KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Fully Grounding Inference and Scoped Effects

Roger Bosman

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

Supervisor:
Prof. dr. ir. Tom Schrijvers

December 2024

FULLY GROUNDING INFERENCE AND SCOPED EFFECTS

Roger BOSMAN

Supervisor:
Prof. dr. ir. Tom Schrijvers

Members of the
Examination Committee:
Prof. dr. ir. Joris De Schutter, chair
Prof. dr. Bart Jacobs
Prof. dr. Marc Denecker
doc. dr. Matija Pretnar
(Univerza v Ljubljani)
dr. Ningning Xie
(University of Toronto)

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

December 2024

© 2024 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Roger Bosman, 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

I want to take the time to thank some of the people that helped me along the way towards writing this thesis.

Firstly I would like to thank my supervisor Tom Schrijvers for his advice, support and patience. His expertise and encouragement have been pivotal in helping me navigate the challenges of my PhD.

I would like to thank Bart Jacobs, Marc Denecker, Matija Pretnar and Ningning Xie for taking part in my examination committee. Thanks to Steven Keuchel for helping me with my Coq code. Thanks to Wouter Swierstra and Richard Eisenberg for introducing me to the world of PL research, and helping me obtain this PhD position. Thanks to my office mates, Birthe van den Berg, Pieter Delobelle, Jesse Hoobergs, Hanne Verstraeten, Ruifeng Xie, Tania Ugarte and Rinat Stryungis.

I would like to thank the members and especially the instructors of the sport clubs LUAC and LUK, for giving me a social net after moving to another country during covid and for allowing me to recharge mentally outside of work, and for the KU Leuven for sponsoring these clubs.

Thanks to my friends, my father Anton and my brother Guido, for supporting me trough the ups and downs. Thanks to Donny, for being an excellent rubber duck stand-in. Finally I thank my mother Els. I miss you.

Abstract

Formal methods allow for the modelling of systems to ensure that a system behaves as expected. *Type systems* (or when accompanied with a reduction: *calculi*) are the particular formal method at the center of this thesis. In particular, in this thesis we utilize calculi to formally reason about two areas.

Part I: Type Inference & Elaboration

In the first part we utilize calculi to model type inference and elaboration for the Hindley-Damas-Milner (HDM) system, one of the first and most well known type systems of its kind. Type *inference* algorithms for the HDM system, most notably algorithm $\mathcal{W}$, has received a lot of attention, and their soundness and completeness have been well-studied and proven both manually (on paper) and mechanically (in a proof assistant). However, *elaboration algorithms* (i.e. type checking algorithms that, in addition to verifying if the input program is well-formed, output a translation of the input program into an often explicitly typed output language such as System F) have not received the same amount of attention, and their implementation in compilers such as GHC is often much more ad-hoc. One key difference between these two algorithms is the treatment of underconstrained types. For type checking, unification variables that do not affect the overall type need not be instantiated. However, for elaboration, instantiating all unification variables is essential, because unification variables are internal to the algorithm and should not leak into the output.

In Part I we put elaboration algorithms on a formal footing by presenting fully formalized elaboration algorithm for the HDM system. We do so in two steps. First, we formalize a *fully grounding* type checking algorithm, and present its (mechanically verified) correctness proof. The novel concept of fully grounding encompasses the notion that at any point any unification variable can be instantiated, which is essential for a type checking algorithm to be extended with elaboration.

Second, we perform exactly this extension by adding elaboration to the inference algorithm to obtain an elaboration algorithm. We extend the correctness theorems, which requires an equivalence relation on elaborated terms, which we define by using logical relations. Both algorithms implement input and output contexts, as well as the novel concept of *full contexts*, which significantly simplifies the state-passing of traditional algorithms.

Part II: Scoped Effects

In the second part, we utilize calculi to reason about effects. Algebraic effects & handlers have become a standard approach for side-effects in functional programming. Their modular composition between different effects and clean separation of syntax and semantics make them attractive to a wide audience. However, not all effects can be classified as algebraic; some effects need a more sophisticated handling. In particular, effects that have or create a delimited scope need special care, as their continuation consists of two parts—in and out of the scope—and their modular composition introduces additional complexity. These effects are called *scoped* and have gained attention by their growing applicability and adoption in popular libraries. While calculi have been designed with algebraic effects & handlers built in to facilitate their use, a calculus that supports scoped effects & handlers in a similar manner does not yet exist. Here we fill this gap: we present λ_{sc}, a calculus with native support for both algebraic and scoped effects & handlers. The most novel part of λ_{sc} is the support for modular composition of different scoped effects & handlers by extending handlers with forwarding clauses, which make λ_{sc} much more expressive than existing calculi with algebraic effects & handlers. Our calculus is based on Eff, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics, a (type-safe) type-and-effect system. We give a prototype implementation of λ_{sc} with type inference and demonstrate λ_{sc} on a range of examples.

Beknopte samenvatting

Formele methoden (*formal methods*) maken het mogelijk om systemen te modelleren te verifiëren dat een systeem zich gedraagt zoals verwacht. *Typesystemen* (of wanneer deze ook evaluatieregels bevat: *calculi*) zijn de specifieke formele methode die centraal staat in deze thesis. We gebruiken in deze thesis calculi om formeel te redeneren over twee gebieden.

Deel I: Type Inferentie & Elaboratie

In het eerste deel gebruiken we calculi om type inferentie en elaboratie te modelleren voor het Hindley-Damas-Milner (HDM) systeem, een van de eerste en meest bekende typesystemen in zijn soort. Type*inferentie* algoritmen voor het HDM-systeem, met name algorithm $\mathcal{W}$, hebben veel aandacht gekregen, en de correctheid (*soundness*) en volledigheid (*completeness*) ervan zijn uitvoerig bestudeerd en bewezen, zowel handmatig (op papier) als mechanisch (in een interactieve stellingbewijzer zoals Coq). Echter, *elaboratie*algoritmen (d.w.z. type checking algoritmen die, naast het verifiëren of het invoerprogramma goed gevormd is, een vertaling van het invoerprogramma naar een vaak expliciet getypeerde uitvoertaal zoals System F geven) hebben niet dezelfde hoeveelheid aandacht gekregen, en hun implementatie in compilers zoals GHC is vaak veel meer ad-hoc. Een belangrijk verschil tussen deze twee algoritmen is de behandeling van typen met delen waarvan bepaalde delen niet gebruikt worden (typen die *underconstrained* zijn). Voor type checking hoeven unificatievariabelen die de algemene type niet beïnvloeden, niet te worden geïnstantieerd. Voor elaboratie is het echter essentieel om alle unificatievariabelen te instantiëren, omdat unificatievariabelen alleen intern in het algoritme mogen bestaan, en niet via de uitvoer naar buiten mogen lekken.

In deel part I plaatsen we elaboratiealgoritmen op een formele basis door volledig geformaliseerde elaboratiealgoritme voor het HDM-systeem te presenteren. We doen dit in twee stappen. Ten eerste formaliseren we een *volledig grondend*

(*fully grounding*) type checking algoritme, en presenteren het (mechanisch geverifieerde) correctheidsbewijs. Het nieuwe concept van *volledig grondend* omvat het idee dat op elk moment elke unificatievariabele kan worden geïnstantieerd, wat essentieel is voor een type checking algoritme om te worden uitgebreid met elaboratie.

Ten tweede voeren we deze uitbreiding uit, door elaboratie aan het inferentie-algoritme toe te voegen. We breiden de correctheidsstellingen uit, wat een equivalentierelatie op geëlaboreerde termen vereist, die we definiëren met behulp van logische relaties (*logical relations*). Beide algoritmen implementeren input- en outputcontexten, evenals het nieuwe concept van *volle contexten*, wat het doorgeven van de huidige omgeving vergeleken met traditionele algoritmen aanzienlijk vereenvoudigt.

Part II: Scoped Effecten

In het tweede deel gebruiken we calculi om over effecten te redeneren. Algebraïsche effecten & *handlers* (afhandelaars) zijn een standaardtechniek geworden voor het beheren en traceren van neveneffecten in functioneel programmeren. De modulaire samenstelling van verschillende effecten en duidelijke scheiding van syntaxis en semantiek maken ze aantrekkelijk voor een breed publiek. Echter, niet alle effecten kunnen als algebraïsch worden geclassificeerd; sommige effecten hebben een meer geavanceerde behandeling nodig. In het bijzonder moeten effecten die een scope hebben of afbakenen op een andere manier verwerkt worden, aangezien hun continuatie uit twee delen bestaat—binnen en buiten de scope—en hun modulaire samenstelling extra complexiteit met zich meebrengt. Deze effecten worden *scoped* genoemd en hebben aandacht gekregen door hun groeiende toepasbaarheid en adoptie in populaire bibliotheken. Hoewel er al calculi zijn ontworpen met ingebouwde algebraïsche effecten & handlers bestaat er nog geen calculus die scoped effecten & handlers op een vergelijkbare manier ondersteunt. Hier vullen we dit gat: we presenteren λ_{sc}, een calculus met ingebouwde ondersteuning voor zowel algebraïsche als scoped effecten & handlers. Nieuw in λ_{sc} is de ondersteuning voor modulaire samenstelling van verschillende scoped effecten & handlers door handlers uit te breiden met *forwarding* clausules, wat λ_{sc} veel expressiever maakt dan bestaande calculi met algebraïsche effecten & handlers. Onze calculus is gebaseerd op Eff, een bestaande calculus voor algebraïsche effecten, uitgebreid met rij-polymorfisme in de stijl van Koka, en bestaat uit een formele grammatica, operationele semantiek, een (*type-safe*) type-en-effect systeem. We geven een prototype implementatie van λ_{sc} met type inferentie en demonstreren λ_{sc} aan de hand van een reeks voorbeelden.

Contents

Abstract iii

Beknopte samenvatting v

Contents vii

List of Figures xi

1 Introduction 1
- 1.1 Part I: Type Inference & Elaboration . . . 2
- 1.2 Part II: Scoped effects . . . 4
- 1.3 Scientific output . . . 5

I Type Inference & Elaboration 7

2 Fully Grounding HDM Type Inference 8
- 2.1 Introduction . . . 8
- 2.2 Overview . . . 10
- 2.3 Declarative System . . . 13
  - 2.3.1 Syntax . . . 13
  - 2.3.2 Typing . . . 13
- 2.4 Algorithmic System . . . 14
  - 2.4.1 Syntax . . . 14
  - 2.4.2 Inference algorithm . . . 16
  - 2.4.3 Unification . . . 18
- 2.5 Metatheory . . . 20
  - 2.5.1 Context instantiation . . . 20
  - 2.5.2 Soundness . . . 21
  - 2.5.3 Completeness . . . 22
  - 2.5.4 Decidability . . . 22

2.6 Mechanization . . . 23
2.6.1 Ott . . . 24
2.6.2 The locally nameless representation . . . 24
2.6.3 Quirks of the locally nameless representation . . . 26
2.6.4 Delta between the chapter and the mechanization . . . 28
2.7 Related Work . . . 29
2.8 Conclusion & Next Step . . . 30
2.9 Scientific output . . . 31

3 Elaboration **33**
3.1 Introduction . . . 33
3.2 Syntax & Shared Judgments . . . 34
3.2.1 Syntax . . . 34
3.2.2 Well-formedness . . . 35
3.3 Target system . . . 36
3.4 Declarative specification . . . 37
3.4.1 Adapted declarative system . . . 39
3.5 Elaboration algorithm . . . 41
3.5.1 Unification . . . 43
3.6 Context instantiation . . . 44
3.7 Term equivalence . . . 46
3.7.1 Logical relations . . . 48
3.7.2 Logical Equivalence . . . 50
3.8 Correctness . . . 51
3.8.1 Soundness . . . 52
3.8.2 Completeness . . . 52
3.9 Implementation . . . 53
3.9.1 Locally nameless & the Ott/LNgen stack . . . 53
3.9.2 Implementation of the logical relation in Coq . . . 53
3.9.3 Soundness of the logical relation & Filling holes . . . 56
3.10 Conclusion & Future Work . . . 58

