number of distinct nested calls is finite. Hence, on any infinite derivation path, a repeated call has to occur after a finite number of steps. This makes the loop detection effective at curtailing divergence.

A more extensive account can be found in Appendix C.6.

Theorem 5.10 (Termination)**.**
The algorithm extended with loop detection terminates.

Decidability In summary, if we take the soundness, completeness and termination of the subtyping together, we can conclude that the declarative subtyping is decidable.

Corollary 5.2 (Decidable Subtyping)**.**
In λ_i^{MP}*,* $A <: B$ *is decidable for any types* A *and* B*.*

5.6 Coherence

A desired property for calculi with a merge operator is *coherence*. This ensures that no semantic ambiguity arises in programs, i.e. programs and their subparts

always have a single, unambiguous meaning. Recall that the semantics of λ_i^{MP} is given indirectly, through an elaboration to λ_c^{MP} and its operational semantics. Hence, coherence for λ_i^{MP} means that all possible elaborations of a well-typed source expression for a given type are contextually equivalent in the target language.

Two target expressions, that may be syntactically different, are *contextually equivalent* iff, when used in the context of any program, there is no difference in that program's behavior. In our purely functional and strongly normalising setting, we can reduce a program's behavior to the value it returns. Moreover, as usual, we can restrict ourselves without loss of generality to programs of type Int to facilitate the comparison of the results.

Program contexts express the above notion of using an expression in the context of a program and are defined as follows:

$$\text{Program contexts } \mathcal{C} ::= [\cdot] \mid \lambda x.\mathcal{C} \mid \mathcal{C}\, E \mid E\, \mathcal{C} \mid \mathcal{C} \,,, E \mid E \,,, \mathcal{C} \mid \mathcal{C} : A$$

Intuitively, a program context is an expression with a missing subexpression. This hole can be filled in with an appropriate term to form an actual expression. Which terms are eligible to be used in a program context is captured by the context typing judgment that consists of judgments of the form $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}$[12]. Essentially, the judgment expresses that program context $\mathcal{C}$ expects a term of type A under typing context Γ to form an expression of type B under context Γ'. Moreover, the source context $\mathcal{C}$ elaborates straightforwardly to a target context $\mathcal{D}$. Indeed, for the coherence of λ_i^{MP} we do not account for all possible target contexts, but only for those that are the image of a source context $\mathcal{C}$. This restriction is important, as Bi et al. [2018] have demonstrated that there are other target contexts that can distinguish between alternative elaborations.

We can now formally state coherence for our source calculus[13].

Theorem 5.11 (Coherence of λ_i^{MP})**.** *For any* $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D}$*, if* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$*, then there is a value* $\bullet \vdash v : \mathsf{Int}$ *such that* $\mathcal{D}\{e_1\} \longrightarrow^* v$ *and* $\mathcal{D}\{e_2\} \longrightarrow^* v$*.*

Here, $\mathcal{D}\{e\}$ denotes the term that results from filling in the hole of $\mathcal{D}$ with expression e.

[12] There are also judgments for the three other combinations of the two typing directions that appear in the judgment. The full specification can be found in Appendix C.2.

[13] There is a second statement for the type checking direction of the two hypotheses.

The remainder of this section summarizes our proof approach for coherence and discusses the issues regarding the conjectures that we use. A more extensive account can be found in Appendix C.5.

5.6.1 A Context Dependent Logical Relation

We follow the general approach for showing coherence as NeColus. This approach is based on a rather atypical heterogeneous binary logical relation, called *canonicity*, by Bi et al. [2019].

Canonicity The canonicity relation is in fact two families of relations, one for values $\mathcal{V}$ and one for expressions $\mathcal{E}$, indexed by types. Two values v_1 and v_2 of types τ_1 and τ_2 are related by canonicity, written $(v_1, v_2) \in \mathcal{V}(\tau_1, \tau_2)$, iff any parts of the same type have the same meaning. By parts we mean values v_1' and v_2' derivable from respectively v_1 and v_2 by means of subtyping coercions. If two values have non-trivial (i.e., non-top-like) parts with the same type, we say that they overlap. For instance, values $\langle 1, \mathsf{true}\rangle$ and 1 overlap since both have a part of type Int; because their overlapping part is the same, namely 1, these values are related by canonicity. In contrast, $\langle 1, \mathsf{true}\rangle$ and 2 are not related since their overlapping Int parts are different.

While the λ_i^{MP} canonicity relation follows the above principles just like that of NeColus, there are considerable differences in the actual definition because of the addition of modus ponens.

Context Dependency A major complicating factor in the design of the new logical relation is that the modus ponens subtyping rule introduces a form of context dependency. Indeed, when reasoning about overlapping parts, the NeColus approach of decomposing terms and their types, and looking at their syntactic components in isolation no longer works. Instead, in λ_i^{MP} the components have to be considered in the context of the whole value they appear in. For example, the target term $\lambda x : \mathsf{Int}.\,\mathsf{true}$ does not overlap with false, since one is an arrow type and the other a boolean. Yet, when $\lambda x : \mathsf{Int}.\,\mathsf{true}$ is part of the larger term $\langle \lambda x : \mathsf{Int}.\,\mathsf{true}, 1\rangle$, a problematic overlap arises: via modus ponens, the function applied to 1 yields the boolean value true that differs from false.

To account for that dependency on the whole value, or contextual dependency for short, we equip the canonicity relation of λ_i^{MP} with two additional indices:

the whole values and their types. Thus, we write

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1; \tau_2]\!]$$

to express that v_1 and v_2 are related values of types τ_1 and τ_2 that are components of respectively the values v_1' of type τ_1' and v_2' of type τ_2'.

We refer to Appendix C.5 for the full definition of this context-dependent canonicity relation $\mathcal{V}$ for values and its companion $\mathcal{E}$ for expressions.

Properties A key property of the canonicity relation is *coercion compatibility*. This states that any coercions of related values are related as well.

Lemma 5.5 (Coercion Compatibility)**.**
If $(v_1, v_2) \in \mathcal{V}^{(v_1:\tau_1;v_2:\tau_2)}[\![\tau_1; \tau_2]\!]$*, then* $(c_1\ v_1, c_2\ v_2) \in \mathcal{E}^{(c_1\ v_1:\tau_1';c_2\ v_2:\tau_2')}[\![\tau_1'; \tau_2']\!]$*, for any coercions* c_1*,* c_2 *and types* τ_1'*,* τ_2' *such that* $c_1 \vdash \tau_1 \triangleright \tau_1'$ *and* $c_2 \vdash \tau_2 \triangleright \tau_2'$*.*

The following lemma asserts another key property of the value relation, which establishes its connection to disjointness. This follows from the design principle that any overlap in types should concern indistinguishable values. If there is no overlap, then this is trivially satisfied.

Lemma 5.6 (Disjointly-Typed Values are Related)**.**
If $\bullet \vdash v_1 : |A_1|$ *and* $\bullet \vdash v_1 : |A_2|$*, where* $A_1 * A_2$*,*
then $(v_1, v_2) \in \mathcal{V}^{(v_1:|A_1|;v_2:|A_2|)}[\![|A_1|; |A_2|]\!]$*.*

5.6.2 Coherence Proof

With the canonicity relation in place, we can turn to the actual coherence proof. This proof is split in two parts by the canonicity relation. The first and hardest part is to show that any two elaborations of the same source term are related by canonicity; this is an adaptation of the so-called *fundamental property*. The second part is to show that related programs of type Int return the same value; it follows almost trivially from the definition of the canonicity relation.

Logical Equivalence To express the fundamental property, we need to introduce a notion of logical equivalence for open expressions on top of the canonicity relations for closed values ($\mathcal{V}$) and closed terms ($\mathcal{E}$). Open expressions are considered in some typing context Δ. A *closing substitution* γ of a typing context Δ maps variables to values of the type specified in Δ. Two open expressions are related iff applying any two related closing substitutions to e

and e', respectively, results in $\mathcal{E}$-related closed terms:

$$\Delta; \Delta' \vdash e \simeq_{\log} e' : \tau; \tau' \triangleq \forall (\gamma; \gamma') \in \mathcal{G}[\![\Delta; \Delta']\!], \quad (e[\gamma], e'[\gamma']) \in \mathcal{V}^{(e[\gamma]:\tau; e'[\gamma']:\tau')}[\![\tau; \tau']\!]$$

Two closing substitutions γ and γ' of Δ and Δ' are related iff the two typing contexts differ only in the types of the variables (otherwise, they contain the same variables and in the same order) and the values that γ and γ' contain for each variable are $\mathcal{V}$-related.

Fundamental Property Now we can state the fundamental property[14] restricted on λ_c^{MP} terms that elaborate from the same λ_i^{MP} term:

Theorem 5.12 (Fundamental property)**.**
If $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$,
then $|\Gamma|; |\Gamma| \vdash e_1 \simeq_{log} e_2 : |A|; |A|$.

Due to the definition of logical equivalence, we have proven this theorem under Conjectures 1 and 2. The proof proceeds by induction on the first hypothesis and calls coercion compatibility for the T-SUB case. The conjectures are used in the inductive cases of T-ABS and T-MERGE.

Conjecture 1 (Compatibility of abstractions)**.**
If $\Delta, x : \tau_1'; \Delta, x : \tau_2' \vdash e_1 \simeq_{log} e_2 : \tau_1; \tau_2$,
then $\Delta; \Delta \vdash \lambda x : \tau_1'.\, e_1 \simeq_{log} \lambda x : \tau_2'.\, e_2 : \tau_1' \to \tau_1; \tau_2' \to \tau_2$.

Conjecture 2 (Compatibility of pairs)**.**
If $\Delta; \Delta \vdash e_1 \simeq_{log} e_2 : \tau_1; \tau_2$ *and* $\Delta; \Delta \vdash e_1' \simeq_{log} e_2' : \tau_1'; \tau_2'$ *and* $\tau_1 * \tau_2'$ *and* $\tau_2 * \tau_1'$,
then $\Delta; \Delta \vdash \langle e_1, e_1' \rangle \simeq_{log} \langle e_2, e_2' \rangle : \tau_1 \times \tau_1'; \tau_2 \times \tau_2'$.

Partiality of the Coherence Proof Logical equivalence, as defined in this chapter, enjoys the coercion compatibility lemma, an important property for coherence. However, while compatibility of merges and abstractions hold for the $\mathcal{V}$- and $\mathcal{E}$-relations (because we can abstract over their context indexes), they do not hold for logical equivalence. This suggests that our logical equivalence does not properly express the invariant between target expressions that elaborate from the same source term. Indeed, the problem lies in capturing the context under which two target terms are logically equivalent.

Regarding compatibility of pairs, a tuple value can result either from an explicit merge or from a coercion applied on some value. In the first case,

[14] There is a second statement for the type checking direction of the two hypotheses.

the subcomponents of the tuple are disjoint with each other and thus related (by Lemma 5.6). In the second case, given that the original expression is unambiguous and since coercions do not introduce ambiguity, the coercion application is also unambiguous. However, with the current formulation of logical equivalence we cannot recover the original expression and inspect it.

For example, consider the values $\langle \mathsf{true}, \lambda x : \mathsf{Bool}.\, 1 \rangle$ and $\langle \mathsf{true}, 2 \rangle$. Two values are logically equivalent if and only if they are $\mathcal{V}$-related under themselves as contexts. Thus, while the components of the two tuples are pointwise logically equivalent, the tuples themselves are not, making a counter-example for compatibility of merges. However, there is no source expression that can elaborate to both these two terms and, therefore, their comparison for logical equivalence will never occur (note that the fundamental property is restricted only for target expressions that result from the elaboration of the same source expression). For a similar reason, compatibility of abstraction also does not hold.

Logical Equivalence Entails Contextual Equivalence Lastly, we need to show that any logically related target expressions are contextually equivalent. The proof is fairly straightforward.

Theorem 5.13 (Logical Equivalence Entails Contextual Equivalence)**.** *If* $|\Gamma|; |\Gamma| \vdash e_1 \simeq_{log} e_2 : |A|; |A|$*, then for any* $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D}$*, there is a value* $\bullet \vdash v : \mathsf{Int}$ *such that* $\mathcal{D}\{e_1\} \longrightarrow^* v$ *and* $\mathcal{D}\{e_2\} \longrightarrow^* v$*.*

Taken together, the (conditional) fundamental property and the entailment of contextual equivalence establish Theorem 5.11, the coherence of λ_i^{MP}.

5.7 Related Work

Resolution and Subtyping The main contribution of our work is to unify resolution and subtyping into a single mechanism. Resolution is a common technique for automatically constructing implicit values, such as Haskell's *type class dictionaries* [Wadler & Blott 1989] or Scala's *implicits* [Odersky 2010]. Subtyping is widely used in Object-Oriented programming and various other type systems [Barendregt et al. 1983; Chen 2003; Gay & Hole 2005; Pierce & Sangiorgi 1996]. In our work we interpret resolution as a special case of coercive subtyping [Chen 2003], where coercions are generated automatically in a type-directed fashion and then "inserted" into the program. This fits well with the common type-directed elaboration style usually employed by resolution mechanisms, which work in a similar way.

As far as we are aware, the unification of subtyping and resolution is novel. Despite the fact that languages like Scala integrate both mechanisms, only Jeffery [2019] combines them both within his DIF calculus. Unfortunately—in contrast to λ_i^{MP}—DIF's subtyping is known to be undecidable. Furthermore, Jeffery does not study the coherence of DIF. Scala implicits [Odersky 2010] have motivated other researchers to look into the foundations of the mechanism. However, all previous work on implicit calculi, including the *implicit calculus* [Oliveira et al. 2012], *Cochis* [Schrijvers et al. 2019] and the *SI calculus* [Odersky et al. 2017], do not account for subtyping. Instead, they all focus on a System F [Reynolds 1974] based language (without subtyping), and model resolution for such a language. One aspect that these implicit calculi preserve from Scala is a biased form of resolution where innermost implicits are preferred. In contrast, our disjointness approach is more reminiscent of Haskell's non-overlapping type class instances, which are unbiased.

The most common technique to algorithmically construct values by resolution, attributed to Kowalski [1974], is commonly known as *SLD resolution*, or *backwards chaining*. Another technique, mostly used as a proof search method, is *focusing* [Andreoli 1992; Liang & Miller 2009; Miller et al. 1991] and has been used to great effect in similar calculi (e.g. Cochis [Schrijvers et al. 2019] and Haskell with quantified class constraints [Bottu et al. 2017]) to create coherent resolution mechanisms. Our subtyping algorithm of Section 5.5 closely resembles the focusing technique.