II Scoped Effects **61**

4 A Calculus For Scoped Effects & Handlers **62**
4.1 Introduction . . . 62
4.2 Background & Motivation . . . 64
4.2.1 Algebraic effects & handlers . . . 64
4.2.2 Scoped effects . . . 68
4.3 Design Decisions & Challenges . . . 70
4.3.1 Scoped Effects as Built-in Operations . . . 70
4.3.2 Eff with Row Typing . . . 71

4.3.3 Scoped result type . . . 71
4.4 Term Syntax . . . 75
4.4.1 Computations . . . 76
4.4.2 Values . . . 76
4.5 Operational Semantics . . . 77
4.6 Type-and-Effect System . . . 80
4.6.1 Grammar . . . 80
4.6.2 Value typing . . . 81
4.6.3 Computation typing . . . 82
4.6.4 Handler typing . . . 83
4.7 Generalized Forwarding . . . 85
4.7.1 *Inc*, revisited . . . 85
4.7.2 Adding generalized forwarding clauses . . . 86
4.7.3 Syntax & typing . . . 88
4.7.4 h_{inc}, revisited again . . . 89
4.8 Metatheory . . . 89
4.8.1 Syntax-directed version of λ_{sc} . . . 89
4.8.2 Safety . . . 91
4.9 Examples & Implementation . . . 92
4.9.1 Exceptions . . . 92
4.9.2 Reader with Local . . . 95
4.9.3 Nondeterminism with Cut . . . 97
4.9.4 Depth-Bounded Search . . . 98
4.9.5 Parsers . . . 100
4.10 Related Work . . . 102
4.10.1 Algebraic Effects & Handlers . . . 102
4.10.2 Effect Systems . . . 102
4.10.3 Scoped Effects & Handlers . . . 103
4.11 Conclusion and Future Work . . . 104
4.12 Scientific output . . . 105

Conclusion **106**

5 Conclusion **107**
5.1 Summary of contributions . . . 107
5.2 Future work . . . 109

A Semantic Derivations **113**
A.1 Nondeterminism . . . 113
A.2 Increment . . . 114
A.3 Once . . . 115

B Type Equivalence Rules **117**

C Well-scopedness Rules **119**

D Syntax-directed version of λ_{sc} **121**

E Metatheory **125**
E.1 Lemmas 125
E.2 Subject reduction 126
E.3 Progress 129

Bibliography **131**

List of Figures

2.1 Application and generalisation of algorithm $\mathcal{W}$ (2.1a, top) and algorithm $\mathcal{R}$ (2.1b, bottom). . . . 11
2.2 Syntax of the Declarative Specification . . . 13
2.3 Typing of the Declarative Specification . . . 15
2.4 Syntax of Algorithm $\mathcal{R}$ . . . 16
2.5 Typing of Algorithm $\mathcal{R}$ . . . 17
2.6 Unification Algorithm . . . 19
2.7 Context instantiation . . . 20

3.1 Shared Syntax . . . 35
3.2 Well-formedness of types, terms, and contexts . . . 36
3.3 Target system . . . 37
3.4 Typing of the Declarative Specification . . . 38
3.5 Typing of the Declarative Specification with A . . . 40
3.6 Typing of the algorithmic system . . . 42
3.7 Unification Algorithm . . . 43
3.8 Context instantiation . . . 44
3.9 Three equivalent derivations . . . 47
3.10 Logical relations . . . 48
3.11 Hole typing . . . 51

4.1 Terms of λ_{sc}. . . . 75
4.2 Operational semantics of λ_{sc}. . . . 78
4.3 Types of λ_{sc}. . . . 80
4.4 Value typing. . . . 81
4.5 Computation typing. . . . 82
4.6 Handler typing. . . . 84
4.7 `once` escaping its scope when forwarded by **bind** . . . 87
4.8 `once` staying in its scope forwarded by **fwd** . . . 90
4.9 Visual representation of c_{depth}. . . . 100

B.1 Type equivalence of λ_{sc}. 117
B.2 Row equivalence of λ_{sc}. 118

C.1 Well-scopedness rules of λ_{sc}. 119

D.1 Syntax-directed value typing. 121
D.2 Syntax-directed computation typing. 122
D.3 Syntax-directed handler typing. 123
D.4 σ-instantiation. 123
D.5 Handler generalisation and instantiation 124

Chapter 1

Introduction

Programming languages with static type checking claim that they can identify many problematic program behaviours and reject the programs that exhibit them at compile time. Type systems and calculi are prominent applications of formal methods to statically typed programming languages that establish the trustworthiness of these claims.

A calculus, consisting of a formally specified grammar, typing rules and semantics, can be used as a simplified model of a fully fledged programming language. Doing so allows a calculus to serve as a precise documentation of the programming language's core aspects and enables the application of type theory to reliably establish, through formal proofs, what guarantees the type system of the calculus provides. The idea is that, if the calculus is a sufficiently faithful model of the programming language, the guarantees should carry over to the actual programming language.

In this thesis, we employ the methodology of developing a calculus with a static type system and establishing its formal guarantees for two purposes:

- In Part I (Chapters 2 and 3), we formalise a common, but informal practice in elaboration-based language implementations, namely the full grounding of types during type inference, and formally establish its guarantees.
- In Part II (Chapter 4), we design a formal calculus for a new language feature called scoped effects, which so far has only been studied in the realm of denotational semantics, in order to explore the language design space and prepare for fully fledged programming language implementations.

1.1 Part I: Type Inference & Elaboration

A mayor benefit of strongly statically typed languages (a common characteristic of functional languages) is that they often relax the amount of explicit type information that must be supplied by the programmer by deriving the type instead. This concept is known as *type inference* or type reconstruction. The particular system in focus of this part is the Hindley-Damas-Milner (HDM) system. The HDM system is one of the first and most well-known type systems of its kind. It was first described by Hindley [34] and later by Milner [50], who also introduced ML-style let polymorphism. This kind of polymorphism, where only the type of let-bound expressions is generalized, has proven to hit the sweet spot between simplicity and expressivity.

The classic type inference algorithm for the HDM system, algorithm $\mathcal{W}$, as described by Damas and Milner [18], is used as basis for many type inference algorithms. Such type inference algorithms have gotten a lot attention. Properties such as the correctness have been proven both on paper [17, 28] and in proof assistants [21, 53, 25]. Typically, these proofs involve defining a declarative specification, against which the algorithm's correctness is proven. Most often these correctness proofs take shape as the composition of soundness, which states that all results of the inference algorithm are valid in the specification, and completeness, which states that if some input is typable in the specification, the algorithm infers a type at least as general.

The type inference problem described above is a binary check to verify whether an input program is well-typed. In an extension of this problem, called the *elaboration* problem, the objective is not only to verify the input is well-typed, but also, if it *is*, return the input program translated into another language. Technically, any compiler can be seen as an elaboration algorithm, with machine code as its *elaboration target* (i.e. the language to which it translates the input). However, typically and especially in context of programming languages research, the elaboration target is an explicitly typed variant of the input language. For the HDM system this typically and in this thesis is System F. Another example is GHC, which first elaborates the Haskell language to a variant of System F called GHC core, which is then processed further.

The benefit of elaboration algorithms in context of compilers is two-fold. Firstly, most work on the internals of (functional) compilers [85, 92, 23, 70, 87] reasons about a simple explicitly typed calculus instead of a full-blown programming language, because it is easier to formulate and reason about this functionality in a reduced setting. To utilize this functionality it is therefore required to first elaborate to a simpler, explicitly typed language which can then be further processed by algorithms of which there is a high level of confidence of its

correctness because they were developed in a reduced setting.

Secondly, elaboration allows for the implementation of functionality without requiring explicit downstream support for this functionality. The canonical example for this is typeclasses [90], a form of ad-hoc polymorphism present in languages such as Haskell. Since typeclasses can be expressed in terms of dictionaries, if the downstream elaboration target already supports dictionaries, elaboration algorithms allow for the addition of typeclasses to the surface level programming language without altering the downsteam elaboration target by converting typeclasses in the input program into dictionaries in the elaboration target. Other examples are implicits [68, 54, 20] and intersection types [16].

It is therefore surprising that elaboration algorithms have not received the same level of attention that inference algorithms have. In particular, the main contribution of Part I (a formalized elaboration algorithm for the HDM system) seems to be the first of its kind. We arrive at this algorithm in two steps.

Firstly, we observe that there is one key difference between inference and elaboration algorithms: the manner in which they treat derivations that contain *underconstrained* types, which means there exists types of which parts "do not matter" in the sense that the overall derivation's type is unaffected by these parts. Type checking algorithms can (and do) essentially ignore these parts, since the well-typedness of the program does not depend on these parts. For elaboration however, these shortcuts cannot be taken, because even though some parts of the program may not be relevant for the end result, they must still be elaborated, which means their type must be known.

In Chapter 2 we introduce the novel concept of a *fully grounding* type inference algorithm. In short, it is an inference algorithm that does not take any of the shortcuts related to underconstrained types. We mechanically prove it sound and complete w.r.t. a declarative specification.

Secondly, the inference algorithm being fully grounding means that unlike algorithms like algorithm $\mathcal{W}$ our algorithm of Chapter 2 *is* suitable for extension with elaboration. In Chapter 3 we perform exactly this extension: we extend (a simplified version of) the fully grounding type inference algorithm elaboration. We extend the declarative specification with elaboration as well, and formalize the elaboration target, and sketch how to prove the elaboration algorithm correct w.r.t. this extended specification.

1.2 Part II: Scoped effects

Pure functional programming bases itself on mathematical functions. Mathematical functions are fully characterised by their input-output behaviour. We say they are referentially transparent. This means that two alternative function implementations with the same input-output behaviour are indistinguishable. This means that functions cannot have any computational *side effects* besides always producing the same output for the same input. Yet in a practical computational setting, side effects are essential: a program must interact with its environment, to request inputs from the user, to display results, etc. The solution to this conundrum is to have functions return enriched outputs, which consist of their usual output embedded in a context that reflects the effects that occur while computing this result. In other words, side effects are promoted to an explicit part of a function's result and called (just) effects.

Monads have been a popular approach for modelling such explicitly effectful functions since the late 1980's, early 1990's. Moggi [51] first borrowed this abstract concept from category theory to elegantly model the operational semantics of programming languages. Then, in quick succession, Spivey [75] and Wadler [88] realised the practical usefulness for actual functional programming. This resulted in the adoption by the purely functional programming language Haskell as the principal approach to side effects, which led to a much wider visibility of the approach. Soon, a key modularity problem was found: monads (and thus their effects) do not compose. In the middle of the 1990s, this was resolved by Liang et al. [45] with the introduction of monad transformers, a mechanism for constructing a composite monad out of parts that each contribute some effects. Monad and monad transformers are today still the main mechanism for effectful programming in Haskell, although they see only limited adoption elsewhere.

In the early 2000s, Plotkin and Power [59] investigated algebraic effects as an alternative theoretical model for effects that focused less on the denotational representation of these effects, and more on their "API". The shift from theory to practice was enabled in 2009 when Plotkin and Pretnar [58] introduced the concept of handlers, which provide an implementation mechanism for the effect API. This led to the first programming language design, Eff, by Pretnar and Bauer in 2012 [6], and opened the floodgates for many other researchers. Today it is the most widely researched approach to programming with explicit effects, which an active research community and many prototypes [66, 38, 30, 42, 12]. A landmark result is perhaps the adoption of algebraic effects and handlers in the recent OCaml 5 release [1]. We believe there is still much potential for a more widespread adoption, surpassing that of monads. Three key advantages

we see are that: modularity comes out of the box, and does not require a separate concept like monad transformers; the concept is more approachable for programmers already familiar with exception handling; and the mechanism integrates seamlessly with the existing sequential execution order of call-by-value programming languages (whereas monads offer a second, possibly conflicting sequentialisation mechanism).