Coherence Since Reynolds [1991] proved coherence for a calculus with intersection types, based on denotational semantics, many researchers have studied it in a variety of typed calculi. Below we summarize two commonly-found approaches in the literature: normal forms, and logical relations.

Breazu-Tannen et al. [1991] proved the coherence of a coercion translation from Fun [Cardelli & Wegner 1985] extended with recursive types to System F, by showing that any two typing derivations of the same judgment are normalizable to the same form. Curien & Ghelli [1994] presented a translation of System $F_{\leq}$ into a calculus with explicit coercions and showed that any derivations of the same judgment are translated to terms that are normalizable to a unique normal form. Following the same approach, Schwinghammer [2009] proved the coherence of coercion translation from Moggi's computational lambda calculus [Moggi 1991] with subtyping.

Central to the first approach is to find a unique normal form for derivations. However, this approach cannot be directly applied to Curry-style calculi, i.e., where the lambda abstractions are not type-annotated. Also, it does not work when the calculus has general recursion. Biernacki & Polesiuk [2015] considered

the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. They used a logical relation for establishing coherence for coercion semantics, applicable in a variety of calculi, including delimited continuations and control-effect subtyping.

As far as we know, the first work to use logical relations to show the coherence for intersection types and the merge operator was that of Bi et al. [2018] for NeColus, and later for F_i^+ [Bi et al. 2019]. The BCD subtyping in both calculi (as well as our own) poses a non-trivial complication over Bernacki and Polesiuk's simple structural subtyping. In their system any two coercions between given types are behaviorally equivalent in the target language, so their coherence reasoning can all take place in the target language. In NeColus, F_i^+, and λ_i^{MP} this is not the case; coercions can be distinguished by arbitrary target programs but not those that are elaborations of source programs. Hence, reasoning is restricted to the latter class, which is reflected in a more complicated notion of contextual equivalence and the logical relation's non-trivial treatment of pairs. In addition to these complications, our logical relation (and consequently the coherence proof) is significantly more convoluted than that of NeColus and F_i^+, due to the non-compositionality of modus ponens.

Intersection Types and the Merge Operator Forsythe [Reynolds 1988] is one of the first languages with intersection types and a merge-like operator. However, merges in Forsythe cannot contain more than one function; this restriction ensures that the language is coherent. Similarly, Castagna et al.'s $\lambda\&$ calculus [Castagna et al. 1992] includes a special merge operator that works on functions only. More recently, Dunfield [2012] showed significant expressiveness of type systems with intersection types and a merge operator, at the cost of coherence. The limitation was addressed by Oliveira et al. [2016], who introduced disjointness to ensure coherence. The combination of intersection types, a merge operator and parametric polymorphism—while achieving coherence—was first studied in the F_i calculus [Alpuim et al. 2017]. Its successor, F_i^+ [Bi et al. 2019] additionally incorporates BCD subtyping [Barendregt et al. 1983], while remaining coherent. NeColus [Bi et al. 2018] has been the starting point for our design of λ_i^{MP} and incorporates intersection types, a merge operator, and BCD-style distributivity. Though neither NeColus nor λ_i^{MP} accounts for parametric polymorphism, this is a direction we would like to explore in the future.

BCD Type System and Decidability The BCD type system was first introduced by Barendregt et al. [1983]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base

for ours. Bessai et al. [2014] showed how to type classes and mixins in a BCD-style record calculus with the merge operator of Bracha & Cook [1990]. Their merge only operates on records, and they only study a type assignment system. The decidability of BCD subtyping has been shown in several works [Kurata & Takahashi 1995; Pierce 1989; Rehof & Urzyczyn 2011; Statman 2014]. Laurent [2012] has formalized the relation in Coq in order to eliminate transitivity cuts from it, but he does not deliver an algorithm. Based on the work of Statman [2014], Bessai et al. [2016] show a formally verified subtyping algorithm in Coq. Our algorithm follows the decision procedure of Bi et al. [2018] and Pierce [1989]. However, the addition of modus ponens requires significant changes to the algorithm, and complicates the metatheory.

Coq Type Classes Coq employs a flexible type class system [Sozeau & Oury 2008] that permits overlapping instances and provides the user with the means to control resolution: Coq instances are named and can be explicitly passed as arguments. The system poses no restrictions over the implicit environment, like non-overlapping instances in Haskell or disjointness in λ_i^{MP}. Therefore, coherence becomes a choice and responsibility of the user.

In contrast, λ_i^{MP} employs a stricter system that provides less control over subtyping. First, implicit environments are not possible if not internally disjoint. Second, going against the intuition which relates classes to types and instances to appropriately typed terms, expressions in λ_i^{MP} can be rather viewed as typed implicit environments, which can be composed with the use of the merge operator. Passing a value to a function in λ_i^{MP} can be seen as bringing an implicit environment in scope within the body of the function. The function, then, can use that environment to infer values from it, but it does not have access to the names of the instances contained in it. This is conceptually different from the named instances of Coq that can be optionally passed explicitly by the user. Last, Coq uses a global environment to which instances are inserted at their point of declaration.

5.8 Scientific Output

The material found in this chapter is drawn from the following publication:

> Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as intersection subtyping via Modus Ponens. Proc. ACM Program. Lang. 4, OOPSLA, Article 206 (November 2020), 30 pages. DOI: `https://doi.org/10.1145/3428274`

The above is multi-author work, where my contribution lies mainly in its meta-theory related content. This work builds upon the NeColus calculus [Bi et al. 2018], which was delivered with a mechanization of its coherence proof in the Coq language. I adapted the NeColus mechanization to our λ_i^{MP} calculus and developed the partial coherence proof in Coq, which is based on the two conjectures discussed in this chapter. Furthermore, I did part of the work for proving the transitivity and termination properties of our subtyping algorithm, again in a mechanized fashion using the Coq language and the AProVe termination-checking tool [Giesl et al. 2017].

Chapter 6

Conclusion and Future Work

As is empirically known, the human brain has a low threshold in the amount of information it can process, compared to computers. This affects software productivity in two ways. Firstly, the development of complex software becomes a hard and error prone task for humans. Secondly, while code tangles up due to software complexity, it becomes harder to read it, and generally, to manipulate it. Fortunately, software developers can use implicit programming to hide information in their code, allowing them to write cleaner code that focuses on the more significant parts of the software design.

However, implicit programming may introduce semantic ambiguity in the language, in which case a program's behaviour becomes dependent on the language's implementation, harming the code's compatibility with different compilers. For that reason, the specification of a language's resolution mechanism should be predictable and independent from its implementation.

There are various designs of implicit programming that respect the semantic unambiguity condition, and each enables different features. This thesis aimed to explore the expressive power of these designs and to investigate meta-theoretical techniques for ensuring their correct behaviour.

6.1 Coherence of Type Class Resolution

Chapter 3 focuses on type class resolution (with superclasses), a well-known resolution mechanism where implicits, or type-class instances, have a global scope and are non-overlapping. While the expressive power of type classes is

well-studied, a formal proof for the coherence of its elaboration based semantics was lacking. This chapter has formally proven that type class resolution is coherent by means of logical relations and an intermediate language with explicit dictionaries.

Future Work An interesting direction for future work would be to mechanize the coherence proof and adapt it to extensions of type class resolution, such as *quantified class constraints* [Bottu et al. 2017], *type families* [Schrijvers et al. 2008] and/or *functional dependencies* [Jones 2000]. Furthermore, a formal exploration of the interaction of type classes with other language features, such as *GADT*'s and *existential types* [Läufer & Odersky 1994], would be very relevant, given their wide use in functional programming.

6.2 Cochis: Stable and Coherent Implicits

Chapter 4 presents COCHIS, the Calculus Of CoHerent ImplicitS, a new calculus for implicit programming that improves upon the implicit calculus and strikes a good balance between flexibility and desirable properties such as coherence and stability. COCHIS supports local scoping, overlapping implicits, first-class implicits, and higher-order rules, while remaining type safe, coherent and stable.

Future Work Interesting future work includes integrating Genus's solution for the instance coherence problem [Y. Zhang et al. 2015] in COCHIS; and adding more features that show up in various IP mechanisms, such as *associated types* [Chakravarty, Keller, & P. Jones 2005; Chakravarty, Keller, P. Jones, & Marlow 2005] and *type families*.

6.3 Resolution as Intersection Subtyping with Modus Ponens

Chapter 5 has shown that, with a small extension, subtyping with intersection types can be used to model resolution. We have materialized our ideas in the design of λ_i^{MP}, a type-safe extension of NeColus with the logical rule for modus ponens. This small extension significantly increases the expressive power of λ_i^{MP}'s predecessor, allowing us to model features such as type class resolution and first-class environments.

Future Work While a significant attempt has been made to formally prove coherence for λ_i^{MP}, the proof remains incomplete. Thus, a direction for future work is to improve λ_i^{MP}'s meta-theory so that coherence is provable. This means finding an appropriate definition of logical equivalence and perhaps introducing an intermediate elaboration step, similarly to the approach of Chapter 3. However, a difference is that in the $\boldsymbol{\lambda}_{\mathrm{TC}}$ calculus of Chapter 3 the context is statically known (instance definitions are given at compile time), while in λ_i^{MP} this is not always the case: subtyping can be triggered on a concrete value, but also on a function's parameter.

Apart from the meta-theory, a natural avenue for future work is the extension of λ_i^{MP} with parametric polymorphism, a missing important feature. When done naively, it can make the subtyping algorithm diverge and, like in Haskell, additional conditions will have to be imposed to recover termination. Moreover, there are various applications for our work that we wish to explore in the future.

In a Haskell-like setting, our design would enable multiple implicit environments, instead of a single global one. Such environments could be first-class, could be explicitly passed as arguments or returned as a result, or could be composed to form larger implicit environments. The programmer would then be able to choose which of the implicit environments to use for resolution.

In a Scala-like setting, where subtyping is present, our design helps clarify the interaction between resolution and subtyping. This is important, from a practical point of view, since the interaction between the two mechanisms has been a continuous source of compiler bugs[1]. Furthermore, our work could also be interesting to offer a unified view of subtyping and resolution, which could significantly reduce the implementation effort for the two mechanisms.

[1] See for example `https://github.com/scala/bug/issues/2509` or `https://github.com/scala/bug/issues/9764`.

Appendix A

System F

This appendix chapter contains the syntax and typing rules of the System F calculus, for reference. Section A.1 considers the pure System F calculus, which is used as the target calculus of COCHIS (Chapter 4). Section A.2 shows the syntax and typing rules of the $\mathbf{F}_{\{\}}$ calculus, which extends the pure System F calculus with booleans, records and let-expressions.

A.1 Pure System F

Figure A.1 shows the syntax (top) and typing rules (bottom) of the pure System F calculus.

A.2 Extensions for $\mathbf{F}_{\{\}}$

Figure A.2 shows the syntax of $\mathbf{F}_{\{\}}$. This figure extends the syntax of the pure System F calculus, shown in Figure A.1 (top). The typing rules for the new syntax are shown in Figure A.3.

$$
\begin{array}{llll}
\text{Types} & T & ::= & \alpha \mid T \to T \mid \forall \alpha.T \\
\text{Expressions} & E & ::= & x \mid \lambda(x:T).E \mid E\ E \mid \Lambda\alpha.E \mid E\ T
\end{array}
$$

$\boxed{\Gamma \vdash_{\mathrm{F}} E : T}$ System F Typing

$$\textsc{F-Var}\ \frac{(x:T) \in \Gamma}{\Gamma \vdash_{\mathrm{F}} x : T}$$

$$\textsc{F-Abs}\ \frac{\Gamma, x:T_1 \vdash_{\mathrm{F}} E : T_2}{\Gamma \vdash_{\mathrm{F}} \lambda x:T_1.E : T_1 \to T_2} \qquad \textsc{F-TAbs}\ \frac{\Gamma, \alpha \vdash_{\mathrm{F}} E : T}{\Gamma \vdash_{\mathrm{F}} \Lambda\alpha.E : \forall\alpha.T}$$

$$\textsc{F-App}\ \frac{\Gamma \vdash_{\mathrm{F}} E_1 : T_2 \to T_1 \qquad \Gamma \vdash_{\mathrm{F}} E_2 : T_2}{\Gamma \vdash_{\mathrm{F}} E_1\ E_2 : T_1} \qquad \textsc{F-TApp}\ \frac{\Gamma \vdash_{\mathrm{F}} E : \forall\alpha.T_2}{\Gamma \vdash_{\mathrm{F}} E\ T_1 : T_2[T_1/\alpha]}$$

Figure A.1: System F Syntax and Typing Rules

$$
\begin{array}{llll}
\text{Types} & T, \sigma & ::= & T \mid \mathit{Bool} \mid \{\overline{m_i : \sigma_i}^{\,i \in 1..n}\} \\
\text{Labels} & m & & \\
\text{Expressions} & E, e & ::= & E \mid \mathit{True} \mid \mathit{False} \mid \{\overline{m_i = e_i}^{\,i \in 1..n}\} \mid \\
 & & & \mathbf{let}\, x : \sigma = e_1\, \mathbf{in}\, e_2
\end{array}
$$

Figure A.2: $\mathbf{F}_{\{\}}$ syntax, extends Figure A.1 (top)

$\boxed{\Gamma \vdash_{\mathrm{F}} e : \sigma}$

$$\textsc{F-True}\ \frac{}{\Gamma \vdash_{\mathrm{F}} \mathit{True} : \mathit{Bool}} \qquad \textsc{F-False}\ \frac{}{\Gamma \vdash_{\mathrm{F}} \mathit{False} : \mathit{Bool}}$$

$$\textsc{F-Rec}\ \frac{\left(\Gamma \vdash_{\mathrm{F}} e_i : \sigma_i\ ,\ \forall i \in \{1, \ldots, n\}\right)}{\Gamma \vdash_{\mathrm{F}} \{\overline{m_i = e_i}^{\,i \in 1..n}\} : \{\overline{m_i : \sigma_i}^{\,i \in 1..n}\}}$$

$$\textsc{F-Let}\ \frac{\Gamma, x : \sigma_1 \vdash_{\mathrm{F}} e_2 : \sigma_2 \qquad \Gamma \vdash_{\mathrm{F}} e_1 : \sigma_1}{\Gamma \vdash_{\mathrm{F}} \mathbf{let}\, x : \sigma_1 = e_1\, \mathbf{in}\, e_2 : \sigma_2}$$

Figure A.3: $\mathbf{F}_{\{\}}$ typing rules, extends Figure A.1 (bottom)

Appendix B

Cochis Sketch of Proofs

This appendix section, which accompanies Chapter 4, delineates the proofs of Cochis' correctness properties. It states the lemmas involved in those proofs and shows their interdependecies by means of diagrams. In these diagrams, main results are indicated with double borders. The detailed proofs can be found at the appendix of [Schrijvers et al. 2019].