An issue with algebraic effects is that not all effects are algebraic. Plotkin et al. [59] realised this early on, with the `catch` operation of exception handling as an example. They presented handlers as a mechanism for encoding such non-algebraic operations (alongside with using handlers for giving meaning to the algebraic effects themselves). Yet, as Wu et al. [93] have pointed out, this second-class status of non-algebraic operations is not only unsatisfying, but also leads to modularity issues. An ensuing line of work [93, 56, 98] has investigated a generalisation of algebraic effects, called scoped effects, that addresses this shortcoming and promotes non-algebraic operations to the same level and degree of modularity as algebraic effects. Yet, the existing work on scoped effects has mostly focused on the denotational semantics of this mechanism. As a consequence, the feature is not as widely understood and adopted as algebraic effects themselves. It is now well-established in the Haskell ecosystem, where it is natural to programming directly in terms of the denotational semantics, and several libraries directly support the feature (including GitHub's fused-effects library [66]). Yet, outside of Haskell, the bare denotational semantics are a lot less suited for practical programming.

In Part II want to address the current situation by exploring a proper language design for scoped effects, that makes it more accessible and suited for adoption as a native feature in programming languages. We do so by presenting λ_{sc}, a calculus with native support for both algebraic and scoped effects & handlers. The most novel part of λ_{sc} is the support for modular composition of different scoped effects & handlers by extending handlers with forwarding clauses, which make λ_{sc} much more expressive than existing calculi with algebraic effects & handlers. Our calculus is based on Eff, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics, a (type-safe) type-and-effect system. We give a prototype implementation of λ_{sc} with type inference and demonstrate λ_{sc} on a range of examples.

1.3 Scientific output

The scientific output is specified in the last section of each of the main chapters.

Part I

Type Inference & Elaboration

Chapter 2

Fully Grounding HDM Type Inference

2.1 Introduction

Classic unification-based type inference algorithms for the Hindley–Damas-Milner (HDM) system such as algorithm $\mathcal{W}$ [18] solve the *type inference problem*. That is, they determine whether programs that lack type signatures are well-typed or not, by assigning every subterm the most general type possible (i.e. an unconstrained unification variable) and solving any constraints that arise. Programs are well-typed if and only if all constraints can be solved.

However, depending on the setting, we would like to not only verify that a program is well-typed but also determine the type of every subterm. The canonical example of this is elaboration to System F [27, 65], but the problem arises in other settings as well. For example, real-world implementations of programming languages often allow developers to query the types of subterms, either via a REPL[1] like GHCi [79] or in a GUI-based editor, for example by supporting [81, 60] the Language Server Protocol [13]. We name this the *subterm type reconstruction problem*.

An crucial difference between type inference and subterm type reconstruction is the treatment of underconstrained types (i.e. types with unconstrained parts).

[1] Read–Eval–Print Loop

Consider the following program.

$$\textbf{let } x = (\lambda f.\ ())\ (\lambda y.\ y) \textbf{ in } \ldots$$

Here, the type of y is not subject to any constraints: since the abstraction $\lambda y.y$ is passed to a function that discards its argument and instead returns (), the abstraction is never applied an argument, nor is its output used, either which would impose constraints. This underconstrained type is not a problem in the case of type checking, since the program is well-typed regardless of y's type. However, in the case of subterm type reconstruction, underconstrained types *are* problematic, because we need to provide types for both f and y. When elaborating, we may only output *fully ground* types; unification variables are internal to the algorithm and should not be returned. Thus, to ground these types, we must instantiate all remaining unification variables. Generally, there are two options: (1) to generalise over the remaining variables, or (2) to default them to an arbitrary type (e.g. 1).

$$(1)\ \textbf{let } x = \Lambda\alpha.\, (\lambda f : \alpha \to \alpha.\ ())\ (\lambda y : \alpha.\ y) \textbf{ in } \ldots$$

$$(2)\ \textbf{let } x = (\lambda f : 1 \to 1.\ ())\ (\lambda y : 1.\ y) \textbf{ in } \ldots$$

Crucially, the determination of which variables to ground cannot be based on the type of the overall expression, since not all these variables need to occur in the overall expression's type. Consider again the example above. Since $(\lambda f.\ ())\ (\lambda y.\ y)$ beta-reduces to (), x's type is 1. Hence, the type of y does not occur in x's type. Therefore, additional machinery is needed to keep track of unsolved unification variables and apply whichever *grounding strategy* has been chosen. While solutions to this problem are not necessarily complicated in practice, implementations are often ad hoc, making reasoning about their correctness hard.

In this chapter, we address this very issue. We present algorithm $\mathcal{R}$, a *fully grounding* type inference algorithm for the HDM system. The algorithm explicitly tracks the scope of unification variables, which allows for fully grounding type inference, meaning we can infer fully ground types for all subexpressions. Since type grounding is internal to algorithm $\mathcal{R}$, its correctness proof (which we have mechanised in the Coq proof assistant [80]) carries over to the grounding strategy as well. As far as we know, we are the first to mechanically formalize a type inference algorithm for the HDM system that includes type grounding.

The algorithm utilizes in- and output contexts in the style of Dunfield and Krishnaswami [22] as well as a novel approach to unification, using a concept we dub *full contexts*. Here, contexts always contain all existing unification variables. Traditionally, inference algorithm thread through a substitution to

reflect equalities found during unification in other branches of the derivation. With our approach we avoid this threading: when an equality $\alpha := \tau$ is found, α can immediately be substituted for τ in the current context. Since the context is full, no further occurrences of α exist, and the equality can be discharged in one go.

In summary, the specific contributions of this chapter are:

- This chapter presents a new, *fully grounding* type inference algorithm $\mathcal{R}$ for ML-style polymorphism. The algorithm keeps track of all unification variables and their scope and uses the novel concept of full contexts to discharge all unifications in one go.
- We have mechanised both algorithm $\mathcal{R}$ as well as its correctness proof in the Coq proof assistant. Since algorithm $\mathcal{R}$ is fully grounding, we are—to our knowledge—the first to mechanically prove the correctness of an inference algorithm that features grounding. We admit an axiom about unification (see Section 2.5.3) and about the declarative specification (see Section 2.6.3).

2.2 Overview

This section describes the difference between unification-based algorithms like algorithm $\mathcal{W}$ and our algorithm $\mathcal{R}$. We first describe how algorithm $\mathcal{W}$ loses track of unconstrained type variables. We then propose our algorithm $\mathcal{R}$, which explicitly tracks the scope of unification variables, and show how this information yields fully grounding type inference.

Algorithm $\mathcal{W}$ Unification-based algorithms like algorithm $\mathcal{W}$ derive equality constraints at application sites $e_1\, e_2$. Rule $\mathcal{W}$-App of Figure 2.1 describes algorithm $\mathcal{W}$ in the case of applications.

Let us apply this to the example $(\lambda f.\ ())\ (\lambda y.\ y)$ (from in Section 2.1) under an empty context.

$$\dfrac{\dfrac{\begin{array}{c}\bullet \vdash_{\mathcal{W}} \lambda f.\ () : \alpha_1 \to 1, \bullet \\ \bullet \vdash_{\mathcal{W}} \lambda y.\ y : \alpha_2 \to \alpha_2, \bullet \\ \alpha_3 \text{ fresh} \end{array}}{\theta = \mathsf{unify}(\theta_2\tau \sim \tau_2 \to \alpha) = (\alpha_3 := 1, \alpha_1 := \alpha_2 \to \alpha_2)} \\ \Gamma \vdash_{\mathcal{W}} (\lambda f.\ ())\ (\lambda y.\ y) : 1, \theta}{\bullet \vdash_{\mathcal{W}} (\lambda f.\ ())\ (\lambda y.\ y) : 1, \theta} \quad \bullet = fv(1) \setminus fv(\bullet)$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{W}} e_1 : \tau, \theta_1 \\ \theta_1\Gamma \vdash_{\mathcal{W}} e_2 : \tau_2, \theta_2 \qquad \alpha \,\mathsf{fresh} \qquad \theta_3 = \mathsf{unify}(\theta_2\tau \sim \tau_2 \to \alpha)\end{array}}{\Gamma \vdash_{\mathcal{W}} e_1 e_2 : \theta_3\alpha, \theta_3\theta_2\theta_1}\ \mathcal{W}\text{-App}$$

$$\frac{\Gamma \vdash_{\mathcal{W}} e : \tau, \theta \qquad \overline{\alpha} = fv(\tau) \setminus fv(\theta\Gamma)}{\Gamma \vdash_{\mathcal{W}} e : \overline{\forall\alpha}.\tau, \theta}\ \mathcal{W}\text{-Gen}$$

(a) Algorithm $\mathcal{W}$

$$\frac{\begin{array}{c}\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \qquad \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \\ \widehat{\alpha} \# \Psi_2; (A_1', A_2) \qquad \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}\end{array}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}}\ \textsc{App}$$

$$\frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \qquad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}}\ \textsc{Gen}$$

(b) Algorithm $\mathcal{R}$

Figure 2.1: Application and generalisation of algorithm $\mathcal{W}$ (2.1a, top) and algorithm $\mathcal{R}$ (2.1b, bottom).

First, we infer the type $\alpha_1 \to 1$ for $\lambda f.\ ()$. Then, we infer the type $\alpha_2 \to \alpha_2$ for $(\lambda y.\ y)$. Both steps result in empty unifiers. Then, with α_3 fresh, we unify $\alpha_1 \to 1$ with $(\alpha_2 \to \alpha_2) \to \alpha_3$, yielding $\theta = (\alpha_3 := 1, \alpha_1 := \alpha_2 \to \alpha_2)$. Finally, we return $\theta(\alpha_3)$, which equates to 1. Since algorithm $\mathcal{W}$ only returns the function's result type 1, it loses track of free variables that only occur in the parameter's type (i.e. α_2). As α_2 is no longer reachable, it will not be further constrained and will remain unsolved.

Algorithm $\mathcal{W}$'s generalisation logic, extracted as $\mathcal{W}$-Gen[2] in Figure 2.1, turns an expression's monotype into a type scheme. In our running example, since the monotype 1 does not contain any free variables, algorithm $\mathcal{W}$ generalises over the empty list, which simply yields the 1 type scheme. Observe in particular that the unsolved unification variable α_2 is not generalised over. Hence, the type for $\lambda y.\ y$ remains $\alpha_2 \to \alpha_2$, but we do not know in which context α_2 is defined, and whether or where it can be generalised.

Algorithm $\mathcal{R}$ Our algorithm solves this problem by not only inferring the type T of an expression but also a list of unification variables that are in scope for T.

[2]Normally, this logic would be incorporated as part of the rule for let expressions.

By "in scope" we mean those variables that are safe to generalise over. Note that this list need not be a superset or subset of the free unification variables of T. We denote unification variables as $\widehat{\alpha}$ with A denoting a list of $\widehat{\alpha}$. Furthermore, we use $[A]T$ to denote the type T having A in scope.

Algorithm $\mathcal{R}$ utilises in- and output contexts [22] as well as the notion of *full contexts* to avoid having to pass around unifiers θ. We postpone fully introducing algorithm $\mathcal{R}$ to Section 2.4.2. For now, we present an informal preview of the application of algorithm $\mathcal{R}$ to the same example as covered above, highlighting how algorithm $\mathcal{R}$ infers fully ground types, and showing the benefit of full contexts.

$$\dfrac{\dfrac{\dfrac{\bullet \vdash \lambda f.\ () : [\widehat{\alpha}_1]\widehat{\alpha}_1 \to 1 \dashv \bullet}{\bullet; \{[\widehat{\alpha}_1]\widehat{\alpha}_1 \to 1\} \vdash \lambda y.\ y : [\widehat{\alpha}_2]\widehat{\alpha}_2 \to \widehat{\alpha}_2 \dashv \bullet; \{[\widehat{\alpha}_1]\widehat{\alpha}_1 \to 1\}} \quad \widehat{\alpha}_3 \# \bullet; \{[\widehat{\alpha}_1]\widehat{\alpha}_1 \to 1\}}{\bullet; (\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha}_3); \{\widehat{\alpha}_3\} \vdash \widehat{\alpha}_1 \to 1' \sim \widehat{\alpha}_2 \to \widehat{\alpha}_2 \dashv \bullet; \widehat{\alpha}_2; \{1\}}}{\Psi_{in} \vdash (\lambda f.\ ())\ (\lambda y.\ y) : [\widehat{\alpha}_2]1 \dashv \bullet} \quad \mathsf{gen}(1, \widehat{\alpha}_2) = \forall \widehat{\alpha}_2.1}{\Psi_{in} \vdash (\lambda f.\ ())\ (\lambda y.\ y) : \forall \widehat{\alpha}_2.1 \dashv \bullet}$$

For the application $(\lambda f.\ ())\ (\lambda y.\ y)$, in App, we first derive the type $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to 1)$ for $\lambda f.\ ()$. Then, we infer the type $[\widehat{\alpha}_2](\widehat{\alpha}_2 \to \widehat{\alpha}_2)$ for $\lambda y.\ y$. Here our notion of full contexts comes in: instead of deriving a unifier that needs to be applied to the type of $\lambda f.\ ()$, we instead append $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to 1)$ to the input context of the inference on $\lambda y.\ y$, and obtain a possibly further instantiated $[A_1']T'$ from the output context (as seen in rule App in Figure 2.1). Here, $[A_1']T' = [\widehat{\alpha}_1](\widehat{\alpha}_1 \to 1)$.

With $\widehat{\alpha}_3$ fresh, we unify $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to 1)$ with $[\widehat{\alpha}_3, \widehat{\alpha}_2]((\widehat{\alpha}_2 \to \widehat{\alpha}_2) \to \widehat{\alpha}_3)$. Again, we apply our notion of full contexts, appending all variables in scope for our to-be-unified types ($[\widehat{\alpha}_3, \widehat{\alpha}_1, \widehat{\alpha}_2]$) to the context, allowing us to retrieve a possibly further instantiated A_{out} from the output context. Here, $A_{out} = [\widehat{\alpha}_2]$. Furthermore, we append $\widehat{\alpha}_3$ once more, now occurring as a type instead of an in-scope unification variable. Since $\widehat{\alpha}_3$ enjoys any substitution occurring during unification, we obtain the possibly further instantiated T_{out} from the output context. Here, $T_{out} = 1$. Finally, we return $[\widehat{\alpha}_2]1$. Observe that, even though we are dropping the argument type, we are **not dropping the variables in scope of the argument type**. Generalisation, as displayed in Gen, of type $[\widehat{\alpha}_2]1$ is (almost) trivial.

To conclude this section, we have shown that our algorithm $\mathcal{R}$ not only infers a type T, but also a list of type variables A in scope for T. This way it can infer a fully ground type for every subterm. The following sections formally introduce algorithm $\mathcal{R}$.

$$
\begin{array}{rrcl}
x & \text{Variables} & \alpha & \text{Skolem type variables} \\
& \text{Terms} \quad e & ::= & x \mid () \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
& \text{Monotypes} \quad \tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \\
& \text{Type Schemes} \quad \sigma & ::= & \tau \mid \forall \alpha.\sigma \\
& \text{Scoping/Typing Context} \quad \Gamma & ::= & \bullet \mid \Gamma; \alpha \mid \Gamma; x : \sigma
\end{array}
$$

Figure 2.2: Syntax of the Declarative Specification

2.3 Declarative System

Before we present our algorithm, we present the declarative system that serves as its specification. The declarative system is essentially the syntax-directed system of Clement et al. [15], with two changes. First, like System F [27, 65], we explicitly track type variables in an ordered context. Consequently, we only generalise over variables that occur at the end of the context (i.e., not occurring to the left of term variable bindings). The second change is a purely syntactic one: we have extracted generalisation into a separate judgment.

2.3.1 Syntax

Figure 2.2 displays the syntax of the declarative system. The terms and types are as given by Damas and Milner [18]. Terms consist of term variables, () values, lambda abstractions, applications, and let-bindings. Type schemes are in Skolem normal form, consisting of a number of quantifiers in front of a monotype. Finally, contexts Γ track the scope of type and term variables that are in scope of an expression.

2.3.2 Typing

Figure 2.3 displays the typing rules of our declarative system. As stated, we have extracted the generalisation logic in a separate judgment, giving rise to both a monomorphic typing judgment $\Gamma \Vdash_{\texttt{mono}} e : \tau$, and a polymorphic judgment $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, the latter of which is exclusively used in the typing rule for let-bindings rule DecLet. Rule DecGen uses the auxiliary function $\mathsf{gen}(\tau, \overline{\alpha})$, which generalises the passed τ over the passed $\overline{\alpha}$ in the usual way. The type- and context well-formedness judgments $\Gamma \vdash_{\texttt{ty}} \sigma$ and $\texttt{wf}(\Gamma)$ are standard. Finally, rule DecVar uses type subsumption [18, 15] to instantiate a type scheme. Since subsumption is only used in this manner, we could have given it the signature

$\Gamma \Vdash \sigma \geq \tau$ and omitted rule SSGen. Yet, the advantage of the subsumption rules in Figure 2.3 is that subsumption proofs can be done in multiple parts and combined using transitivity.

2.4 Algorithmic System

We now introduce algorithm $\mathcal{R}$. We discuss its syntax, rules and unification algorithm.

2.4.1 Syntax

Figure 2.4 displays the syntax used by algorithm $\mathcal{R}$. Observe that we now have two kinds of type variables: like our declarative system we have (Skolem) type variables representing types generalised over by a type scheme. We have added unification variables $\widehat{\alpha}$, which we refer to as existential type variables. Like Skolem type variables they are placeholders which can be substituted for other types. Accordingly, monotypes T may now also take the form of an existential type variable.

Contexts Ψ differ from their declarative counterparts in two significant ways. First, besides Skolem type variables, contexts also track the scope of existential type variables, similar to [22, 101]. However, unlike Skolem type variables, they are not simply appended as individual variables, but instead come in a list-like structure A. As unification may both split and solve existential type variables, reasoning about ranges of existential type variables traditionally [22] requires adding markers to the context. By putting them in a list we obtain the same reasoning power, without having to add explicit markers.

Secondly, types with their list of existential variables in scope may live in the context as an *invisible* object $\{[A]S\}$. These invisible objects, when combined with input and output contexts, are the essence behind *full contexts*, which we already introduced in Section 2.2. These allow us to append As and Ss on the context in branches of the inference algorithm that normally would not have them in scope. Invisible objects are invisible to membership $\in$, but visible to both substitution and fresh variable generation $\#$.

$\boxed{\Gamma \Vdash_{\texttt{mono}} e : \tau}$ Expression Typing

$$\frac{(x : \sigma) \in \Gamma \qquad \Gamma \Vdash \sigma \geq \tau}{\Gamma \Vdash_{\texttt{mono}} x : \tau}\ \textsc{DecVar} \qquad \frac{}{\Gamma \Vdash_{\texttt{mono}} () : 1}\ \textsc{DecUnit}$$

$$\frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma; x : \tau_1 \Vdash_{\texttt{mono}} e : \tau_2}{\Gamma \Vdash_{\texttt{mono}} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{DecAbs}$$

$$\frac{\Gamma \Vdash_{\texttt{mono}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \Vdash_{\texttt{mono}} e_2 : \tau_1}{\Gamma \Vdash_{\texttt{mono}} e_1\ e_2 : \tau_2}\ \textsc{DecApp}$$

$$\frac{\Gamma \Vdash_{\texttt{poly}} e_1 : \sigma \qquad \Gamma; x : \sigma \Vdash_{\texttt{mono}} e_2 : \tau}{\Gamma \Vdash_{\texttt{mono}} (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \tau}\ \textsc{DecLet}$$

$\boxed{\Gamma \Vdash_{\texttt{poly}} e : \sigma}$ Generalisation

$$\frac{\overline{\alpha} \# \Gamma \qquad \Gamma; \overline{\alpha} \Vdash_{\texttt{mono}} e : \tau \qquad \mathsf{gen}(\tau, \overline{\alpha}) = \sigma}{\Gamma \Vdash_{\texttt{poly}} e : \sigma}\ \textsc{DecGen}$$

$\boxed{\Gamma \Vdash_{\texttt{ty}} \sigma}$ Type Well-formedness

$$\frac{\alpha \in \Gamma}{\Gamma \Vdash_{\texttt{ty}} \alpha} \qquad \frac{}{\Gamma \Vdash_{\texttt{ty}} 1} \qquad \frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash_{\texttt{ty}} \tau_2}{\Gamma \Vdash_{\texttt{ty}} \tau_1 \to \tau_2} \qquad \frac{\Gamma; \alpha \Vdash_{\texttt{ty}} \sigma}{\Gamma \Vdash_{\texttt{ty}} \forall \alpha.\sigma}$$

$\boxed{\texttt{wf}(\Gamma)}$ Scoping/Typing Context Well-formedness

$$\frac{\texttt{wf}(\Gamma)}{\texttt{wf}(\Gamma; \bullet)} \qquad \frac{\texttt{wf}(\Gamma) \qquad \alpha \notin \Gamma}{\texttt{wf}(\Gamma, \alpha)} \qquad \frac{\texttt{wf}(\Gamma) \qquad \Gamma \Vdash_{\texttt{ty}} \sigma}{\texttt{wf}(\Gamma; x : \sigma)}$$

$\boxed{\Gamma \Vdash \sigma_1 \geq \sigma_2}$ Type Subsumption

$$\frac{}{\Gamma \Vdash \tau \geq \tau}\ \textsc{SSMon} \qquad \frac{\alpha \# \Gamma \qquad \Gamma; \alpha \Vdash \sigma_1 \geq \sigma_2}{\Gamma \Vdash \sigma_1 \geq \forall \alpha.\sigma_2}\ \textsc{SSGen}$$

$$\frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash [\tau_1/\alpha]\sigma \geq \tau_2}{\Gamma \Vdash \forall \alpha.\sigma \geq \tau_2}\ \textsc{SSInst}$$

Figure 2.3: Typing of the Declarative Specification

x Variables α Skolem type variables $\widehat{\alpha}$ Existential type variables

$$
\begin{array}{rlll}
\text{Terms} & e & ::= & x \mid () \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
\text{Monotypes} & T & ::= & \alpha \mid \widehat{\alpha} \mid 1 \mid T_1 \to T_2 \\
\text{Type Schemes} & S & ::= & T \mid \forall \alpha.S \\
\text{Local Existential Context} & A & ::= & \bullet \mid A, \widehat{\alpha} \\
\text{Scoping/Typing Context} & \Psi & ::= & \bullet \mid \Psi; \alpha \mid \Psi; A \mid \Psi; x : S \mid \Psi; \{[A]S\} \\
\text{Type Equalities} & \Omega & ::= & \bullet \mid T_1 \sim T_2, \Omega \\
 & \{S\} & \doteq & \{[\bullet]S\}
\end{array}
$$

Figure 2.4: Syntax of Algorithm $\mathcal{R}$

2.4.2 Inference algorithm

Figure 2.5 shows the rules of algorithm $\mathcal{R}$. Its main judgments feature in and output contexts, where the output context consists of the input context subjected to all unifications made in the derivation, which means sequences A may shrink or grow and substitutions may be made, but their basic structure is the same.

Rule Var looks up a variable in the context, and instantiates polytype S to $[A]T$ using instantiation, discussed below. Rule Unit is trivial.

While Abs may visually look different from conventional abstraction typing rules, it follows the same approach, with added machinery to derive the list of existential type variables in scope. Term variable x is assigned a fresh existential variable $\widehat{\alpha}$; this assignment is added to the context as well as (the singleton list) $\widehat{\alpha}$. We utilize full contexts to let $[\widehat{\alpha}]\widehat{\alpha}$ enjoy any unifications made during the recursive inference by appending them to the input context and obtaining the possibly further instantiated $[A_1]T_1$ from the output context.

Rule App, as already discussed in Section 2.2, first infers a type $[A_1]T$ for e_1. Inference proceeds on e_2, with the input context extended with $[A_1]T$, by using an invisible object. By usage of this invisible object we ensure that we can safely extend the context with $[A_1]T$, because it does not bring either A_1 or T into scope. We now unify e_1's type with a function consisting of e_2's type as argument, and fresh variable $\widehat{\alpha}$ as result. We do so under an context extended with all existential variables in scope for both types being unified, as well as $\widehat{\alpha}$, occurring as a type, instead of an in-scope variable. For this second occurrence of $\widehat{\alpha}$ we again use an invisible object, which avoids us bringing $\widehat{\alpha}$ into scope twice. We obtain the results from the unification's output.