Several of the proofs have been mechanized in Coq; they are available from `https://bitbucket.org/KlaraMar/cochiscoq`.

B.1 Type Preservation

Lemma B.1 states that the translation of expressions to System F preserves types. Its proof relies on Lemma B.2, which states that the translation of resolution preserves types.

Lemma B.1. *If* $\Gamma \vdash e : \rho \rightsquigarrow E$*, then* $|\Gamma| \vdash_{\mathrm{F}} E : |\rho|$.

Lemma B.2. *If* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow e$*, then* $|\Gamma| \vdash_{\mathrm{F}} E : |\rho|$.

B.2 Auxiliary Lemmas About Non-Determistic Resolution

The non-deterministic resolution judgment, $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$, enjoys a number of typical binder-related properties.

The first lemma is the weakening lemma: it states that an extended type environment preserves all the derivations of the original environment.

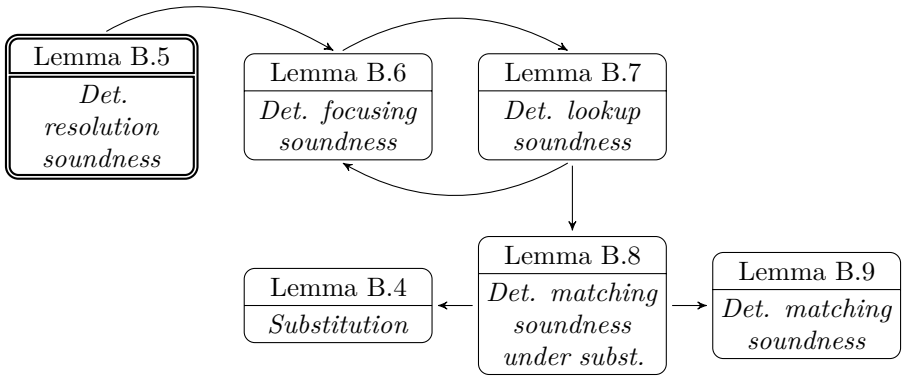
Lemma B.3 (Weakening). *If* $\Gamma, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$, *then* $\Gamma, \Gamma'', \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$.

The second lemma is the substitution lemma which states that we can drop an axiom from the type environment if it is already implied by the remainder of the type environment.

Lemma B.4 (Substitution). *If* $\Gamma, ?\rho \rightsquigarrow x, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho' \rightsquigarrow E'$ *and* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$, *then* $\Gamma, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho' \rightsquigarrow E'[E/x]$.

B.3 Soundness of Deterministic Resolution

The proofs of this section are split into subproofs for each sub-judgment (focusing, lookup and matching) of the deterministic resolution judgment. They all proceed by induction on the corresponding deterministic resolution judgment. Therefore, their dependencies, depicted in the diagram below, follow those of the sub-judgments that constitute the main deterministic resolution judgment. Because the proofs of this section are mechanized, the auxiliary Lemma B.6 is necessary to handle the multiple substitutions that rule L-Match uses.



Lemma B.5 states that deterministic resolution is sound with respect to non-deterministic resolution. It immediately follows from Lemma B.6.

Lemma B.5. *If* $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$*, then* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$*.*

Lemma B.6. *If* $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$*, then* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$*.*

The above lemma relies on the following auxiliary lemma for the resolution of simple types. The proof of this auxiliary lemma proceeds by mutual induction with Lemma B.6, while also relying on Lemma B.8.

Lemma B.7. *If* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$*, then* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \tau \rightsquigarrow E$*.*

Lemma B.8. *If* $\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{x}; \tau \rightsquigarrow E'$ *and* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$ *and* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{E}$*, then* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \tau \rightsquigarrow E'[\bar{E}/\bar{x}]$*.*

Lemma B.9. *If* $\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{x}; \tau \rightsquigarrow E'$ *and* $\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$*, then* $\Gamma, ?\bar{\rho} \rightsquigarrow \bar{x} \vdash^{\mathrm{a}}_{\mathrm{r}} \tau \rightsquigarrow E'$

B.4 Deterministic Resolution is Deterministic

Similarly to Subsection B.3, the proofs of this subsection are split into subproofs for each sub-judgment (focusing, lookup and matching) of the deterministic resolution judgment. Their dependencies, depicted in the diagram below, follow those of the sub-judgments that constitute the main deterministic resolution judgment.

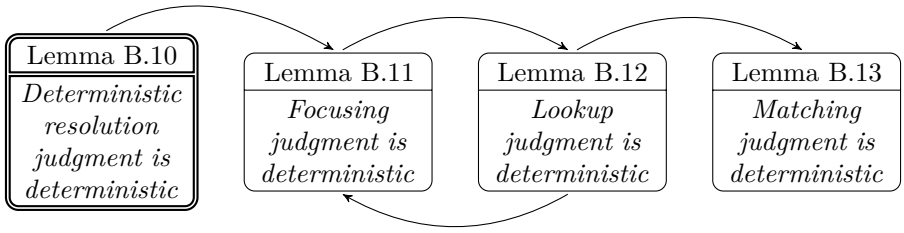


Figure B.1 defines the judgment $\vdash_{\mathrm{unamb}} \Gamma$ which states that a typing environment is unambiguous. Essentially, the definition requires that all types ρ in Γ are unambiguous. This definition will be used throughout this section.

$\boxed{\vdash_{\text{unamb}} \Gamma}$

$$\textsc{UE-Empty}\ \frac{}{\vdash_{\text{unamb}} \epsilon} \qquad \textsc{UE-Var}\ \frac{\vdash_{\text{unamb}} \Gamma}{\vdash_{\text{unamb}} \Gamma, x : \rho}$$

$$\textsc{UE-TVar}\ \frac{\vdash_{\text{unamb}} \Gamma}{\vdash_{\text{unamb}} \Gamma, \alpha} \qquad \textsc{UE-Impl}\ \frac{\vdash_{\text{unamb}} \Gamma \qquad \vdash_{\text{unamb}} \rho}{\vdash_{\text{unamb}} \Gamma, ?\rho \rightsquigarrow x}$$

Figure B.1: Unambiguous Type Environment

$\boxed{\bar{\sigma}; \Gamma; [\rho] \rightsquigarrow E \vdash_{\text{r}} \Sigma; \tau \rightsquigarrow E'}$

$$\textsc{M-Simp'}\ \frac{}{\epsilon; \Gamma; [\tau] \rightsquigarrow E \vdash_{\text{r}} \epsilon; \tau \rightsquigarrow E}$$

$$\textsc{M-IApp'}\ \frac{\bar{\sigma}; \Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\, x \vdash_{\text{r}} \Sigma; \tau \rightsquigarrow E' \qquad x\ \textit{fresh}}{\bar{\sigma}; \Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash_{\text{r}} \rho_1 \rightsquigarrow x, \Sigma; \tau \rightsquigarrow E'}$$

$$\textsc{M-TApp'}\ \frac{\bar{\sigma}; \Gamma; [\rho[\sigma/\alpha]] \rightsquigarrow E\, |\sigma| \vdash_{\text{r}} \Sigma; \tau \rightsquigarrow E' \qquad \Gamma \vdash \sigma}{\bar{\sigma}, \sigma; \Gamma; [\forall \alpha.\rho] \rightsquigarrow E \vdash_{\text{r}} \Sigma; \tau \rightsquigarrow E'}$$

Figure B.2: Annotated matching judgment

Lemma B.10. *If* $\vdash_{\text{unamb}} \Gamma$ *and* $\vdash_{\text{unamb}} \rho$ *and* $\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_1$ *and* $\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_2$, *then* $E_1 = E_2$.

Lemma B.11. *If* $\vdash_{\text{unamb}} \Gamma$ *and* $\vdash_{\text{unamb}} \rho$ *and* $\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E_1$ *and* $\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E_2$, *then* $E_1 = E_2$.

Lemma B.12. *If* $\vdash_{\text{unamb}} \Gamma$ *and* $\vdash_{\text{unamb}} \Gamma'$ *and* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{r}} \tau \rightsquigarrow E_1$ *and* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{r}} \tau \rightsquigarrow E_2$, *then* $E_1 = E_2$.

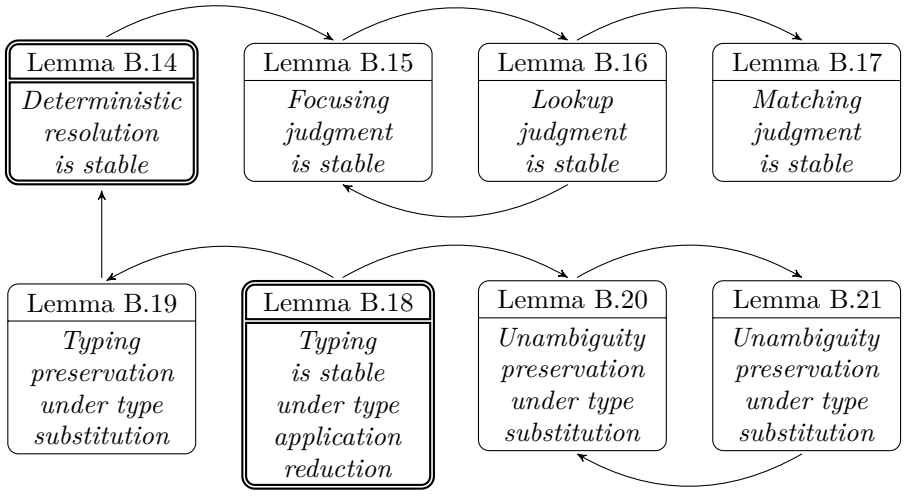
Before proceeding with the matching judgment, we present in Figure B.2 a version of it that is annotated with the sequence of substitution types $\bar{\sigma}$ used to instantiate the universal quantifiers. It is not difficult to see that any derivation of the annotated judgment is in one to one correspondence with a derivation of the unannotated judgment.

Now we are ready to show that the judgment is deterministic.

Lemma B.13. *If* $\bar{\alpha} \vdash_{\text{unamb}} \rho$ *and* $\bar{\sigma}_1; \Gamma; [\rho[\bar{\sigma}_2/\bar{\alpha}]] \rightsquigarrow E[|\bar{\sigma}_2|/\bar{\alpha}] \vdash_{\text{r}} \Sigma_1; \tau \rightsquigarrow E_1$ *and* $\bar{\sigma}'_1; \Gamma; [\rho[\bar{\sigma}'_2/\bar{\alpha}]] \rightsquigarrow E[|\bar{\sigma}'_2|/\bar{\alpha}] \vdash_{\text{r}} \Sigma_2; \tau \rightsquigarrow E_2$,
then $\bar{\sigma}_1 = \bar{\sigma}'_1$ *and* $\bar{\sigma}_2 = \bar{\sigma}'_2$ *and* $E_1 = E_2$ *and* $\Sigma_1 = \Sigma_2$ *and* $\vdash_{\text{unamb}} \Sigma_1$.

B.5 Resolution and Typing Stability

The proofs of this subsection concern stability of the deterministic resolution judgment and of typing. The first part, similarly to Subection B.4, is split into subproofs for each auxiliary judgment (focusing, lookup and matching) of the deterministic resolution judgment and follows their dependencies. The second part uses the main result of the first part to establish stability of typing under type application reduction. In addition, it uses the preservation-under-type-substitution properties of typing and of the unambiguity judgment. The dependencies of the proofs of this subsection are depicted in the diagram below, where main results are indicated with double borders.



Deterministic resolution is stable under type substitution.

Lemma B.14. *If* $\Gamma, \alpha, \Gamma' \vdash_{\text{r}} \rho \rightsquigarrow E$, *then* $\Gamma, \Gamma'[\sigma/\alpha] \vdash_{\text{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$.

The next lemmas make use of the auxiliary definition R, shown in Figure B.3. The relation $R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$ states that when performing a type

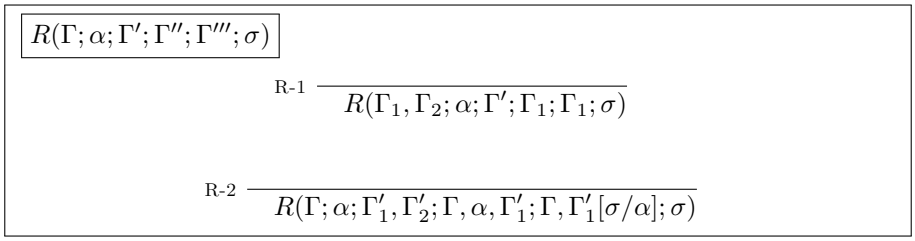


Figure B.3: Auxiliary Definition R

substitution $\Gamma,\alpha,\Gamma' \vdash [\sigma/\alpha]$, then any $\Gamma'' \subseteq \Gamma,\alpha,\Gamma'$, results to Γ''' after the substitution has been performed. The two cases of the relation are necessary to separate between $\alpha \in \Gamma''$ and $\alpha \notin \Gamma''$.

Lemma B.15. *If* $\bar{\alpha},\alpha,\bar{\alpha};\Gamma,\alpha,\Gamma' \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$,
then $\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$.

Lemma B.16. *If* $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma';[\Gamma''] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$ *and* $\Gamma'' \subseteq \Gamma,\alpha,\Gamma'$,
then $\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha];[\Gamma'''] \vdash_{\mathrm{r}} \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$ *and* $R(\Gamma;\alpha;\Gamma';\Gamma'';\Gamma''';\sigma)$.

Lemma B.17. *If* $\Gamma,\alpha,\Gamma';[\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{x};\tau \rightsquigarrow E'$,
then $\Gamma,\Gamma'[\sigma/\alpha];[\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha] \vdash_{\mathrm{r}} \bar{\rho}[\sigma/\alpha] \rightsquigarrow \bar{x};\tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$.

Lemma B.18. *If* $\Gamma \vdash (\Lambda\alpha.e)\,\sigma : \rho[\sigma/\alpha] \rightsquigarrow (\Lambda\alpha.E)\,|\sigma|$,
then $\Gamma \vdash e[\sigma/\alpha] : \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$.

Lemma B.19. *If* $\Gamma,\alpha,\Gamma' \vdash e : \rho \rightsquigarrow E$,
then $\Gamma,\Gamma'[\sigma/\alpha] \vdash e[\sigma/\alpha] : \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$.

Lemma B.20. *If* $\vdash_{\mathrm{unamb}} \rho$, *then* $\vdash_{\mathrm{unamb}} \rho[\sigma/\alpha]$.

Lemma B.21. *If* $\bar{\alpha} \vdash_{\mathrm{unamb}} \rho$ *and* $\alpha \notin \bar{\alpha}$, *then* $\bar{\alpha} \vdash_{\mathrm{unamb}} \rho[\sigma/\alpha]$.

B.6 Auxiliary Lemmas About the Unification Algorithm

This subsection concerns properties of the unification algorithm that are required by the proofs of the next sections. For this purpose, a notion of a partial order

over substitutions is also used.

Definition B.1 (Partial Order over Substitutions)**.** *A substitution θ_1 is more general than a substitution θ_2, denoted with $\theta_2 \sqsubseteq \theta_1$, iff there is a θ' such that $\theta_2 = \theta' \cdot \theta_1$.*

Lemma B.22 states that the unification algorithm produces a substitution that indeed unifies its two input types.

Lemma B.22. *If $\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$, then $\theta(\tau) = \theta(\tau')$ and $\bar{\alpha}; \Gamma \vdash \theta$.*

We assume the following two properties of unification as given. Assumption B.1 states that $\mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$ is the most general unifier of τ and τ'.

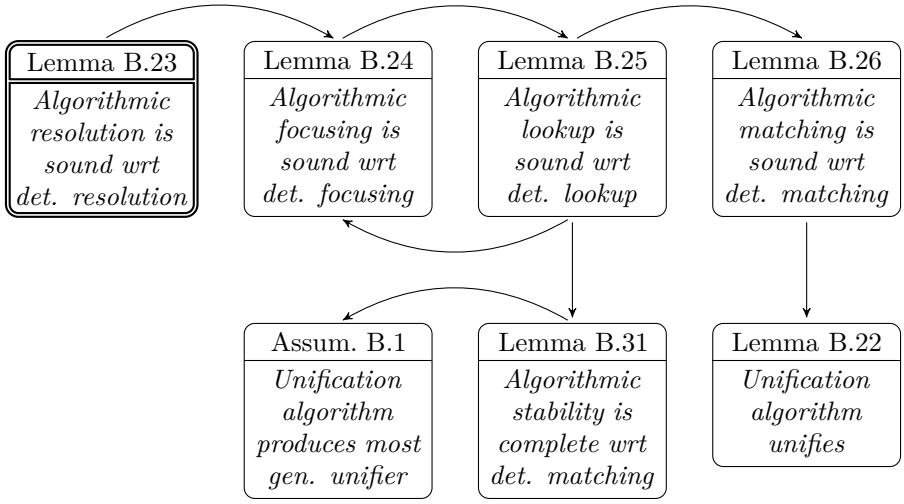
Assumption B.1. *If $\theta(\tau) = \theta(\tau')$ and $\bar{\alpha}; \Gamma \vdash \theta$, then $\theta \sqsubseteq \mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$.*

Assumption B.2. *If $\theta(\tau) = \tau'$ and $\bar{\alpha}; \Gamma \vdash \theta$ and $\bar{\alpha} \subseteq ftv(\tau)$ and if $\forall \alpha \in \bar{\alpha}, \forall \beta \in ftv(\tau'), \beta >_\Gamma \alpha$, then $\theta = \mathsf{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$.*

B.7 Soundness of the Algorithmic wrt Deterministic Resolution

This subsection proves soundness of the resolution algorithm with respect to its declarative specification. As summarized in the diagram below, the main lemma depends on three auxiliary soundness lemmas, one for each of the algorithmic focusing, lookup and matching judgments. The dependencies of these proofs follow those of the definitions of the corresponding judgments.

Interestingly, the proof for algorithmic lookup soundness also uses completeness of the algorithmic stability judgment. The latter requires a correctness property of the unification algorithm. One more auxiliary correctness property is used by the algorithmic matching soundness proof.



Lemma B.23. *If* $\Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E$*, then* $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$*.*

Lemma B.24. *If* $\bar{\alpha}; \Gamma \vdash_{\mathrm{g}} \rho \rightsquigarrow E$*, then* $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$

Lemma B.25. *If* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{g}} \rho \rightsquigarrow E$*, then* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \rho \rightsquigarrow E$*.*

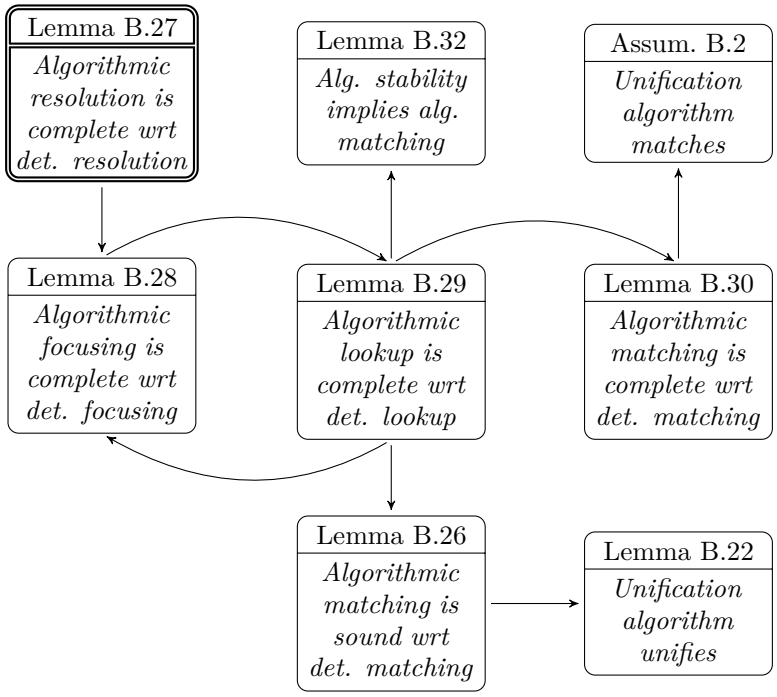
Lemma B.26. *If* $\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'$*, then there is a substitution* θ *such that* $\bar{\alpha}; \Gamma \vdash \theta$ *and* $\theta(\Gamma); [\theta(\rho)] \rightsquigarrow |\theta|(E) \vdash_{\mathrm{r}} \Sigma''; \theta(\tau) \rightsquigarrow E'$*, for* Σ'' *such that* $\Sigma' = \theta(\Sigma), \Sigma''$*.*

B.8 Completeness of the Algorithm wrt Deterministic Resolution

This section proves completeness of the resolution algorithm with respect to its declarative counterpart. Like in Section B.7, the main lemma depends on three auxiliary completeness lemmas, one for each of the focusing, lookup and matching judgments. The dependencies of these proofs follow those of the definitions of the corresponding declarative judgments.

In symmetry with the proof of the auxiliary lookup lemma for algorithmic soundness, the proof for algorithmic lookup completeness also uses soundness of the algorithmic matching judgment, resulting in interdependent soundness

and completeness proofs. Additionally, an auxiliary correctness property of the unification algorithm is necessary for proving completeness of algorithmic matching.



Lemma B.27. *If* $\vdash_{\text{unamb}} \Gamma$ *and* $\vdash_{\text{unamb}} \rho$ *and* $\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E$, *then* $\Gamma \vdash_{\text{g}} \rho \rightsquigarrow E$.

Lemma B.28. *If* $\vdash_{\text{unamb}} \Gamma$ *and* $\vdash_{\text{unamb}} \rho$ *and* $\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E$, *then* $\bar{\alpha}; \Gamma \vdash_{\text{g}} \rho \rightsquigarrow E$.

Lemma B.29. *If* $\vdash_{\text{unamb}} \Gamma'$ *and* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{r}} \rho \rightsquigarrow E$, *then* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{g}} \rho \rightsquigarrow E$.

Lemma B.30. *If* $\theta(\Gamma); [\theta(\rho)] \rightsquigarrow |\theta|(E) \vdash_{\text{r}} \Sigma'; \tau \rightsquigarrow E'$ *and* $\bar{\alpha} \vdash_{\text{unamb}} \rho$ *and* $dom(\theta) \subseteq \bar{\alpha}$ *and if* $\forall \alpha \in \bar{\alpha}, \forall \beta \in ftv(\tau), \beta >_\Gamma \alpha$, *then* $\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\text{g}} \theta(\Sigma), \Sigma'; \tau \rightsquigarrow E'$, *for all* Σ.

Lemma B.31. *If* $\theta_1(\Gamma); [\theta_1(\rho)] \rightsquigarrow E \vdash_{\text{r}} \Sigma'; \theta_1(\tau) \rightsquigarrow E'$ *and* $\bar{\alpha}; \Gamma \vdash \theta_1$, *then* $\bar{\alpha}; \Gamma; \rho \vdash_{\text{sta}} \tau$.

Lemma B.32. *If* $\bar{\alpha}; \Gamma; \rho \vdash_{\mathrm{sta}} \tau$*, then for all* E, Σ *there exist* E', Σ' *such that* $\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\mathrm{g}} \Sigma'; \tau \rightsquigarrow E'$.

Appendix C

λ_i^{MP} and λ_c^{MP} Related Sources

C.1 Complete Specification of λ_c^{MP}

$\boxed{c \vdash \tau_1 \rhd \tau_2}$ Coercion Typing

$$\textsc{CT-refl}\ \frac{}{\mathsf{id}_\tau \vdash \tau \rhd \tau} \qquad \textsc{CT-trans}\ \frac{c_2 \vdash \tau_2 \rhd \tau_3 \qquad c_1 \vdash \tau_1 \rhd \tau_2}{c_2 \circ c_1 \vdash \tau_1 \rhd \tau_3} \qquad \textsc{CT-top}\ \frac{}{\mathsf{top}_\tau \vdash \tau \rhd \mathsf{Unit}}$$

$$\textsc{CT-topArr}\ \frac{}{\mathsf{top}_\to \vdash \mathsf{Unit} \rhd \mathsf{Unit} \to \mathsf{Unit}} \qquad \textsc{CT-mp}\ \frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd (\tau_2 \to \tau_3)}{c_2 \mathbin{\ominus} c_1 \vdash \tau_1 \rhd \tau_3}$$

$$\textsc{CT-arr}\ \frac{c_1 \vdash \tau_1' \rhd \tau_1 \qquad c_2 \vdash \tau_2 \rhd \tau_2'}{c_1 \to c_2 \vdash \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'} \qquad \textsc{CT-pair}\ \frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd \tau_3}{\langle c_1, c_2 \rangle \vdash \tau_1 \rhd \tau_2 \times \tau_3}$$

$$\textsc{CT-projL}\ \frac{}{\pi_1^{\tau_1,\tau_2} \vdash \tau_1 \times \tau_2 \rhd \tau_1} \qquad \textsc{CT-projR}\ \frac{}{\pi_2^{\tau_1,\tau_2} \vdash \tau_1 \times \tau_2 \rhd \tau_2}$$

$$\textsc{CT-distArr}\ \frac{}{\mathsf{dist}_\to^{\tau_1,\tau_2,\tau_3} \vdash (\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3) \rhd \tau_1 \to \tau_2 \times \tau_3}$$

$\boxed{\Delta \vdash e : \tau}$ Term Typing

$$\textsc{typ-var}\ \frac{x : \tau \in \Delta}{\Delta \vdash x : \tau} \qquad \textsc{typ-unit}\ \frac{}{\Delta \vdash \langle\rangle : \mathsf{Unit}} \qquad \textsc{typ-lit}\ \frac{}{\Delta \vdash i : \mathsf{Int}}$$

$$\textsc{typ-abs}\ \frac{\Delta, x : \tau_1 \vdash e : \tau_2}{\Delta \vdash \lambda x : \tau_1.\, e : \tau_1 \to \tau_2} \qquad \textsc{typ-app}\ \frac{\Delta \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta \vdash e_2 : \tau_1}{\Delta \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{typ-pair}\ \frac{\Delta \vdash e_1 : \tau_1 \qquad \Delta \vdash e_2 : \tau_2}{\Delta \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \qquad \textsc{typ-capp}\ \frac{\Delta \vdash e : \tau \qquad c \vdash \tau \triangleright \tau'}{\Delta \vdash c\ e : \tau'}$$

$\boxed{e \longrightarrow e'}$ Operational Semantics

$$\textsc{step-id}\ \frac{}{(\mathsf{id}_\tau)\ v \longrightarrow v} \qquad \textsc{step-trans}\ \frac{}{(c_1 \circ c_2)\ v \longrightarrow c_1\ (c_2\ v)} \qquad \textsc{step-mp}\ \frac{}{(c_2 \multimap c_1)\ v \longrightarrow (c_2\ v)\ (c_1\ v)}$$

$$\textsc{step-top}\ \frac{}{\mathsf{top}_\tau\ v \longrightarrow \langle\rangle} \qquad \textsc{step-topArr}\ \frac{}{(\mathsf{top}_\to\ \langle\rangle)\ \langle\rangle \longrightarrow \langle\rangle} \qquad \textsc{step-arr}\ \frac{}{((c_1 \to c_2)\ v_1)\ v_2 \longrightarrow c_2\ (v_1\ (c_1\ v_2))}$$

$$\textsc{step-pair}\ \frac{}{\langle c_1, c_2\rangle\ v \longrightarrow \langle c_1\ v, c_2\ v\rangle} \qquad \textsc{step-distArr}\ \frac{}{(\mathsf{dist}_\to^{\tau,\tau_1,\tau_2}\ \langle v_1, v_2\rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3\rangle}$$

$$\textsc{step-projl}\ \frac{}{\pi_1^{\tau_1,\tau_2}\ \langle v_1, v_2\rangle \longrightarrow v_1} \qquad \textsc{step-projr}\ \frac{}{\pi_2^{\tau_1,\tau_2}\ \langle v_1, v_2\rangle \longrightarrow v_2} \qquad \textsc{step-beta}\ \frac{}{(\lambda x : \tau.\, e)\ v \longrightarrow [v/x]e}$$

$$\textsc{step-app1}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \textsc{step-app2}\ \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'} \qquad \textsc{step-capp}\ \frac{e \longrightarrow e'}{c\ e \longrightarrow c\ e'}$$

$$\textsc{step-pair1}\ \frac{e_1 \longrightarrow e_1'}{\langle e_1, e_2\rangle \longrightarrow \langle e_1', e_2\rangle} \qquad \textsc{step-pair2}\ \frac{e_2 \longrightarrow e_2'}{\langle v_1, e_2\rangle \longrightarrow \langle v_1, e_2'\rangle}$$

C.2 Context Typing

$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$ Context Typing I

$$\textsc{CTyp-empty1}\ \frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

$$\textsc{CTyp-appL1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\textsc{CTyp-appR1}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\textsc{CTyp-mergeL1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C} \,,, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\textsc{CTyp-mergeR1}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1 \,,, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\textsc{CTyp-anno1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{(\mathcal{C} : A) : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$ Context Typing II

$$\textsc{CTyp-empty2}\ \frac{}{[\cdot] : (\Gamma \Leftarrow A) \mapsto (\Gamma \Leftarrow A) \rightsquigarrow [\cdot]}$$

$$\textsc{CTyp-abs2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x.\, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\, \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$ Context Typing III

$$\textsc{CTyp-appL2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\textsc{CTyp-appR2}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\textsc{CTyp-mergeL2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C} \,,, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\textsc{CTyp-mergeR2}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1 \,,, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\textsc{CTyp-anno2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$ Context Typing IV

$$\textsc{CTyp-abs1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x.\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\, \mathcal{D}}$$

C.3 Additional Definitions and Meta-functions

Definition C.1 (Type Translation).

$$\begin{array}{lcl} |A| & = & \mathsf{Int} \\ |\top| & = & \langle\rangle \\ |A \to B| & = & |A| \to |B| \\ |A \,\&\, B| & = & |A| \times |B| \end{array}$$

Definition C.2 (Typing Context Translation).

$$\begin{array}{lcl} |\bullet| & = & \bullet \\ |\Gamma, x : A| & = & |\Gamma|, x : |A| \end{array}$$

Definition C.3 (Well-formed typing context).