Rule Gen, as already discussed in Section 2.2, is (almost) trivial: based on the

$\boxed{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}}$ Type Inference

$$\frac{(x : S) \in \Psi \qquad \Psi \vdash S \geq [A]T}{\Psi \vdash x : [A]T \dashv \Psi} \textsc{Var} \qquad \frac{}{\Psi \vdash () : [\bullet]1 \dashv \Psi} \textsc{Unit}$$

$$\frac{\widehat{\alpha} \# \Psi_{in} \qquad \Psi_{in}; \widehat{\alpha}; x : \widehat{\alpha} \vdash e : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1}{\Psi_{in} \vdash \lambda x.e : [A_1, A_2](T_1 \to T_2) \dashv \Psi_{out}} \textsc{Abs}$$

$$\frac{\begin{array}{c}\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \qquad \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \\ \widehat{\alpha} \# \Psi_2; (A_1', A_2) \qquad \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}\end{array}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}} \textsc{App}$$

$$\frac{\Psi_{in} \vdash e_1 : S \dashv \Psi \qquad \Psi; x : S \vdash e_2 : [A]T \dashv \Psi_{out}; x : S'}{\Psi_{in} \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [A]T \dashv \Psi_{out}} \textsc{Let}$$

$\boxed{\Psi_{in} \vdash e : S \dashv \Psi_{out}}$ Generalization

$$\frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \qquad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}} \textsc{Gen}$$

$\boxed{\Psi \vdash_{\mathtt{ty}} S}$ Type Well-formedness

$$\frac{\alpha \in \Psi}{\Psi \vdash_{\mathtt{ty}} \alpha} \qquad \frac{\widehat{\alpha} \in \Psi}{\Psi \vdash_{\mathtt{ty}} \widehat{\alpha}} \qquad \frac{}{\Psi \vdash_{\mathtt{ty}} 1} \qquad \frac{\Psi \vdash_{\mathtt{ty}} T_1 \qquad \Psi \vdash_{\mathtt{ty}} T_2}{\Psi \vdash_{\mathtt{ty}} T_1 \to T_2} \qquad \frac{\Psi; \alpha \vdash_{\mathtt{ty}} S}{\Psi \vdash_{\mathtt{ty}} \forall \alpha.S}$$

$\boxed{\mathtt{wf}(\Psi)}$ Scoping/Typing Context Well-formedness

$$\frac{\mathtt{wf}(\Psi)}{\mathtt{wf}(\Psi; \bullet)} \qquad \frac{\mathtt{wf}(\Psi) \qquad A \# \Psi}{\mathtt{wf}(\Psi, A)} \qquad \frac{\mathtt{wf}(\Psi) \qquad \Psi \vdash_{\mathtt{ty}} S}{\mathtt{wf}(\Psi; x : S)}$$

$$\frac{\mathtt{wf}(\Psi; A) \qquad \Psi; A \vdash_{\mathtt{ty}} T}{\mathtt{wf}(\Psi; \{[A]T\})}$$

$\boxed{\Psi \vdash S \geq [A]T}$ Polymorphic Type Instantiation

$$\frac{}{\Psi \vdash T \geq [\bullet]T} \textsc{InstMono} \qquad \frac{\widehat{\alpha} \# \Psi \qquad \Psi; (\widehat{\alpha}) \vdash [\widehat{\alpha}/\alpha]S \geq [A]T}{\Psi \vdash \forall \alpha.S \geq [(\widehat{\alpha}), A]T} \textsc{InstPoly}$$

Figure 2.5: Typing of Algorithm $\mathcal{R}$

recursive, monomorphic inference, we generalise T over A. Note that we do not derive a list of variables in scope of S: since we generalise over all existential variables in scope, this list would always be empty. Finally, we have rule Let, which first infers a polytype using Gen. Inference proceeds on e_2, on which the output is based.

Type Instantiation Type instantiation is of form $\Psi \vdash S \geq [A]T$, where context Ψ and polytype S are inputs, and the monomorphic instance T and list in scope A are outputs. Essentially, type instantiation takes a type of form $\overline{\forall \alpha}^i.T$, removes all quantifiers, and generates a fresh existential type variable $\widehat{\alpha}^i$ for each Skolem type variable α^i, and returns $[\overline{\widehat{\alpha}}^i]([\overline{\widehat{\alpha}}^i/\overline{\alpha}^i]T)$. For example, the `fst` projection of pairs instantiates to $\bullet \vdash \forall\alpha_1.\forall\alpha_2.(\alpha_1, \alpha_2) \to \alpha_1 \geq [\widehat{\alpha}_1, \widehat{\alpha}_2](\widehat{\alpha}_1, \widehat{\alpha}_2) \to \widehat{\alpha}_1$.

Well-formedness Type well-formedness for the algorithmic system is a moderate extension of the declarative one, adding a single rule that checks if existential type variables $\widehat{\alpha}$ are in the context Ψ. Observe that, since objects are invisible to set membership $\in$, $\{[\widehat{\alpha}]1\} \not\vdash_{\texttt{ty}} \widehat{\alpha}$.

Contexts are well-formed iff all contained existential type variables are unique and all contained types are well-formed w.r.t. the context to their left, with any A enclosed in an invisible object temporarily added to the context. The notation $A\#\Psi$ ensures not only that A is fresh w.r.t. Ψ, but also that all $\widehat{\alpha}$ in A are fresh w.r.t. each other. Since objects *are* visible to freshness $\#$, context $\{[\widehat{\alpha}]1\}; \widehat{\alpha}$ is ill-formed. Another interesting detail is that, while contexts Ψ may contain Skolem type variables α (and this is used to verify the well-formedness of types), well-formed contexts may not contain any Skolem type variables.

2.4.3 Unification

Figure 2.6 displays our unification algorithm. The judgment $\Psi_{in} \vdash \Omega \dashv \Psi_{out}$ unifies a list of constraints E of form $T_1 \sim T_2$ under input context Ψ_{in} and produces an output context Ψ_{out}. It can be viewed as the transitive closure of the single-step unification judgment $\Psi_1 \vdash \Omega_1 \longrightarrow \Psi_2 \vdash \Omega_2$, restricted to those sequences that end in $\bullet$.

Hole Notation The syntax $\Psi[\widehat{\alpha}]$ denotes the context $\Psi_L; (A_L, \widehat{\alpha}, A_R); \Psi_R$, where $\Psi[]$ is the context $\Psi_L; (A_L, A_R); \Psi_R$. A multi-hole notation splits the context into more parts. For example, $\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2]$ means $\Psi_1; (A_1, \widehat{\alpha}_1, A_2, \widehat{\alpha}_2, A_3); \Psi_2$

$\boxed{\Psi_{in} \vdash \Omega \dashv \Psi_{out}}$ Unification Algorithm

$$\frac{}{\Psi \vdash \bullet \dashv \Psi}\ \textsc{SolNil} \qquad \frac{\Psi_{in} \vdash T_1 \sim T_2, \Omega \longrightarrow \Psi \vdash \Omega \qquad \Psi \vdash \Omega \dashv \Psi_{out}}{\Psi_{in} \vdash T_1 \sim T_2, \Omega \dashv \Psi_{out}}\ \textsc{SolCons}$$

$\boxed{\Psi_1 \vdash \Omega_1 \longrightarrow \Psi_2 \vdash \Omega_2}$ Unification Algorithm (Single-step)

$$\begin{array}{rcl}
\Psi \vdash 1 \sim 1, \Omega & \longrightarrow_1 & \Psi \vdash \Omega \\
\Psi \vdash \widehat{\alpha} \sim \widehat{\alpha}, \Omega & \longrightarrow_2 & \Psi \vdash \Omega \\
\Psi \vdash (T_1 \to T_2) \sim (T_3 \to T_4), \Omega & \longrightarrow_3 & \Psi \vdash T_1 \sim T_3, T_2 \sim T_4, \Omega \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \to T_2), \Omega & \longrightarrow_4 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \\
 & & \quad (\widehat{\alpha}_1 \to \widehat{\alpha}_2) \sim (T_1 \to T_2), \Omega) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \\
 & & \quad \text{and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}] \vdash (T_1 \to T_2) \sim \widehat{\alpha}, \Omega & \longrightarrow_5 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \\
 & & \quad (T_1 \to T_2) \sim (\widehat{\alpha}_1 \to \widehat{\alpha}_2), \Omega) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \\
 & & \quad \text{and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim \widehat{\alpha}_2, \Omega & \longrightarrow_7 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash \Omega) \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_2 \sim \widehat{\alpha}_1, \Omega & \longrightarrow_8 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash \Omega) \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim 1, \Omega & \longrightarrow_9 & [1/\widehat{\alpha}](\Psi \vdash \Omega) \\
\Psi[\widehat{\alpha}] \vdash 1 \sim \widehat{\alpha}, \Omega & \longrightarrow_{10} & [1/\widehat{\alpha}](\Psi \vdash \Omega)
\end{array}$$

Figure 2.6: Unification Algorithm

or $\Psi_1; (A_1, \widehat{\alpha}_1, A_2); \Psi_2; (A_3, \widehat{\alpha}_2, A_4); \Psi_3$. Note that hole notation does not split invisible objects.

Single-step Unification The single-step unification algorithm essentially is a subset of Zhao et al.'s [101], taking only the cases that apply. Rules 1 and 2 simply discharge already-solved constraints. Rule 3 splits constraints on function types. Rules 7 and 8 deal with constraints on two existential type variables. Since our contexts are ordered, we avoid existential type variables escaping their scope by always substituting away the rightmost variable. Rules 9 and 10 solve constraints with an existential variable on one side, and 1 on the other.

Finally, rules 4 and 5 solve constraints with an existential variable $\widehat{\alpha}$ on one side, and a function type $T_1 \to T_2$ on the other. Because our contexts are ordered, and both T_1 and T_2 may contain existential variables to the left of

$\boxed{\Psi \rightsquigarrow \Gamma}$ Context instantiation

$$\frac{}{\Gamma \rightsquigarrow \Gamma} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\mathtt{ty}} \tau \qquad \Gamma; \overline{\alpha}; [\tau/\widehat{\alpha}]A; \Psi \rightsquigarrow \Gamma'}{\Gamma; (\widehat{\alpha}; A); \Psi \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma; \overline{\alpha}_1; \overline{\alpha}_2 \vdash_{\mathtt{ty}} \tau \qquad \Gamma; \{[\overline{\alpha}_1; \overline{\alpha}_2; A][\tau/\widehat{\alpha}]T\}; \Psi \rightsquigarrow \Gamma'}{\Gamma; \{[\overline{\alpha}_1; \widehat{\alpha}; A]T\}; \Psi \rightsquigarrow \Gamma'}$$

Figure 2.7: Context instantiation

$\widehat{\alpha}$, we do not directly unify $\widehat{\alpha} := T_1 \to T_2$, but instead split $\widehat{\alpha}$ into a function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, where $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are fresh w.r.t. the context. This way, rules 7 and 8 may correctly determine which existential variable to eliminate. Because of our notion of *full* contexts, after substitution we can discharge the fact that $\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2$, since no other occurrences of $\widehat{\alpha}$ exist. Finally, to ensure termination, we require $\widehat{\alpha}$ does not occur in $T_1 \to T_2$.

2.5 Metatheory

To reason about how declarative and algorithmic derivations relate, we first need a way of converting between them. We do so through context instantiation, which takes an algorithmic context and converts it to a declarative one. However, this instantiation leaves us with a problem: what to do about invisible objects? To make reasoning about the declarative system easier, we extend declarative contexts Γ with a rule for objects $\Gamma; \{[\overline{\alpha}]\sigma\}$, and assert we can rewrite these away.