$\boxed{\vdash \Gamma}$

$$\textit{WFC-empty}\ \frac{}{\vdash \bullet} \qquad\qquad \textit{WFC-cons}\ \frac{\vdash \Gamma \qquad x \notin \Gamma}{\vdash \Gamma, x : A}$$

The three meta-functions used in Section 5.5 are formally defined as follows:

Definition C.4.

$$\begin{array}{rcl} [\![[]]\!]_{\&}^{c,B_1,B_2} & = & c \\ [\![A,\mathcal{L}]\!]_{\&}^{c,B_1,B_2} & = & (\mathsf{id}_{|A|} \to [\![\mathcal{L}]\!]_{\&}^{c,B_1,B_2}) \circ \mathsf{dist}_{\to}^{|A|,|\mathcal{L}\to B_1|,|\mathcal{L}\to B_2|} \end{array}$$

Definition C.5.

$$[\![\mathcal{L}]\!]_{\&}^{B_1,B_2} = [\![\mathcal{L}]\!]_{\&}^{(\mathsf{id}_{|B_1\ \&\ B_2|}),B_1,B_2}$$

Definition C.6.

$$\begin{array}{rcl} [\![[]]\!]_{\top} & = & \mathsf{id}_{|\top|} \\ [\![A,\mathcal{L}]\!]_{\top} & = & (\mathsf{top}_{|A|} \to [\![\mathcal{L}]\!]_{\top}) \circ \mathsf{top}_{\to} \end{array}$$

C.4 Complete Derivation Examples of Typing and Algorithmic Subtyping

Typing Derivation Example Below is the typing derivation for the example expression $1 : \mathsf{Int} \,\&\, \mathsf{Int}$, taken from Subsection 5.4.4.

$$\dfrac{\dfrac{\dfrac{\vdash \bullet}{\bullet \vdash 1 \Rightarrow \mathsf{Int} \rightsquigarrow 1}\ \text{T-NAT} \qquad \dfrac{\dfrac{}{\mathsf{Int} <: \mathsf{Int} \rightsquigarrow \mathsf{id}_{\mathsf{Int}}}\ \text{S-REFL} \quad \dfrac{}{\mathsf{Int} <: \mathsf{Int} \rightsquigarrow \mathsf{id}_{\mathsf{Int}}}\ \text{S-REFL}}{\mathsf{Int} <: \mathsf{Int} \,\&\, \mathsf{Int} \rightsquigarrow \langle \mathsf{id}_{\mathsf{Int}}, \mathsf{id}_{\mathsf{Int}} \rangle}\ \text{S-AND}}{\bullet \vdash 1 \Leftarrow \mathsf{Int} \,\&\, \mathsf{Int} \rightsquigarrow \langle \mathsf{id}_{\mathsf{Int}}, \mathsf{id}_{\mathsf{Int}} \rangle\, 1}\ \text{T-SUB}}{\bullet \vdash 1 : \mathsf{Int} \,\&\, \mathsf{Int} \Rightarrow \mathsf{Int} \,\&\, \mathsf{Int} \rightsquigarrow \langle \mathsf{id}_{\mathsf{Int}}, \mathsf{id}_{\mathsf{Int}} \rangle\, 1}\ \text{T-ANNO}$$

The generated target term is $\langle \mathsf{id}_{\mathsf{Int}}, \mathsf{id}_{\mathsf{Int}} \rangle\, 1$, which reduces to $\langle 1, 1 \rangle$, following the operational semantics of λ_c^{MP}.

Algorithmic Subtyping Derivation Example Here we illustrate how the algorithm of Section 5.5.1 works with an example derivation. Due to space restrictions, we shorten Int, String, and Bool to N, S, and B, respectively, and omit all mention of coercions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash_{\mathrm{R}} \mathsf{S} \prec: \top}\ \text{AR-TOP} \qquad D_2}{([], \mathsf{S}); []; \top \to (\mathsf{B} \,\&\, (\mathsf{B} \to \mathsf{N})) \vdash_{\mathrm{L}} \top \to (\mathsf{B} \,\&\, (\mathsf{B} \to \mathsf{N})) \prec: \mathsf{N}}\ \text{AL-ARR}}{([], \mathsf{S}) \vdash_{\mathrm{R}} \top \to (\mathsf{B} \,\&\, (\mathsf{B} \to \mathsf{N})) \prec: \mathsf{N}}\ \text{AR-INT}}{[] \vdash_{\mathrm{R}} \top \to (\mathsf{B} \,\&\, (\mathsf{B} \to \mathsf{N})) \prec: \mathsf{S} \to \mathsf{N}}\ \text{AR-ARR}}{\top \to (\mathsf{B} \,\&\, (\mathsf{B} \to \mathsf{N})) \prec: \mathsf{S} \to \mathsf{N}}\ \text{A-MAIN}$$

where D_2 is the following derivation:

$$\dfrac{\dfrac{\dfrac{D_3 \qquad \dfrac{}{[];(\mathsf{S},[]);\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \mathsf{N} \prec: \mathsf{N}}\,\textsc{al-int}}{[];(\mathsf{S},[]);\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \mathsf{B}\to\mathsf{N} \prec: \mathsf{N}}\,\textsc{al-mp}}{[];(\mathsf{S},[]);\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N}) \prec: \mathsf{N}}\,\textsc{al-and2}$$

where D_3 is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash_{\mathrm{R}} \mathsf{S} \prec: \top}\,\textsc{ar-top} \qquad D_4}{([],\mathsf{S});[];\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \prec: \mathsf{B}}\,\textsc{al-arr}}{([],\mathsf{S}) \vdash_{\mathrm{R}} \top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \prec: \mathsf{B}}\,\textsc{ar-bool}}{[] \vdash_{\mathrm{R}} \top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \prec: \mathsf{S}\to\mathsf{B}}\,\textsc{ar-arr}$$

where D_4 is the following derivation:

$$\dfrac{\dfrac{}{([],\mathsf{S});[];\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \mathsf{B} \prec: \mathsf{B}}\,\textsc{al-bool}}{([],\mathsf{S});[];\top \to (\mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N})) \vdash_{\mathrm{L}} \mathsf{B}\,\&\,(\mathsf{B}\to\mathsf{N}) \prec: \mathsf{B}}\,\textsc{al-and1}$$

C.5 The Logical Relation for λ_i^{MP}

Figure C.1 shows the λ_i^{MP} definitions of the logical relations $\mathcal{V}$ for values and $\mathcal{E}$ for terms. These definitions reconcile the context-dependent nature of modus ponens with the structural recursion needed for the well-foundedness of the definitions.

The intent of the relations is of course the same as in NeColus, to only allow semantically indistinguishable overlap. Also the three general principles at the basis of their definition are preserved: 1) two values of type Int have indistinguishable overlap only when they are identical, 2) related functions should take related inputs to related outputs, and 3) the coercion compatibility property should hold.

The first novelty to deal with modus ponens is that the family of value relations is equipped with additional indices $v_1' : \tau_1'$ and $v_2' : \tau_2'$ to keep track of the values that v_1 and v_2 are components of; we call these indices the respective *contexts* of v_1 and v_2. To be precise, the intention is that we use $(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1;\tau_2]\!]$ only when there exist coercions c_1 and c_2 such that firstly $c_1 \vdash \tau_1' \triangleright \tau_1$ and $c_2 \vdash \tau_2' \triangleright \tau_2$, and secondly $c_1\ v_1' \longrightarrow^* v_1$ and $c_2\ v_2' \longrightarrow^* v_2$. When the contexts and the values are the same, we sometimes omit those additional

$\boxed{(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1; \tau_2]\!]}$ Value Relation

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\mathsf{Int}; \mathsf{Int}]\!] \triangleq \exists i, v_1 = v_2 = i$$

$$(\langle v_1, v_2\rangle, v_3) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1 \times \tau_2; \tau_3]\!] \triangleq (v_1, v_3) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1; \tau_3]\!] \\ \wedge\ (v_2, v_3) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_2; \tau_3]\!]$$

$$(v_3, \langle v_1, v_2\rangle) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_3; \tau_1 \times \tau_2]\!] \triangleq (v_3, v_1) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_3; \tau_1]\!] \\ \wedge\ (v_3, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_3; \tau_2]\!]$$

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1 \to \tau_2; \tau]\!] \triangleq \qquad (\tau \text{ not function type}) \\ \forall c, c \vdash \tau_1' \triangleright \tau_1 \Rightarrow (v_1\ (c\ v_1'), v_2) \in \mathcal{E}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_2; \tau]\!]$$

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau; \tau_1 \to \tau_2]\!] \triangleq \qquad (\tau \text{ not function type}) \\ \forall c, c \vdash \tau_2' \triangleright \tau_1 \Rightarrow (v_1, v_2\ (c\ v_2')) \in \mathcal{E}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau; \tau_2]\!]$$

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1 \to \tau_2; \tau_3 \to \tau_4]\!] \triangleq \\ \left(\begin{array}{l} \forall v\ v'\ \tau_0\ \tau_0'\ c\ c'\ d\ d'\ \tau_{01}\ \tau_{02}, \\ d \vdash \tau_{01} \triangleright \tau_1 \Rightarrow d' \vdash \tau_{02} \triangleright \tau_3 \Rightarrow (d\ v, d'\ v') \in \mathcal{E}^{(v:\tau_{01};v':\tau_{02})}[\![\tau_1; \tau_3]\!] \Rightarrow \\ c \vdash \tau_1' \triangleright \tau_{01} \to \tau_0 \Rightarrow c' \vdash \tau_2' \triangleright \tau_{02} \to \tau_0' \Rightarrow \\ (v_1\ (d\ v), v_2\ (d'\ v')) \in \mathcal{E}^{(\langle v_1\ (d\ v), c\ v_1'\ v\rangle:\tau_2\times\tau_0;\langle v_2\ (d'\ v'), c'\ v_2'\ v'\rangle:\tau_4\times\tau_0')}[\![\tau_2; \tau_4]\!] \end{array}\right) \\ \wedge\ \forall c, c \vdash \tau_1' \triangleright \tau_1 \Rightarrow (v_1\ (c\ v_1'), v_2) \in \mathcal{E}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_2; \tau_3 \to \tau_4]\!] \\ \wedge\ \forall c, c \vdash \tau_2' \triangleright \tau_3 \Rightarrow (v_1, v_2\ (c\ v_2')) \in \mathcal{E}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1 \to \tau_2; \tau_4]\!]$$

$$(v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1; \tau_2]\!] \triangleq \mathsf{true} \qquad \text{(otherwise)}$$

$\boxed{(e_1, e_2) \in \mathcal{E}^{(e_1':\tau_1';e_2':\tau_2')}[\![\tau_1; \tau_2]\!]}$ Expression Relation

$$(e_1, e_2) \in \mathcal{E}^{(e_1':\tau_1';e_2':\tau_2')}[\![\tau_1; \tau_2]\!] \triangleq \exists v_1\ v_2\ v_1'\ v_2', \\ e_1 \longrightarrow^* v_1\ \wedge\ e_2 \longrightarrow^* v_2 \wedge\ e_1' \longrightarrow^* v_1'\ \wedge\ e_2' \longrightarrow^* v_2' \\ \wedge\ (v_1, v_2) \in \mathcal{V}^{(v_1':\tau_1';v_2':\tau_2')}[\![\tau_1; \tau_2]\!]$$

Figure C.1: Logical Relations

indices, i.e., we abbreviate $(v_1, v_2) \in \mathcal{V}^{(v_1:\tau_1;v_2:\tau_2)}[\![\tau_1;\tau_2]\!]$ to $(v_1, v_2) \in \mathcal{V}[\![\tau_1;\tau_2]\!]$. The term relation $\mathcal{E}$ is equipped with similar context indices; the one difference is that, just like the elements in the term relation, these indices are terms rather than values.

Unmodified Cases The first three cases of the $\mathcal{V}$ definition are essentially the same as those of NeColus: Two natural numbers are related if they are identical, which is a direct implementation of the first general principle. Also, a pair is related to another value if both components of the pair are related to that value, which is a similar specialisation of the coercion compatibility property to coercions π_1 and π_2.

New Modus Ponens Cases The next two cases that relate functions to non-functions are different from NeColus because of the addition of the modus ponens coercion. In NeColus all functions are related to all non-functions because the only common types they can be coerced to are top-like types whose values are indistinguishable.

This is not so in λ_c^{MP}, where the S-MP rule can be used to derive a non-function value from a function. Hence, our logical relation states that a function and non-function are related only if all S-MP-enabled applications of the function are related to the non-function value. Using modus ponens implies that not only the function, but also its argument is derived from the original value. This is why the two symmetric cases of the relation universally quantify over all coercions that derive an argument of the appropriate input type.

New Function Case The one-but-last case, which relates two functions, is the most complicated one. It states that two functions are related iff they satisfy three conditions. These three conditions concern the two different ways in which the functions can be applied to an argument.

The second and third condition are a new way compared to NeColus and analogous to function/non-function cases; these are the cases where one of the two functions is applied to an argument through S-MP. They expose that $\langle \lambda x : \mathsf{Int}.\, \lambda y : \mathsf{Bool}.\, y, 1 \rangle$ can be coerced to $\lambda y : \mathsf{Bool}.\, y$ and thus has distinguishable overlap with, for instance, $\lambda y : \mathsf{Bool}.\, \mathsf{true}$.

The first condition corresponds to the NeColus condition for function types and captures the main design principle that related functions take related inputs to related outputs. However, as a consequence of modus ponens subtyping, the formulation is much more elaborate. First, we need to consider the context

indices. An explicit function application demarcates the point where the implicit subtyping derivation ends and, therefore, the previous context is abandoned. One might thus expect the following much simpler formulation of the condition, where the result of the function application becomes its own new context:

$$\forall(v, v') \in \mathcal{V}^{(v:\tau_1;v':\tau_3)}[\![\tau_1;\tau_3]\!],\ (v_1\ v, v_2\ v') \in \mathcal{E}^{(v_1\ v:\tau_2;v_2\ v':\tau_4)}[\![\tau_2;\tau_4]\!]$$

Sadly, this formulation does not properly account for functions that – unlike v_1 and v_2 – can be derived from the context without them being structural components. Consider the composition of multiple subcomponents by means of the S-distArr rule. For instance, that rule can derive from the context $\langle \lambda x : \mathsf{Bool}.\, 1, \lambda x : \mathsf{Bool}.\, \mathsf{true}\rangle$ the function $\lambda x : \mathsf{Bool}.\, \langle 1, \mathsf{true}\rangle$ which is not a syntactic component of the initial type. In NeColus, we do not have to separately account for these cases, because the components of the function body never interact and can thus be considered separately. For that reason, the logical relation of NeColus does not explicitly have to consider S-distArr. However, in λ_i^{MP} matters are different because of S-mp. When the context is $\langle \lambda x : \mathsf{Bool}.\, \lambda y : \mathsf{Bool}.\, 1, \lambda x : \mathsf{Bool}.\, \mathsf{true}\rangle$, then using S-distArr we can derive the function $\lambda x : \mathsf{Bool}.\, \langle \lambda y : \mathsf{Bool}.\, 1, \mathsf{true}\rangle$. When this function is explicitly applied to a value, we get a new original value $\langle \lambda y : \mathsf{Bool}.\, 1, \mathsf{true}\rangle$ whose two components can interact through S-mp. Thus, just checking $\lambda x : \mathsf{Bool}.\, \lambda y : \mathsf{Bool}.\, 1$ on its own is insufficient. To reckon with all ways in which the functions v_1 and v_2 under consideration can be combined by means of S-distArr with other functions derived from the context, we augment the condition as shown in the following extended condition:

$$\forall(v, v') \in \mathcal{E}^{(v:\tau_1;v':\tau_3)}[\![\tau_1;\tau_3]\!],\ \forall c \vdash \tau_1' \triangleright \tau_1 \to \tau_0,\ \forall c' \vdash \tau_2' \triangleright \tau_3 \to \tau_0',$$
$$(\langle v_1\ v, c\ v_1'\ v\rangle, \langle v_2\ v', c'\ v_2'\ v'\rangle) \in \mathcal{E}^{(\langle v_1\ v, c\ v_1'\ v\rangle:\tau_2\times\tau_0;\langle v_2\ v', c'\ v_2'\ v'\rangle:\tau_4\times\tau_0')}[\![\tau_2\times\tau_0;\tau_4\times\tau_0']\!]$$

Yet, this formulation breaks the well-foundedness of the recursive definition because $\tau_2 \times \tau_0$ and $\tau_4 \times \tau_0'$ are not necessarily smaller than $\tau_1 \to \tau_2$ and $\tau_3 \to \tau_4$, respectively. Fortunately, we can mention the additional components only in the context index to make them available for interaction through S-mp. This restores the well-foundedness of the definition.

$$\forall(v, v') \in \mathcal{E}^{(v:\tau_1;v':\tau_3)}[\![\tau_1;\tau_3]\!],\ \forall c \vdash \tau_1' \triangleright \tau_1 \to \tau_0,\ \forall c' \vdash \tau_2' \triangleright \tau_3 \to \tau_0',$$
$$(v_1\ v, v_2\ v') \in \mathcal{E}^{(\langle v_1\ v, c\ v_1'\ v\rangle:\tau_2\times\tau_0;\langle v_2\ v', c'\ v_2'\ v'\rangle:\tau_4\times\tau_0')}[\![\tau_2;\tau_4]\!]$$

There is one last twist. Rule S-distArr can still be used on two function components with different domains. In particular, we can make the domains agree by means of the contra-variant part of the S-arr rule. For instance, we can derive $\lambda x : \mathsf{Int}\times\mathsf{Bool}.\, \langle \lambda y : \mathsf{Bool}.\, 1, \mathsf{true}\rangle$ from the original value $\langle \lambda x : \mathsf{Bool}.\, \lambda y : \mathsf{Bool}.\, 1, \lambda x : \mathsf{Int}\times\mathsf{Bool}.\, \mathsf{true}\rangle$ by first strengthening the domain of the first component before distributing the product over the functions. This explains the remaining complication in the actual condition of Figure C.1.

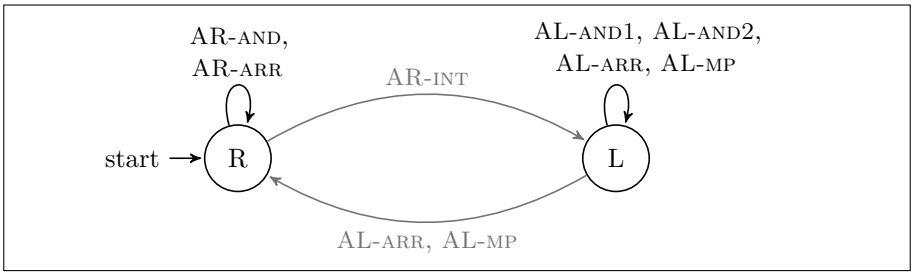


Figure C.2: Dependency graph between algorithmic subtyping modes

Top Cases Finally, the last case summarizes all combinations of $\top$ with itself and with Int, where all values are trivially related.

Terms Figure C.1 also contains the definition of the logical relation for terms, whose form is fairly standard for strongly normalising languages. The one extension is that we also normalise the original-term indices and not just the terms in the relation.

C.6 Termination of Algorithmic Subtyping

In this section, we prove that with the use of a loop detection mechanism, the subtyping algorithm presented in Figure 5.8 of Section 5.5 terminates for any input types, A and B.

The proof is partly mechanized in Coq and partly written in LaTeX. The mechanized part is presented here in the form of lemmas, while the written part is contained in the text and the remarks of this section.

C.6.1 The proof

The subtyping algorithm uses a left and a right focusing mode. Figure C.2 shows the dependency graph of the two modes. Nodes (L) and (R) stand for left- and right-focusing, respectively. The black looping edges above them represent algorithmic subtyping rules that recursively continue in the same mode, while the gray edges represent rules that trigger subtyping in the other mode. The algorithm always starts with right-focusing, which decomposes B in $\mathcal{L} \vdash_{\text{R}} A \prec: B$, and eventually enters the left-focusing phase which works by

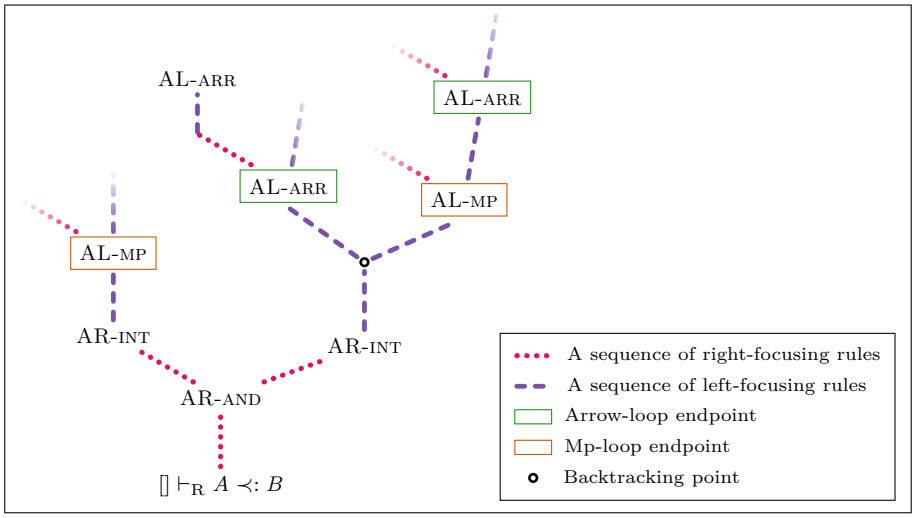


Figure C.3: Visualization of the algorithmic subtyping process

decomposing A. In left focusing, rules AL-ARR and AL-MP recursively continue in this mode, while at the same time triggering back the right-focusing mode.

In an effort to detect potential diverging behavior, we can start by examining a variation of the algorithm where the interdependencies of the focusing modes are removed (represented by the black-colored part of Figure C.2), thus isolating the two modes of the algorithm[1]. Since both modes follow the structure of their corresponding focal type, they are both terminating when considered in isolation. Adding back the left-focusing dependency in rule AR-INT still retains termination, because left-focusing (which still ignores the right-focusing calls) terminates. To conclude, the only diverging loops that could possibly occur involve mutual recursion between both focusing modes, through the rules AL-ARR and AL-MP.

Figure C.3 offers a visual aid to closely examine the shape of these loops. It shows part of an arbitrary computation tree of the algorithmic subtyping process, including backtracking branches. The algorithm starts by applying a number of right-focusing rules (represented by a dotted pink path) until it reaches an AR-INT rule application. Now it continues in left-focusing mode (represented by a dashed purple path), sometimes igniting a new right-focusing

[1] In this variation, the right-focusing premises in the left-focusing rules AL-MP and AL-ARR, and the left-focusing premise of the right-focusing rule AR-INT are stripped away. This makes the algorithm incomplete w.r.t. the declarative version, but this is not a concern here.

branch through rules AL-ARR and AL-MP, as shown in the rightmost left-focusing path of Figure C.3. In the computation tree, a path that has an initial part in right-focusing mode, its remainder in left-focusing mode and ends on a AL-ARR or AL-MP application, represents what we call a *single loop* of mutual recursion between the two focusing modes. Informally, it is a computation path of the form

$$\vdash_{\text{R}} \longrightarrow \cdots \longrightarrow \vdash_{\text{R}} \xrightarrow{\text{AR-INT}} \vdash_{\text{L}} \longrightarrow \cdots \longrightarrow \vdash_{\text{L}} \xrightarrow[\text{or AL-MP}]{\text{AL-ARR}} \vdash_{\text{R}}$$

where the last node is the right-focusing premise of an AL-ARR or an AL-MP rule. If it ends with an AL-ARR rule application, we call it an *arrow-loop*, and otherwise an *mp-loop*. For instance, in Figure C.3 all the paths that start at the root and end on an outlined AL-ARR or AL-MP rule are single loops.

It is now clearer that every right-focusing judgment to be resolved, signifies the start of a number (possibly zero) of sequences (possibly empty) of single loops. Therefore, a single loop, itself ending in a right-focusing judgment to be resolved, can be considered as the first loop in a number of sequences of single loops of the same kind (arrow- or mp-loops). Our termination proof examines the finiteness of the number of single loop sequences that the algorithm might go through, as well as the finiteness of the size of each of those sequences.

We will show that a) arrow-loops are not a source of non-termination, and b) a loop-detection mechanism that tracks previously tried mp-loops makes the algorithm terminating. For that, we define two relations, one for arrow-loops and one for mp-loops, each capturing the ways in which left-focusing triggers the right-focusing mode through an application of the AL-ARR or the AL-MP rule, respectively.

$\boxed{A \in_{\&} B}$

$$\frac{}{A \in_{\&} A}$$

$$\frac{A \in_{\&} B_1}{A \in_{\&} B_1 \,\&\, B_2}$$

$$\frac{A \in_{\&} B_2}{A \in_{\&} B_1 \,\&\, B_2}$$

Figure C.4: Definition of intersection membership

Their definition uses an auxiliary intersection-membership relation, written as $A \in_{\&} B$ and defined in Figure C.4. The formula $A \in_{\&} B$ expresses that B is the type A possibly intersected with more types.

$$B = (\ldots \& (A \& \ldots)) \& \ldots$$

We say that type A is a *component* of type B.

The results below on intersection membership are useful for the rest of the proof. The definition of the function $\|\cdot\|$, here referred to as our *norm*, is given in Subsection 5.5.2.

Lemma C.1 (Intersection membership decreases the norm)**.** *If* $A \in_{\&} B$*, then* $\|A\| \leqslant \|B\|$.

$\boxed{(A \prec\!: B) \hookrightarrow (B' \prec\!: A')}$ Single arrow-loops

$$\textsc{la-base}\ \frac{B_1 \to B_2 \in_{\&} B}{(A_1 \to A_2 \prec\!: B) \hookrightarrow (B_1 \prec\!: A_1)} \qquad \textsc{la-mp}\ \frac{(A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')}{(A_1 \to A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')}$$

$$\textsc{la-arr}\ \frac{B_1 \to B_2 \in_{\&} B \qquad (A_2 \prec\!: B_2) \hookrightarrow (B' \prec\!: A')}{(A_1 \to A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')}$$

$$\textsc{la-andL}\ \frac{(A_1 \prec\!: B) \hookrightarrow (B' \prec\!: A')}{(A_1 \,\&\, A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')} \qquad \textsc{la-andR}\ \frac{(A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')}{(A_1 \,\&\, A_2 \prec\!: B) \hookrightarrow (B' \prec\!: A')}$$

Figure C.5: Single arrow-loops, formalized

Remark 1. *Every type has a finite number of components. Formally, for a given type B, the set $\{A \mid A \in_{\&} B\}$ is finite. We also say that, given type B, the intersection membership relation produces a finite number of elements (the components of B).*

C.6.2 Arrow-loops are not a source of non-termination

The arrow-loop relation, $\hookrightarrow$, between right-focusing subtyping judgments, shown in Figure C.5, captures single loops of mutual recursion through an AL-ARR rule application. Informally, $(A \prec\!: B) \hookrightarrow (B' \prec\!: A')$ expresses that resolving $[] \vdash_{\mathrm{R}} A \prec\!: B$, will trigger resolving $[] \vdash_{\mathrm{R}} B' \prec\!: A'$ through a single arrow loop.