Definition 2.5.1. $\Gamma_1 \equiv_{\alpha,x} \Gamma_2 \triangleq (\forall \alpha, \alpha \in \Gamma_1 \iff \alpha \in \Gamma_2) \ \wedge \ (\forall (x : \sigma), (x : \sigma) \in \Gamma_1 \iff (x : \sigma) \in \Gamma_2)$

Lemma 2.5.2. *If* $\Gamma_1 \Vdash_{\mathtt{mono}} e : \tau$ *and* $\Gamma_1 \equiv_{\alpha,x} \Gamma_2$*, then* $\Gamma_2 \Vdash_{\mathtt{mono}} e : \tau$*.*

2.5.1 Context instantiation

Figure 2.7 shows simplified context instantiation rules, which implicitly coerce Ψs to Γs and allow for the appending of α and A. They are meant to convey the intuition; their actual full definition can be found in the supplementary materials.

For existential type variables outside invisible objects, we choose a sequence of Skolem type variables $\overline{\alpha}$ and a declarative type τ that is well-typed w.r.t. the *already-instantiated* context Γ to its left as well as the chosen sequence $\overline{\alpha}$. We proceed by replacing $\widehat{\alpha}$ by $\overline{\alpha}$, and substituting τ for $\widehat{\alpha}$ in the remaining, still-to-be instantiated Ψ to its right. For $\widehat{\alpha}$'s in invisible objects the logic is similar, but the generated sequences A and substitutions stay local to the object itself.

2.5.2 Soundness

Using context instantiation, we can formulate the soundness of the algorithmic system. We want to show that, for every closed algorithmic derivation, *any* instantiation leads to a valid derivation in the declarative system.

Theorem 2.5.3 (Soundness of the algorithmic system)**.** *If* $\bullet \vdash e : [A]T \dashv \bullet$ *then for all* $A; \{T\} \rightsquigarrow \{\tau\}$ *we have that* $\bullet \Vdash_{\texttt{mono}} e : \tau$.

This formulation is too weak to prove directly. Instead, we prove a more general variant, from which soundness follows.

Lemma 2.5.4. *Given* $\texttt{wf}(\Psi_{in})$*:*

1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *then for all* $\Psi_{out}; A; \{T\} \rightsquigarrow \Gamma; \{\tau\}$ *we have that* $\Gamma \Vdash_{\texttt{mono}} e : \tau$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *then for all* $\Psi_{out}; \{S\} \rightsquigarrow \Gamma; \{\sigma\}$ *we have that* $\Gamma \Vdash_{\texttt{poly}} e : \sigma$.

The proof proceeds by mutual induction on the monomorphic and polymorphic algorithmic typing judgments. As the given instantiation instantiates the *output* context, we reason backwards through the algorithm. As a consequence, for rules App and Gen that have multiple recursive hypotheses, to invoke the induction hypotheses the second time we must produce an instantiation of the intermediate context from the instantiation of the output context. To allow for this, we have proven several lemmas about the backwards preservation of instantiation.

Lemma 2.5.5. *Both typing judgments and unification preserve instantiation. That is:*

1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
3. *If* $\Psi_{in} \vdash \Omega \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.

2.5.3 Completeness

Completeness states that, for any declarative derivation, there exists an algorithmic derivation that instantiates to it.

Theorem 2.5.6 (Completeness of the algorithmic system)**.** *For each declarative derivation there exists an algorithmic derivation that instantiates to it. That is,*

1. *If* $\bullet \Vdash_{\texttt{mono}} e : \tau$ *then there exists* $A\ T$ *such that* $A; \{T\} \rightsquigarrow \{\tau\}$ *and* $\bullet \vdash e : [A]T \dashv \bullet$.
2. *If* $\bullet \Vdash_{\texttt{poly}} e : \sigma$ *then there exists* σ' *such that* $\bullet \vdash e : \sigma' \dashv \bullet$ *and* $\Vdash \sigma' \geq \sigma$.

Observe that (2) from Theorem 2.5.6 asserts that a polytype σ' *not containing any existential type variables* is inferred. In other words, σ' is fully ground. Again, we proceed by proving a more general lemma.

Lemma 2.5.7. *Given* $\texttt{wf}(\Psi_{in})$*:*

1. *If* $\Gamma \Vdash_{\texttt{mono}} e : \tau$, $\Gamma' \leqq_{\alpha,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* $T\ A\ \Psi_{out}\ \Gamma''\ \overline{\alpha}$ *s.t.* $\Gamma' = \Gamma''; \overline{\alpha}$, $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out}; A; \{T\}; A_{in} \rightsquigarrow \Gamma''; \{\tau\}; \overline{\alpha}$.
2. *If* $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, $\Gamma' \leqq_{\alpha,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* $S\ \sigma'\ \Psi_{out}\ \Gamma''\ \overline{\alpha}$ *s.t.* $\Gamma' = \Gamma''; \overline{\alpha}$, $\Psi_{in} \vdash e : S \dashv \Psi_{out}$, $\Psi_{out}; \{S\}; A_{in} \rightsquigarrow \Gamma''; \{\sigma'\}; \overline{\alpha}$, *and* $\Gamma' \Vdash \sigma' \geq \sigma$.

Here, $\Gamma_1 \leqq_{\alpha,x} \Gamma_2$ iff two conditions hold. First, the contexts must contain the same type variables in the same order. Second, their term bindings $(x : S)$ must (1) bind the same names in the same order to (2) types that are related by subsumption under Γ_1.

Finally, we admit the following property about unification:

Axiom 2.5.8. *If a unifier exists, unification succeeds. That is, if* $\theta T_1 = \theta T_2$ *and* $\Psi_{in} \rightsquigarrow \Gamma$ *then there exists* Ψ_{out} *such that* $\Psi_{in} \vdash T_1 \sim T_2 \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$.

2.5.4 Decidability

In our algorithm there is only one part of which decidability is not obvious: unification. Hence, we prove its decidability here.

Theorem 2.5.9 (Decidability of unification)**.** *Given* $\forall T_1 T_2.\ T_1 \sim T_2 \in E \implies (\Psi_{in} \vdash_{\mathtt{ty}} T_1 \wedge \Psi_{in} \vdash_{\mathtt{ty}} T_2)$*, it is decidable whether there exists a* Ψ_{out} *such that* $\Psi_{in} \vdash \Omega \dashv \Psi_{out}$*.*

The proof proceeds by induction on the lexicographic measure $\langle |\Psi_{in}|_{\widehat{\alpha}}, |\Omega| + 2 * |\Omega|_{\to} \rangle$, representing the number of existential type variables in Ψ_{in} and the length and number of function arrows in Ω, respectively. All rules directly reduce this measure, except for rules 4 and 5. For these, we need an additional lemma, from which these cases follow. Let us categorize lists of constraints where one side is an existential type variable that does not occur in the rest of the list as Ω_i, and assert that we can solve any head of pattern Ω_i without increasing the length of the tail.

$$\begin{array}{rcll} \Omega_i & ::= & \bullet & \\ & | & \widehat{\alpha} \sim T, \Omega_i & \text{with } \widehat{\alpha} \notin \Omega_i \\ & | & T \sim \widehat{\alpha}, \Omega_i & \text{with } \widehat{\alpha} \notin \Omega_i \end{array}$$

Lemma 2.5.10 (Solving Ω_i)**.** *For all* $\Psi_{in}\ \Omega_i\ \Omega$ *there exist* $\Psi_{out}\ \Omega'$ *such that* $\Psi_{in} \vdash \Omega_i + \Omega \longrightarrow^* \Psi_{out} \vdash \Omega'$ *and* $|\Psi_{out}|_{\widehat{\alpha}} = |\Psi_{in}|_{\widehat{\alpha}} - |\Omega_i|$*.*

Proof. By induction on $\langle |\Omega_i| + 2 * |\Omega_i|_{\to} \rangle$. Rules 1, 9 and 10 do not apply. The rest directly reduce the measure, except for (again) rules 4 and 5. We consider rule 4, where $\Omega_i = \widehat{\alpha} \sim (T_1 \to T_2), \Omega_i'$. It must be immediately followed by rule 3, which gives us $\Psi_{in}[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \to T_2), \Omega_i, \Omega \longrightarrow^* [\widehat{\alpha}_1 \to \widehat{\alpha}_2]\Psi_{in}[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim T_1, \widehat{\alpha}_2 \sim T_2, \Omega_i, \Omega$. Because we know $\widehat{\alpha} \notin \Omega_i'$, we know any substitution of $\widehat{\alpha}$ on Ω_i does not increase $|\Omega_i|_{\to}$. Even though we have added an existential variable, we end up with a decreased measure because we have eliminated an arrow, which counts for two. □

2.6 Mechanization

We have mechanised both the declarative specification presented in Section 2.3 as well as the algorithmic system presented in Section 2.4 in the Coq proof assistant [80]. Furthermore, we have proven the algorithmic system sound and correct w.r.t. the declarative speciation following the approach described in Section 2.5. The mechanization is implemented by generating definitions with Ott [71] and its backend [100] for the locally nameless representation [4, 14, 49]. To reason about the locally nameless representation, we have generated many useful lemmas with LNgen [3]. The mechanisation consists of ±700 handwritten lines of Ott DSL, ±10 000 lines of handwritten Coq code, ±900 lines of Coq code generated by Ott, and ±6 800 lines of Coq code generated by LNgen.

We start this section with a discussion of these tools and the locally nameless representation. Then, we discuss the major points of difference between what is presented in the chapter and the formalization. The mechanisation as well as an exhaustive list of the delta between the chapter and the mechanization are available in the supplementary material, as well as at https://github.com/rogerbosman/hdm-fully-grounding.

2.6.1 Ott

Our mechanization uses the Sewel et al.'s Ott [71] DSL to express both the syntax and inference rules in this chapter and generate corresponding (LaTeX and) Coq definitions, as well as boilerplate definitions such as substitutions and free variable functions. As Sewell et al. already argue the general benefits of Ott, here we focus only on the aspects that we found particularly useful.

Typically, manually written LaTeX specifications make notational liberties that do not translate well to Coq. For example, we have taken such a liberty in the context instantiation judgment as discussed in Section 2.5.1. Ott rejects such ill-typed definitions. Hence, it forces well-typed formulations that can be translated to Coq, but are more verbose in LaTeX. As a compromise, we have stuck to the Ott-generated LaTeX during the development and have manually produced a cleaned-up version for this chapter.[3]

A clear advantage of the Ott-generated outputs is that they both have the same single source of truth. Thus, the LaTeX output can be used to reason about the Coq output. Another substantial advantage is that Ott takes care of generating boilerplate definitions such as free variable functions and substitutions.

2.6.2 The locally nameless representation

Formalizations that contain abstraction must represent variables in some way. Typically, variables are either referred to by explicit name—which suffers from the lack of built-in α-equivalence, and have issues such as shadowing—or a nameless representation such as De Bruijn indices [19], which are sensitive to the context in which they are defined, requiring *shifting* operations whenever such changes occur.

The locally nameless representation combines the two approaches: it uses a named representation for free variables, and a nameless representation for locally bound variables. As a consequence, each alpha-equivalence class of

[3] We describe the difference in Section 2.6.4.

closed lambda terms has a unique representation. At the same time, terms are less sensitive to changes in their context. For example, the lambda expression $\lambda x.\, x\ y$ is represented as $\lambda.\, 0y$, because x is locally bound, while y is free. This implies a well-formedness condition, namely that every nameless variable has a corresponding abstraction, in other words, that nameless variables are not free. This condition is called *locally closed*.

A locally bound variable can be converted to a named, free variable through *opening*, where any reference to the outermost abstraction is replaced by a named variable. We use e^x to denote opening term e with name x. It's dual is *closing*. We use ${}^{\backslash x}e$ to denote closing e w.r.t. x. Our mechanization uses the locally nameless representation for both the declarative and algorithmic term variables x and for the Skolem type variables α. Since existential type variables $\widehat{\alpha}$ do not have a matching abstraction, they are always free, and thus use the named representation.