There are many ways in which rule AL-ARR can be reached starting from the right-focusing mode. The most immediate case is when resolving $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec\!: B_1 \to B_2$. Once rule AL-ARR is reached, we have the following two sources of single arrow-loops. Firstly, because rule AL-ARR triggers $[] \vdash_{\mathrm{R}} B_1 \prec\!: A_1$, this completes a single arrow-loop. Rule LA-BASE expresses this arrow-loop, but it is generalized to account for intersections of function types: resolving $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec\!: B$ will ignite a right-focusing call to resolve $[] \vdash_{\mathrm{R}} B_1 \prec\!: A_1$, for every arrow type component $B_1 \to B_2$ of B. Secondly, because rule AL-ARR recursively continues in left-focusing mode, this allows for more single arrow-loops through the left-focusing branch of the computational process. Rule LA-ARR expresses this second source of arrow loops, but is similarly generalized to account for intersections of function types. Rules LA-ANDL, LA-ANDR and LA-MP encode the arrow-loops that occur through an application of AL-AND1 or AL-AND2 or through the left-focusing branch of a AL-MP application, respectively.

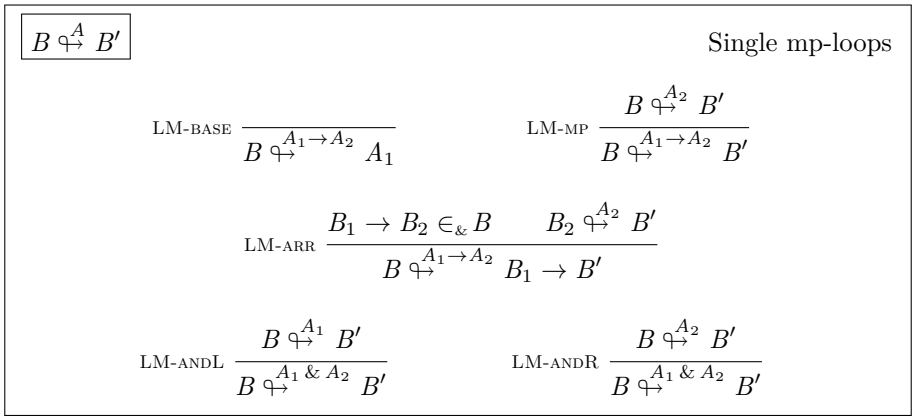


Figure C.6: Single mp-loops, formalized

In Figure C.3, $\hookrightarrow$ relates the right-focusing judgment at the root of the tree with the right-focusing premise of every AL-ARR application that is outlined with green borders. The innermost AL-ARR application is not outlined because, starting from the root, it takes more than one (in particular, it takes two) single mutual recursion loops to reach it. Such arrow-loops, nested in the right-focusing subderivation of a previously applied AL-ARR, are encoded in the transitive closure of $\hookrightarrow$.

Remark 2. *Given types A and B, because of Remark 1, rule LA-BASE produces a finite number of elements (possibly zero) in the $\hookrightarrow$-relation. Since the rest of the rules in Figure C.5 are structurally decreasing on argument A, it follows that given types A and B, the set $\{(B', A') \mid (A \prec: B) \hookrightarrow (B' \prec: A')\})\}$ is finite. Note that the left premise of rule LA-ARR contributes by a finite factor in the number of elements that this rule can produce.*

Lemma C.2 (Arrow-loops strictly decrease the norm)**.**
If $(A \prec: B) \hookrightarrow (B' \prec: A')$, then $\|[] \vdash_R A \prec: B\| < \|[] \vdash_R B' \prec: A'\|$.

It follows from the above remark and from Lemma C.2 that whenever the algorithm is in right-focusing mode, it re-enters that mode through AL-ARR a finite number of times and with a right-focusing judgment that is strictly smaller than the previous (w.r.t. our norm). This ensures that arrow-loops are not a source of non-termination.

C.6.3 Loop detection for termination of MP-loops

Figure C.6 shows the definition of the mp-loop family of relations, which, similarly to the arrow-loop relation, encodes single loops of mutual recursion through rule AL-MP. Informally, $B \nleftrightarrow^{A} B'$ expresses that resolving the judgment $[] \vdash_{\mathrm{R}} A \prec: B$, generates the call $[] \vdash_{\mathrm{R}} A \prec: B'$ through a single mp-loop.

A single right-focusing judgment may generate several mp-loops. The most immediate case, encoded by rule LM-BASE, is when resolving $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec: B$, which triggers $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec: A_1$. Subsequent applications of the AL-MP rule that occur under the left-focusing branch of a previous AL-MP rule application are encoded by rule LM-MP. Similarly, rule LM-ARR encodes applications of the AL-MP rule under the left-focusing branch of a previously applied AL-ARR rule. Lastly, rules LM-ANDL and LM-ANDR encode mp-loops that occur under an application of AL-AND1 and AL-AND2, respectively.

Note that for a type A, $\nleftrightarrow^{A}$ is a set (not a multiset) and, thus, contains unique pairs of related right-focusing judgments. Put differently, starting resolution on $[] \vdash_{\mathrm{R}} A \prec: B$, the judgment does not distinguish between distinct applications of the AL-MP rule that generate the same right-focusing call $[] \vdash_{\mathrm{R}} A \prec: B'$. For example, starting with $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec: \mathsf{Int} \,\&\, (\mathsf{Bool} \to \mathsf{Int})$, the algorithm branches by rule AR-AND and tries to apply the AL-MP rule within both branches. Both applications generate the same right-focusing call, $[] \vdash_{\mathrm{R}} A_1 \to A_2 \prec: A_1$, but are encoded as one mp-loop through rule LM-BASE. This justifies the use of an arbitrary type B in the left component of the relation $\nleftrightarrow^{A_1 \to A_2}$, in the conclusion of the rules LM-BASE and LM-ARR of Figure C.6.

Remark 3. *The rules of the mp-loop relation family are structurally decreasing on the index type A. Thus, taking into consideration Remark 1 to account for rule LM-ARR, it follows that for types A and B, the set $\{B' \mid B \nleftrightarrow^{A} B'\}$ is finite.*

Lemma C.3 (Mp-loops non-strictly decrease the norm)**.**
If $B \nleftrightarrow^{A} B'$, then $\|[] \vdash_{R} A \prec: B\| \leqslant \|[] \vdash_{R} A \prec: B'\|$.

Because a mp-loop signifies the start of a number of sequences of mp-loops to follow, Remark 3 essentially means that when the algorithm is in right-focusing mode it will generate a finite number of mp-loop sequences. However, Lemma C.3 does not guarantee that the right-focusing judgment triggered by a mp-loop is strictly smaller than the initial right-focusing judgment. As a result, although the *number* of mp-loop sequences is finite, their *length* is not necessarily finite. Yet, by Remark 3, Lemma C.3 and the pigeon hole principle, we can conclude that for an mp-loop sequence to be infinite, it must contain an infinite repetition of a finite subsequence of mp-loops. Thus, a mechanism

that detects repetitions of mp-loops is sufficient to salvage the algorithm from non-termination.

□

Bibliography

Abadi, M. (1999). "Protection in Programming-Language Translations". In: *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pp. 19–34.

Ahmed, A. (2006). "Step-indexed syntactic logical relations for recursive and quantified types". In: *European Symposium on Programming (ESOP)*.

Ahmed, A. (2015). *Logical Relations*. `https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html`.

Alpuim, J., S. Oliveira, B. C. d., & Shi, Z. (2017). "Disjoint Polymorphism". In: *ESOP*. Vol. 10201. Lecture Notes in Computer Science, pp. 1–28.

Amin, N., Grütter, K. S., Odersky, M., Rompf, T., & Stucki, S. (2016). "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science. 9600, pp. 249–272.

Amin, N., Moors, A., & Odersky, M. (2012). "Dependent Object Types". In.

Andreoli, J.-M. (1992). "Logic Programming with Focusing Proofs in Linear Logic". In: *Journal of Logic and Computation* 2.3, pp. 297–347.

Barendregt, H. (1985). *The Lambda Calculus - Its Syntax and Semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland.

Barendregt, H., Coppo, M., & Dezani-Ciancaglini, M. (1983). "A Filter Lambda Model and the Completeness of Type Assignment". In: *J. Symb. Log.* 48.4, pp. 931–940.

Barendregt, H., Dekkers, W., & Statman, R. (2013). *Lambda calculus with types*. Cambridge University Press.

Benzaken, V., Castagna, G., & Frisch, A. (2003). "CDuce: An XML-Centric General-Purpose Language". In: *SIGPLAN Not.* 38.9, pp. 51–63.

Bernardy, J.-P., Jansson, P., & Paterson, R. (2012). "Proofs for free: Parametricity for dependent types". In: *Journal of Functional Programming* 22.2, pp. 107–152.

Bessai, J., Düdder, B., Dudenhefner, A., Chen, T.-C., & de'Liguoro, U. (2014). "Typing Classes and Mixins with Intersection Types". In: *ITRS*. Vol. 177. EPTCS, pp. 79–93.

Bessai, J., Dudenhefner, A., Düdder, B., & Rehof, J. (2016). "Extracting a Formally Verified Subtyping Algorithm for Intersection Types from Ideals and Filters". In: TYPES.

Bi, X., Oliveira, B. C. d. S., & Schrijvers, T. (2018). "The Essence of Nested Composition". In: *ECOOP*.

Bi, X., Xie, N., Oliveira, B. C. d. S., & Schrijvers, T. (2019). "Distributive Disjoint Polymorphism for Compositional Programming". In: Lecture Notes in Computer Science.

Biernacki, D. & Polesiuk, P. (2015). "Logical Relations for Coherence of Effect Subtyping". In: *LIPIcs*.

Boehm, H.-J. (1985). "Partial Polymorphic Type Inference is Undecidable". In: *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pp. 339–345.

Bottu, G.-J. & Eisenberg, R. A. (2021). *Seeking Stability by being Lazy and Shallow*. arXiv: `2106.14938 [cs.PL]`.

Bottu, G.-J., Karachalias, G., Schrijvers, T., Oliveira, B. C. d. S., & Wadler, P. (2017). "Quantified Class Constraints". In: *ACM SIGPLAN Haskell Symposium*, pp. 148–161.

Bottu, G.-J. & simonpj (2018). *Quantified Constraints - Proposals*. `https://github.com/Gertjan423/ghc-proposals/blob/quantified-constraints/proposals/0000-quantified-constraints.rst#overlap`.

Bottu, G.-J., Xie, N., Marntirosian, K., & Schrijvers, T. (15, 2019). "Coherence of Type Class Resolution". Version 2. In: arXiv: `1907.00844v2 [cs.PL, 68N15, 68N18, D.3.1]`.

Bracha, G. & Cook, W. R. (1990). "Mixin-based Inheritance". In: *OOPSLA/ECOOP*, pp. 303–311.

Brady, E. (2013). "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5, pp. 552–593.

Brady, E. (2015). *Type Classes in Idris*. https://groups.google.com/forum/#!topic/idris-lang/OQQ3oc6zBaM. Idris Programming Language Mailing List.

Breazu-Tannen, V., Coquand, T., Gunter, C. A., & Scedrov, A. (1991). "Inheritance As Implicit Coercion". In: *Inf. Comput.* 93.1, pp. 172–221.

Camarão, C. & Figueiredo, L. (1999). "Type Inference for Overloading without Restrictions, Declarations or Annotations". In: *Functional and Logic Programming*, pp. 37–52.

Cardelli, L. & Wegner, P. (1985). "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Comput. Surv.* 17.4, pp. 471–523.

Castagna, G., Ghelli, G., & Longo, G. (1992). "A Calculus for Overloaded Functions with Subtyping". In: *SIGPLAN Lisp Pointers* V.1, pp. 182–192.

Chakravarty, M. M. T., Keller, G., & Peyton Jones, S. (2005). "Associated Type Synonyms". In: pp. 241–253.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S., & Marlow, S. (2005). "Associated Types with Class". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05, pp. 1–13.

Chen, G. (2003). "Coercive subtyping for the Calculus of Constructions." In: *POPL*, pp. 150–159.

Chong, S. (2017). *Lecture 2: Logical Relations Part 2*. https://www.seas.harvard.edu/courses/cs252/2017fa/lectures/lec02-contextual-equiv.pdf.

Church, A. (1940). "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2, pp. 56–68.

Coppo, M., Dezani-Ciancaglini, M., & Venneri, B. (1981). "Functional Characters of Solvable Terms". In: *Math. Log. Q.* 27, pp. 45–58.

Curien, P.-L. & Ghelli, G. (1994). "Theoretical Aspects of Object-oriented Programming". In: chap. Coherence of Subsumption, Minimum Typing and Type-checking in F&Le; pp. 247–292.

Damas, L. & Milner, R. (1982). "Principal Type-Schemes for Functional Programs". In: *POPL*, pp. 207–212.

Davies, R. & Pfenning, F. (2000). "Intersection Types and Computational Effects". In: *SIGPLAN Not.* 35.9, pp. 198–208.

Devriese, D. & Piessens, F. (2011). "On the Bright Side of Type Classes: Instance Arguments in Agda". In: *ICFP '11*, pp. 143–155.

Dijkstra, A. & Swierstra, S. D. (2005). *Making Implicit Parameters Explicit.* Tech. rep. Utrecht University.

Dreyer, D., Harper, R., Chakravarty, M. M. T., & Keller, G. (2007). "Modular type classes". In: *POPL*, pp. 63–70.

Dunfield, J. (2012). "Elaborating Intersection and Union Types". In: *SIGPLAN Not.* 47.9, pp. 17–28.

Dunfield, J. (2015). "Elaborating Evaluation-Order Polymorphism". In: *SIGPLAN Not.* 50.9, pp. 256–268.

Dunfield, J. & Krishnaswami, N. R. (2013). "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ICFP '13, pp. 429–442.

Dunfield, J. (2007). "Refined Typechecking with Stardust". In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification.* PLPV '07, pp. 21–32.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., & Weirich, S. (2014). "Closed Type Families with Overlapping Equations". In: *POPL*, pp. 671–683.

Freeman, P. (2017). *PureScript by Example.* `https://leanpub.com/purescript`. Leanpub.

Freeman, T. & Pfenning, F. (1991). "Refinement types for ML". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pp. 268–277.

Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., & Willcock, J. (2007). "An Extended Comparative Study of Language Support for Generic Programming". In: *Journal of Functional Programming* 17.2.

Gay, S. J. & Hole, M. J. (2005). "Subtyping for Session Types in the Pi Calculus". In: *Acta Informatica* 42.2/3, pp. 191–225.

GHC (2017a). *GHC Users Guide: Overlapping Instances.* `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#overlapping-instances`.

GHC (2017b). *Quantified Constraints – GHC.* `https://ghc.haskell.org/trac/ghc/wiki/QuantifiedConstraints`.

Giesl, J., Aschermann, C., Brockschmidt, M., et al. (2017). "Analyzing Program Termination and Complexity Automatically with AProVE". In: *J Autom Reasoning* 58, pp. 3–31.

Gilbert, G., Cockx, J., Sozeau, M., & Tabareau, N. (2019). "Definitional Proof-Irrelevance without K". In: *Proceedings of the ACM on Programming Languages*. POPL'19, pp. 1–28.

Gonthier, G., Ziliani, B., Nanevski, A., & Dreyer, D. (2011). "How to Make Ad Hoc Proof Automation Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11, pp. 163–175.

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., & Lumsdaine, A. (2006). "Concepts: Linguistic Support for Generic Programming in C++". In: *SIGPLAN Not.* 41.10, pp. 291–310.

Hall, C. V., Hammond, K., Peyton Jones, S., & Wadler, P. L. (1996). "Type Classes in Haskell". In: *ACM Trans. Program. Lang. Syst.* 18.2, pp. 109–138.

Harper, R. (2016). *Practical Foundations for Programming Languages*. 2nd. Cambridge University Press.

Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., & Speirs, C. (1996). *The Mercury Language Reference Manual*. Tech. rep.

Hindley, J. R. (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146, pp. 29–60.

Hulley, B. (2009). *A show-stopping problem for modular type classes?* `http://lists.seas.upenn.edu/pipermail/types-list/2009/001405.html`. Email on the Types mailing list.

Jeffery, A. (2019). "Dependent Object Types with Implicit Functions". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. Scala '19, pp. 1–11.

Jones, M. P. (1992). "A Theory of Qualified Types". In: *ESOP '92*, pp. 287–306.

Jones, M. P. (1993). *Coherence for qualified types*. Research Report YALEU/DCS/RR-989. Yale University, Dept. of Computer Science.

Jones, M. P. (1994). *Qualified Types: Theory and Practice*. Cambridge University Press.

Jones, M. P. (1995). "Simplifying and improving qualified types". In: *FPCA*.

Jones, M. P. (2000). "Type Classes with Functional Dependencies". In: *Programming Languages and Systems*. Vol. 1782. LNCS, pp. 230–244.

Kahl, W. & Scheffczyk, J. (2001). “Named Instances for Haskell Type Classes”. In: *Proc. Haskell Workshop 2001, Firenze*. Vol. 59. ENTCS 2.

Kiselyov, O. & Shan, C.-c. (2004). “Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell ’04, pp. 33–44.

Kmett, E. (2015). *Type Classes vs the World*. `https://www.youtube.com/watch?v=hIZxTQP1ifo`. YouTube video.

Kowalski, R. A. (1974). “Predicate Logic as Programming Language”. In: *IFIP Congress*, pp. 569–574.

Kurata, T. & Takahashi, M. (1995). “Decidable Properties of Intersection Type Systems”. In: *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*. TLCA ’95, pp. 297–311.

Lampropoulos, L. & Pierce, B. C. (2018). *QuickChick: Property-Based Testing in Coq*. 1st. Vol. 4. Software Foundations.

Läufer, K. & Odersky, M. (1994). “Polymorphic Type Inference and Abstract Data Types”. In: *ACM Trans. Program. Lang. Syst.* 16.5, pp. 1411–1430.

Laurent, O. (2012). “Intersection Types with Subtyping by Means of Cut Elimination”. In: *Fundam. Inf.* 121.1-4, pp. 203–226.

Le Botlan, D. & Rémy, D. (2003). “ML^{F}: Raising ML to the Power of System F”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’03, pp. 27–38.

Leijen, D. (2008). “HMF: Simple Type Inference for First-Class Polymorphism”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’08, pp. 283–294.

Lewis, J. R., Launchbury, J., Meijer, E., & Shields, M. B. (2000). “Implicit Parameters: Dynamic Scoping with Static Types”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’00, pp. 108–118.

Liang, C. & Miller, D. (2009). “Focusing and Polarization in Linear, Intuitionistic, and Classical Logics”. In: *Theoretical Computer Science* 410.46, pp. 4747–4768.

Luo, Z. (1999). “Coercive Subtyping”. In: *Journal of Logic and Computation* 9.1, pp. 105–130.

Manna, Z. & Waldinger, R. (1980). “A Deductive Approach to Program Synthesis”. In: *ACM Trans. Program. Lang. Syst.* 2.1, pp. 90–121.

Marlow, S. (2012). *Safe Haskell and instance coherence*. https://mail.haskell.org/pipermail/haskell-cafe/2012-October/103887.html.

Martelli, A. & Montanari, U. (1982). "An Efficient Unification Algorithm". In: *ACM Trans. Program. Lang. Syst.* 4.2, pp. 258–282.

Miller, D. (1992). "Unification Under a Mixed Prefix". In: *J. Symb. Comput.* 14.4, pp. 321–358.

Miller, D., Nadathur, G., Pfenning, F., & Scedrov, A. (1991). "Uniform Proofs as a Foundation for Logic Programming". In: *Annals of Pure and Applied Logic* 51.1, pp. 125–157.

Milner, R. (1978). "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3, pp. 348–375.

Moggi, E. (1991). "Notions of Computation and Monads". In: *Inf. Comput.* 93.1, pp. 55–92.

Morris, J. G. (2014). "A simple semantics for Haskell overloading". In: *Haskell 2014*, pp. 107–118.

Morris, J. G. & Jones, M. P. (2010). "Instance Chains: Type Class Programming without Overlapping Instances". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10, pp. 375–386.

Morris Jr, J. H. (1969). "Lambda-calculus models of programming languages." PhD thesis. Massachusetts Institute of Technology.

Moura, L. de, Kong, S., Avigad, J., Van Doorn, F., & Raumer, J. von (2015). "The Lean theorem prover". In.

Mozilla Research, T. (2017). *The Rust Programming Language*. https://www.rust-lang.org/en-US/.

Norell, U. (2009). "Dependently Typed Programming in Agda". In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, pp. 230–266.

nponeccop (2012). *Breaking Data.Set Integrity without GeneralizedNewtypeDeriving*. https://stackoverflow.com/questions/12735274/breaking-data-set-integrity-without-generalizednewtypederiving.

Odersky, M. (2010). *The Scala Language Specification, Version 2.8*.

Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., & Stucki, S. (2017). "Simplicitly: Foundations and Applications of Implicit Function Types". In: *Proc. ACM Program. Lang.* 2.POPL.

Odersky, M. & Läufer, K. (1996). "Putting Type Annotations to Work". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96, pp. 54–67.

Odersky, M., Spoon, L., & Venners, B. (2008). "Implicit Conversions and Parameters". In: *Programming in Scala*. Chap. 21.

oleg (2006). *Typeclass vs. Prolog programming*. `https://mail.haskell.org/pipermail/haskell-cafe/2006-September/018500.html`.

Oliveira, B. C. d. S., Schrijvers, T., Choi, W., Lee, W., & Yi, K. (2012). "The Implicit Calculus: A New Foundation for Generic Programming". In.

Oliveira, B. C. d. S., Moors, A., & Odersky, M. (2010). "Type Classes As Objects and Implicits". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10, pp. 341–360.

Oliveira, B. C. d. S., Shi, Z., & Alpuim, J. (2016). "Disjoint Intersection Types". In: *SIGPLAN Not.* 51.9, pp. 364–377.

Orchard, D. & Schrijvers, T. (2010). "Haskell Type Constraints Unleashed". In: *Functional and Logic Programming*, pp. 56–71.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming. Cambridge University Press.

Peyton Jones, S., Jones, M. P., & Meijer, E. (1997). "Type classes: an exploration of the design space". In: *Proceedings of the 1997 Haskell Workshop*.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Shields, M. (2007). "Practical Type Inference for Arbitrary-Rank Types". In: *Journal of functional programming* 17.1, pp. 1–82.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Washburn, G. (2006). "Simple Unification-based Type Inference for GADTs". In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06, pp. 50–61.

Pfenning, F. (1993). "On the Undecidability of Partial Polymorphic Type Reconstruction". In: *Fundam. Informaticae* 19.1/2, pp. 185–199.

Pierce, B. C. & Sangiorgi, D. (1996). "Typing and Subtyping for Mobile Processes". In: *Math. Structures in Comput. Sci.* 6.5, pp. 409–453.

Pierce, B. C. (1989). *A Decision Procedure for the Subtype Relation on Intersection Types with Bounded Variables*. Tech. rep. Carnegie Mellon University.

Pierce, B. C. (1991). "Programming with Intersection Types and Bounded Polymorphism". Available as School of Computer Science technical report CMU-CS-91-205. PhD thesis. Carnegie Mellon University.

Pierce, B. C. (1997). "Intersection types and bounded polymorphism". In: *Mathematical Structures in Computer Science* 7.2, pp. 129–193.

Pierce, B. C. (2002). *Types and Programming Languages*. 1st. MIT Press.

Pierce, B. C. & Turner, D. N. (2000). "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1, pp. 1–44.

Plotkin, G. D. (1973). *Lambda-definability and logical relations*. Edinburgh University.

Plotkin, G. D. (1981). *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19.

Pottinger, G. (1980). "A type assignment for the strongly normalizable λ-terms". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pp. 561–577.

rHaskell (2006). *The constraint trick for instances*. `https://www.reddit.com/r/haskell/comments/3afi3t/the_constraint_trick_for_instances/cscb33j/?st=jjcz7zeh&sh=1e5f3c8b`.

Raskin, J. (1974). "FLOW: a teaching language for computer programming in the humanities". In: *Computers and the Humanities* 8.4, pp. 231–237.

Rehof, J. & Urzyczyn, P. (2011). "Finite Combinatory Logic with Intersection Types". In: *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications*. TLCA'11, pp. 169–183.

Reynolds, J. C. (1972). "Definitional Interpreters for Higher-order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72, pp. 717–740.

Reynolds, J. C. (1974). "Towards a Theory of Type Structure". In: vol. 19. Lecture Notes in Computer Science, pp. 408–425.

Reynolds, J. C. (1988). *Preliminary Design of the Programming Language Forsythe*. Tech. rep. Carnegie Mellon University.

Reynolds, J. C. (1991). "The Coherence of Languages with Intersection Types". In: *Proceedings of the International Conference on Theoretical Aspects of Computer Software*. TACS '91, pp. 675–700.

Rouvoet, A. (2016). "Programs for Free: Towards the Formalization of Implicit Resolution in Scala". MA thesis. TU Delft.

Russell, B. (1908). “Mathematical Logic as Based on the Theory of Types”. In: *American Journal of Mathematics* 30.3, pp. 222–262.

Scharli, N., Ducasse, S., Nierstrasz, O., & Black, A. P. (2003). “Traits: Composable Units of Behaviour”. In: *European Conference on Object-Oriented Programming (ECOOP 2003)*, pp. 248–274.

Schrijvers, T., Peyton Jones, S., Chakravarty, M. M. T., & Sulzmann, M. (2008). “Type checking with open type functions”. In: *ICFP*.

Schrijvers, T., Oliveira, B. C. d. S., Wadler, P., & Marntirosian, K. (2019). “COCHIS: Stable and coherent implicits”. In: *Journal of Functional Programming* 29, e3.

Schwinghammer, J. (2009). “Coherence of Subsumption for Monadic Types”. In: *J. Funct. Program.* 19.2, pp. 157–172.

Siek, J. G. (2011). *The C++0x “Concepts” Effort*. `http://ecee.colorado.edu/~siek/concepts_effort.pdf`.

Siek, J. G. & Lumsdaine, A. (2005). “Essential Language Support for Generic Programming”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, pp. 73–84.

Sozeau, M. & Oury, N. (2008). “First-Class Type Classes”. In: *TPHOLs*.

Statman, R. (1985). “Logical relations and the typed λ-calculus”. In: *Information and Control* 65.2-3, pp. 85–97.

Statman, R. (2014). “A Finite Model Property for Intersection Types”. In: *ITRS*. Vol. 177. EPTCS, pp. 1–9.

Sulzmann, M., Duck, G. J., Peyton Jones, S., & Stuckey, P. J. (2007). “Understanding functional dependencies via Constraint Handling Rules”. In: *Journal of Functional Programming* 17.1, pp. 83–129.

Tait, W. W. (1967). “Intensional interpretations of functionals of finite type I”. In: *The journal of symbolic logic* 32.2, pp. 198–212.

Vytiniotis, D., Peyton Jones, S., Schrijvers, T., & Sulzmann, M. (2011). “OutsideIn(X): Modular Type Inference with Local Assumptions”. In: *Journal of Functional Programming* 21.4-5, pp. 333–412.

Vytiniotis, D., Weirich, S., & Peyton Jones, S. (2008). “FPH: First-Class Polymorphism for Haskell”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’08, pp. 295–306.

Wadler, P. & Blott, S. (1989). “How to make ad-hoc polymorphism less ad hoc”. In: *POPL*.

Wadler, P. (2015). “Propositions as Types”. In: *Commun. ACM* 58.12, pp. 75–84.

Wehr, S., Lämmel, R., & Thiemann, P. (2007). “JavaGI: Generalized Interfaces for Java”. In: *ECOOP 2007 – Object-Oriented Programming*, pp. 347–372.

White, L., Bour, F., & Yallop, J. (2015). “Modular implicits”. In: vol. 198.

Winant, T. & Devriese, D. (2018). “Coherent Explicit Dictionary Application for Haskell”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018, pp. 81–93.

Wright, A. K. & Felleisen, M. (1992). “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115, pp. 38–94.

Zhang, E. (2014). *Type Classes: Confluence, Coherence and Global Uniqueness*. `http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/`. Blog.

Zhang, Y., Loring, M. C., Salvaneschi, G., Liskov, B., & Myers, A. C. (2015). “Lightweight, Flexible Object-oriented Generics”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15.

List of publications

Input file chapters/publications/publications.tex does not exist. Make sure its starts with "\chapter{List of publications}". To not include this chapter in the table of contents, use the starred version of the \chapter command...



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
koar.marntirosian@kuleuven.be
http://dtai.cs.kuleuven.be