Cofinite Quantification To preserve the locally closed property, whenever we go under a binder, we have to open the term with some named variable quantified over in some way. There are several ways to go about this. One way would be to use existential quantification, where we assert that there exists some name not in the free variables of the term being opened. Consider rule ∀Wf-Ex below, which applies this principle to the well-formedness of declarative type schemes.

$$\frac{\exists \alpha.(\alpha \notin \mathit{fv}(\sigma) \cup \mathit{fv}(\Gamma) \qquad \Gamma; \alpha \vdash_{\mathtt{ty}} \sigma^{\alpha})}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{Wf-Ex}$$

$$\frac{\exists L.\forall \alpha.\alpha \notin L.\ \Gamma; \alpha \vdash_{\mathtt{ty}} \sigma^{\alpha}}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{Wf-Cof}$$

As described by Aydemir et al. [4], existential quantification is weak as an elimination form. For example, since eliminating this rule only gives well-formedness for one particular name, renaming lemmas are required for deriving well-formedness over any other name.

Universal quantification suffers from the opposite problem: it can be cumbersome to prove the well-formedness of *any* variable satisfying the freshness constaints. In particular, sometimes we want to exclude more variables than just those in $\mathit{fv}(\sigma) \cup \mathit{fv}(\Gamma)$.

Cofinite quantification, as displayed by rule ∀Wf-Cof above, offers exactly this. Here, we quantify universally over any name not in some existentially quantified set L. This elimination form is much stronger than with existential quantification, because we know well-formedness to hold for any $\alpha \notin L$, instead of just one, avoiding, in general, the need for renaming lemmas. Yet, as an

introduction form, it is much easier to use than with universal quantification, because it allows us to exclude finitely many names, instead of just the fixed set of free variables. While cofinite quantification is not free of quirks (particularly the control flow of quantification), which we describe below, in general, it strikes the best balance.

Ott's Locally Nameless Backend & LNgen One drawback of cofinite quantification is that implementation details of the variable representation leak to the LaTeX inference rules. Here, Ott's locally nameless backend [100] comes in handy: it automatically converts inference rules as specified in Sections 2.3 and 2.4 to those that use a (cofinitely quantified) locally nameless presentation *for Coq only*. The LaTeX definitions render as the original specification.

By default, Ott's locally nameless backend generates definitions for opening terms, but not for closing them. Weirich's Ott fork [91] adds the generation of these closing definitions.

The opening and closing operations are subject to various laws. One of these, which will become relevant later, is the following.

Proposition 2.6.1 (Subsitution as Open and Close)**.** *Subsitution can be defined in terms of* open *and* close*. That is,* $[T/\alpha]S = ({}^{\backslash\alpha}S)^{T}$.

Proposition 2.6.1 as well as many others are automatically generated and proven by LNgen [3], which bases itself on the Ott specification. Our mechanization uses these laws extensively.

2.6.3 Quirks of the locally nameless representation

As with any variable representation, some quirks arise. We cover three here.

Generalisation First is the gen function used in Gen in Figure 2.5, and its definition[4] is displayed below.

$$\begin{aligned}\mathsf{gen}(S,\ \bullet\ ,\_) &= S\\ \mathsf{gen}(S,(A;\widehat{\alpha}),L) &= \mathbf{let}\ S' = \mathsf{gen}(S,A,L),\ \alpha\#\mathit{fv}(S')\cup L\\ &\quad\ \mathbf{in}\ \ \forall.^{\backslash\alpha}([\alpha/\widehat{\alpha}]\ S')\end{aligned}$$

[4]Observe that gen is parametrised with a third argument, unspecified in Gen, which is included in the set w.r.t. fresh variables are generated, i.e. $\alpha\#\mathit{fv}(S')\cup L$. Since fresh variables are immediately closed away, the generalised term is not affected by a choice for L. It is helpful proving the commutativity of generalisation with for example substitution of existential type variables.

Since variable closing closes nameless Skolem type variables α only, we first substitute in a freshly generated one, only to close it away immediately after. While it would be possible to manually define a closing operation that replaces (named) existential type variables with unnamed Skolem type variables, we would lose the ability to reason over them with the laws generated by LNgen. While we cannot completely avoid having to manually replicate some of these in some instances, here we can avoid doing so. Fortunately, because of these same LN-generated laws, reasoning about this is straightforward. If we open the generalised term with some T, we get $({}^{\backslash\alpha}([\alpha/\widehat{\alpha}]\ S'))^{T}$. By Proposition 2.6.1, this can be rewritten into $[T/\alpha][\alpha/\widehat{\alpha}]\ S'$, which simplifies to $[T/\widehat{\alpha}]\ S'$.

Lists of variables Rule DecGen in Figure 2.2 quantifies over a list of variables $\overline{\alpha}$. Quantifying cofinitely over and opening with a list of type variables instead of a singular variable requires additional machinery and is not supported by Ott. Attempts at patching the generated definitions manually were unsuccessful (we discuss this again in Section 2.8). As a consequence, the list of variables $\overline{\alpha}$ is quantified existentially, which is why we used an axiom in our proof of the weakening lemma for declarative typing judgments.

Control Flow When inducting over typing derivations, we have existentially quantified sets of variables L, and universally quantified variables fresh w.r.t. L. Sets L flow downwards from the induction hypothesis to the conclusion. Yet, variables flow upwards from the conclusion to the induction hypothesis. Consider the abstraction case for completeness, which essentially consists of proving the following implication.

$$\begin{aligned}(\exists L.\forall x.x \notin L \implies \exists\ \Psi_{out}\ A\ T_2.\ \Psi_{in}; [\widehat{\alpha}]; x : \widehat{\alpha} \vdash e^x : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1 \\ \wedge\ \Psi_{out}; A_1; x : T_1; A_2; \{T_2\} \rightsquigarrow \Gamma; x : \tau_1; \{\tau_2\}) \\ \implies \Psi_{in} \vdash \lambda.e : [A_1; A_2]T_1 \to T_2 \dashv \Psi_{out} \wedge \\ \Psi_{out}; A_1; A_2; \{T_1 \to T_2\} \rightsquigarrow \Gamma; \{\tau_1 \to \tau_2\}\end{aligned}$$

There is a problem here. Since we only obtain the term variable to open e with after applying the Abs constructor in the right branch of the conclusion, we do not have access to it in the left branch of the conclusion. Since the IH existentially quantifies objects that occur in both branches of the conclusion, we cannot simply apply the IH twice, once per branch. While the IH can probably be strengthened to shift the $\forall x$ to each of its two branches, we found it easier to apply the IH to a sufficiently fresh variable before splitting the conclusion. This leaves us with a typing derivation opened with a different term than required. However, this can be remedied straightforwardly with the following renaming lemma.

Lemma 2.6.2. $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \implies [y/x]\Psi_{in} \vdash [y/x]e : [A]T \dashv [y/x]\Psi_{out}$

2.6.4 Delta between the chapter and the mechanization

We cover the two most important differences between the system as presented in this chapter and the mechanization.

Unification To facilitate easier reasoning over unification, the mechanisation's single-step unification judgment rules do not apply the substitution directly, but instead output the substitution as a third output, giving unification the form $\Psi_1 \vdash \Omega_1 \longrightarrow \Psi_2 \vdash \Omega_2, \gamma$, where γ has form $[\overline{T}/\overline{\widehat{\alpha}}]$. Note that single steps return either the empty list, or a singleton list. The auxiliary judgment $\Psi_{in} \vdash \Omega \dashv \Psi_{out}, \gamma$ takes the substitution generated by the single-step judgment, applies it to the step's result (yielding the same result as the paper's single-step judgment), and then combines it with the inductive result. Finally, $\Psi_{in} \vdash \Omega \dashv \Psi_{out}$ is defined in terms of this auxiliary judgment by simply discarding the substitution.

Context Instantiation The instantiation as presented in Section 2.5.1 contains notation that is not properly translatable to an inductive type. We present instantiation in this manner to obtain a simpler overview of the logic of context instantiation. The instantiation in the mechanization can be obtained by applying the following three transformations.

First, instead of concatenating the already-processed Γ with the yet-to-be processed Ψ, we define instantiation inductively on Ψ, where we pattern match on the different heads of Ψ, process the tail, and then add the processed head. This means that when generating a substitution for an existential type variable $\widehat{\alpha}$ we do not have access to the yet-to-be processed Ψ, since now $\widehat{\alpha}$ is at the head. Therefore, we flip the control flow, instead deriving a substitution θ of form $[\overline{\tau}/\overline{\widehat{\alpha}}]$, and apply it to any bound type later. This yields a signature of $\Gamma \rightsquigarrow \Psi, \theta$.

Then we split out the instantiation of A's in a dedicated judgment, $A \rightsquigarrow \overline{\alpha}, \theta$. Finally, to make it easier to reason about instantiation, we add a context Γ_{in}, θ_{in} such that the following holds.

Theorem 2.6.3 (Splitting and merging context instantiation)**.** *Context instantiation judgments can be split and merged. That is:*

- $\Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma, \theta \implies \exists\, \Gamma_1\ \Gamma_2\ \theta_1\ \theta_2,\ \Gamma = \Gamma_1; \Gamma_2 \wedge \theta = \theta_2; \theta_1 \wedge \Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2$.
- $\Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2 \implies \Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma_1; \Gamma_2, \theta_2; \theta_2$.

In the next chapter (specifically, in Section 3.6) we present a version of this style of context instantiation.

2.7 Related Work

The algorithm presented in this chapter extends a long line of work on the inference of the HDM system [21, 53, 25, 28]. Yet, a surprisingly small amount of work addresses the issue of underconstrained type variables.

Pottier [61] gives an (not formalised) elaboration algorithm which inspects the accumulated constraints to determine the list of variables in scope of types. In the appendix, they identify the problem of potentially unnecessary quantification. They address this with a non-deterministic specification that "magically" chooses which variables to abstract over.

Vytiniotis et al. advocate [86] removing the generalisation of lets altogether, citing unwanted interactions and needless complexity in context of generalising types with constraints arising from, for example, type classes or GADTs [94]. They observe that removing let generalisation would not be a significant restriction, since most programs do not utilize this functionality. Yet, removing let generalisation would not address the problem of underconstrained types: they would still need to be dealt with, only now by defaulting, since generalisation is no longer an option.

Zhao et al. mechanised [101] an algorithm for Dunfield and Krishnaswami's [22] type system featuring higher-rank polymorphism. However, since these systems are bidirectional, it is left to the programmer to decide which type variable should be generalised over where, if at all. Yet, we have taken a great deal of inspiration from both these works, adopting the in- and output contexts from Dunfield and Krishnaswami, and manner of tracking existential type variables and approach to unification from Zhao et al.

Zhao et al. rewrote Dunfield and Krishnaswami's algorithmic system, citing the lack of support by their proof assistant of choice (Abella [24]) as one of their reasons. Since we are not using any built-in variable binding support (like what is supported by Abella), we did not encounter such limitations. Thus, we were

able to maintain the tree-like structure of Dunfield and Krishnaswami instead of the flatter, list-based approach of Zhao et al.

2.8 Conclusion & Next Step

In this chapter we have presented algorithm $\mathcal{R}$: the first mechanically verified, fully grounding type inference algorithm for the HDM system. The contribution features the novel approach to unification by using full contexts, in which the current context always represents the *entire* context. The algorithm lays the foundation for formalizing algorithms that require determining types for every subterm.

Alternate solution The most canonical example of this a problem that requires determining types for every subterm is elaboration to an explicitly typed language. An alternative approach for grounding in the context of elaboration to an explicitly typed language is to base grounding on the elaborated terms. As motivated in Section 2.1, a free-variables based approach does not work when taking the free variables from the derived types, since not all variables occur in the type. However, depending on the situation, the elaborated term *does* contain all required information.

Whilst this is a valid solution, it creates a dependency of the inference algorithm to a specific elaboration target. This means that switching the elaboration target, for example to a non-explicitly typed language, impacts the inference logic, as a new grounding strategy must be found. By ensuring that our inference algorithm is grounding on its own we do not have such dependencies, which means that the algorithm as presented in this chapter should be suitable for extensions with numerable features without needing to change the grounding strategy.

Locally nameless While any variable representation will have its quirks, the quirks of locally nameless as discussed in Section 2.6.3 make us wonder if a fully nameless representation would be easier to work with. Our design choice of a separate judgment for generalisation did not turn out well. This approach requires mutual induction on the monomorphic and polymorphic typing judgments, which is a nuisance. Furthermore, Coq not being able to generate this mutual induction scheme is what left us unable to manually patch the inference rule for generalisation to quantify the list of variables $\overline{\alpha}$ cofinitely, as discussed in Section 2.6.3.

Extension with elaboration The logical next step is to extend the system as presented in this chapter with elaboration. We tackle this in the next chapter, Chapter 3.

Before extending the system as presented in this chapter with elaboration, we highlight two simplifications that can be made. Firstly we observe that the rules that generalize expressions into polytypes (the declaratie rule DecGen and algorithmic Gen) can be inlined in their respective let rules (rule DecLet and Let). This simplification is mostly one in the mechanisation, since it avoids mutually recursive definitions and therefore problems with mutual induction.

Secondly, and more impactfully, we observe that the distinction between existential type variables $\widehat{\alpha}$ (meant to symbolize yet-to-be-solved holes in types) and Skolem type variables α (meant for usage as type variables in the traditional sense) is not actually necessary because of how they occur. In fact, as can be seen below, existential type variables $\widehat{\alpha}$ occur only free. Skolem type variables α however, in general, are used only to refer to locally bound variables (i.e. they are only used to refer to refer to abstractions present within the type). Sometimes, most notably during generalisation and as part of the well-formedness judgment for type schemes, Skolem type variables α occur free, but only temporarily. This means the distinction between existential and Skolem type variables can be dropped, which simplifies the algorithmic types. More importantly however, since the main difference between the declarative and algorithmic grammars is the presence of existential type variables, is that after merging existential and Skolem type variables, the declarative and algorithmic grammars can be merged as well. This significantly simplifies the systems as presented here and in particular the mechanisation.

	Occurs free	Occurs locally bound
Existential type variables $\widehat{\alpha}$	Yes	No
Skolem type variables α	No[5]	Yes

2.9 Scientific output

This chapter is based on the following publication [9]:

> Bosman, R., Karachalias, G., and Schrijvers, T. No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM

[5] In rare circumstances they *do* occur free, but this can be mitegated.

System. In 14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (2023), A. Naumowicz and R. Thiemann, Eds., vol. 268 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:18

With the exception of an original version of the (on-paper) calculus presented here, which was written by this paper's second author, all of the contributions described here, including the adaptation of said original calculus, have been authored by me.

Chapter 3

Elaboration

3.1 Introduction

Modern, statically typed programming languages can alleviate the burden on programmers by the automatic generation of boilerplate code based on the available type information. Type classes are the most well-known example, originally introduced in Haskell [90] but occuring (in some form) in many languages such as C++, Mercury, OCaml and Rust. Type classes are an instance of boilerplate code generation since typeclasses can be seen as syntactic sugar for dictionary passing, which is much more verbose. Other examples of features implemented by automatic code generation are Implicits [68] (arguments of functions that are passed implicitly, as implemented in Scala [54] and Agda [20]) and intersection types [16].

What binds these features together is that they are all implemented by means of elaboration algorithms, i.e. algorithms that, in addition to verifying an input program is well-typed, give back the input program translated into another language called the *elaboration target*. The elaboration target is often explicitly typed, and in the case of features implemented by elaboration, more verbose. For example, type classes are typically implemented by elaboration by converting type classes in the input into dictionaries in the elaboration target. This way there is no need for supporting type classes further downstream, since they have been desugared away.

Whereas the *type inference* problem has been analysed in great detail, establishing properties both on paper and trough mechanical proofs, the *elaboration problem* has not gotten the same amount of attention. In this chapter

we take a step towards a similar understanding for elaboration algorithms by extending the fully grounding type inference algorithm of Chapter 2 with elaboration. While this does not yet support any features implemented by automatic code generation, any such algorithm can be seen as an extension of the algorithm developed in this chapter. As far as we know, a formalization of an type-inference-with-elaboration algorithm for the HDM system does not yet exist. Therefore, formalizing such an elaboration algorithm is the logical first step.

In the rest of this chapter we present an elaboration system for the HDM system. It is based on the system of Chapter 2 with the simplifications described in Section 2.8 applied. Since one of those simplifications merges the syntax, we first present the shared syntax and some shared judgments (Section 3.2). After that we present the target system (Section 3.3), declarative specification (Section 3.4) and elaboration algorithm (Section 3.5). We then set up for the metatheory by defining context instantiation (Section 3.6) and set up an equivalence relation for terms (Section 3.7). Finally we discuss correctness (Section 3.8) and its formalisation in Coq (Section 3.9). While unlike the formalization of Chapter 2, the formalisation correctness proofs have not been fully mechanized, all definitions have been formalized.

3.2 Syntax & Shared Judgments

The target, specification and elaboration systems presented next all use the same syntax and well-formedness judgments. Therefore, we present these here, after which we can present the individual systems.

3.2.1 Syntax

Figure 3.1 displays the syntax of the algorithmic, declarative, and target systems. The algorithmic and declarative systems type expressions e and elaborate to terms t. We utilize *type schemes* σ, which consist of types that do not contain any (universal) quantification (i.e. monotypes τ) prepended with a sequence of universal quantifiers. Types and contexts are shared between the specification and target systems: both expressions e and terms t are assigned types τ and σ under contexts Ψ. The objects as introduced in Chapter 2 have been extended to include a term.

Evaluation contexts E are used to define the operational semantics, only defined in the target system, since our logical relation uses this judgement. Furthermore,

x Variables $\quad \alpha$ Type variables

Monotypes	τ	$::=$	$\alpha \mid 1 \mid \mathcal{B} \mid \tau_1 \to \tau_2$
Polytypes	σ	$::=$	$\tau \mid \forall \alpha . \sigma$
Expressions	e	$::=$	$x \mid () \mid e_1\ e_2 \mid \lambda x.\, e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
Terms	t	$::=$	$x \mid () \mid \top \mid \bot \mid t_1\ t_2 \mid t\,[\tau] \mid \lambda (x : \tau).\, t \mid \Lambda \alpha .\, t$
Typing Context	Ψ	$::=$	$\bullet \mid \Psi ; x : \sigma \mid \Psi ; \alpha \mid \Psi ; \{t : [A]\sigma\}$
Type Equalities	Ω	$::=$	$\bullet \mid \tau_1 \sim \tau_2, \Omega$
Evaluation Context	E	$::=$	$\bullet \mid E\ t \mid v\ E \mid E\,[\tau]$
Program Holes	C	$::=$	$\bullet \mid t\ C \mid C\ t \mid C\,[\tau] \mid \lambda (x : \tau).\, C \mid \Lambda \alpha .\, C$

Figure 3.1: Shared Syntax

we define program holes C which are used to verify the soundness of our logical relation.

Observe that our terms t are not as expressive as Girard's System F terms [27], because of the types used: we reuse our stratified HDM types to type terms t, instead of using the more general System F types, which allow (universal) quantification at any point. Furthermore, term abstractions may abstract over monotypes τ only, and types of bindings in contexts Ψ are of HDM type σ. This reduction in expressiveness is not a problem, because we are only interested in those terms that are in the image of our elaboration algorithm. Since the domain of our elaboration algorithm is the HDM system, we never run into this limitation in expressivity. Finally, our terms t and types σ trivially embed into System F terms and types, which means our results carry over to System F as well.

Even though expressions e and terms t share types, expressions e do not have productions for $\top$ and $\bot$. Terms t have these to prove the logical relation sound, and it was easier to have a slightly bigger shared type than to have two types.

3.2.2 Well-formedness

Since the well-formedness judgments are syntactic in nature and the syntax is shared over all systems, the well-formedness judgments (shown in Figure 3.2) are shared as well. Their definition is standard, with the exception of WFE-OBJ, which extends context Ψ with the object's A before asserting the object's term t and type σ under this extended context.

$\boxed{\Psi \vdash_{\mathtt{ty}} \sigma}$ Type Well-formedness

$$\frac{\alpha \in \Psi}{\Psi \vdash_{\mathtt{ty}} \alpha} \qquad \frac{}{\Psi \vdash_{\mathtt{ty}} 1} \qquad \frac{}{\Psi \vdash_{\mathtt{ty}} \mathcal{B}} \qquad \frac{\Psi \vdash_{\mathtt{ty}} \tau_1 \qquad \Psi \vdash_{\mathtt{ty}} \tau_2}{\Psi \vdash_{\mathtt{ty}} \tau_1 \to \tau_2} \qquad \frac{\Psi; \alpha \vdash_{\mathtt{ty}} \sigma}{\Psi \vdash_{\mathtt{ty}} \forall \alpha . \sigma}$$

$\boxed{\Psi \vdash_{\mathtt{tm}} t}$ Term Well-formedness

$$\frac{x \in \Psi}{\Psi \vdash_{\mathtt{tm}} x} \qquad \frac{}{\Psi \vdash_{\mathtt{tm}} ()} \qquad \frac{}{\Psi \vdash_{\mathtt{tm}} \top} \qquad \frac{}{\Psi \vdash_{\mathtt{tm}} \bot} \qquad \frac{\Psi \vdash_{\mathtt{tm}} t \qquad \Psi \vdash_{\mathtt{ty}} \tau}{\Psi \vdash_{\mathtt{tm}} t\,[\tau]}$$

$$\frac{\Psi \vdash_{\mathtt{ty}} \tau \qquad \Psi; x : \tau \vdash_{\mathtt{tm}} t}{\Psi \vdash_{\mathtt{tm}} \lambda (x : \tau) . t} \qquad \frac{\Psi; \alpha \vdash_{\mathtt{tm}} t}{\Psi \vdash_{\mathtt{tm}} \Lambda \alpha . t}$$

$\boxed{\mathtt{wf}(\Psi)}$ Scoping/Typing Context Well-formedness

$$\frac{}{\mathtt{wf}(\Psi)}\ \textsc{WfE-Nil} \qquad \frac{\mathtt{wf}(\Psi) \qquad A \# \Psi}{\mathtt{wf}(\Psi, A)}\ \textsc{WfE-A} \qquad \frac{\mathtt{wf}(\Psi) \qquad \Psi \vdash_{\mathtt{ty}} \sigma}{\mathtt{wf}(\Psi; x : \sigma)}\ \textsc{WfE-Var}$$

$$\frac{\mathtt{wf}(\Psi; A) \qquad \Psi; A \vdash_{\mathtt{tm}} t \Psi; A \vdash_{\mathtt{ty}} \tau}{\mathtt{wf}(\Psi; \{t : [A]\tau\})}\ \textsc{WfE-Obj}$$

Figure 3.2: Well-formedness of types, terms, and contexts

3.3 Target system

Figure 3.3 displays the target system. As described in Section 3.2.1, terms t are less expressive (but embed in) System F terms [27], because they are typed using the HDM types σ. Given these types, the rules system as presented in Figure 3.3 are the rules from System F adapted in the expected way.

Term variables are looked up in the context, and have the type assigned to them by the context (rule TmVar). As for base cases, () has type 1 (rule TmUnit), and $\top$ and $\bot$ both have type $\mathcal{B}$ (rule TmTrue and rule TmFalse). Term abstractions, annotated with a *monotype* τ_1 (in accordence with being in the image of HDM expressions) have type $\tau_1 \to \tau_2$ if τ_1 is well-formed, and if the body has type τ_2 under the context extended with τ_1 assigned to the specified argument (rule TmAbs). An application $t_1\ t_2$ is valid if t_1 has a function type and if t_2 is of the argument type, and the whole expression's type is the result type of t_1 (rule TmApp). A type abstraction introduces an universal quantifier in the type, with a tail equal to the type of the abstractions body when typed under a context extended with the type variable abstracted over (rule TmTAbs).

$\boxed{\Psi \Vdash t : \sigma}$ Elaborated Term Typing

$$\frac{\Psi(x) = \sigma}{\Psi \Vdash x : \sigma}\ \textsc{TmVar} \qquad \frac{}{\Psi \Vdash () : 1}\ \textsc{TmUnit} \qquad \frac{}{\Psi \Vdash \top : \mathcal{B}}\ \textsc{TmTrue}$$

$$\frac{}{\Psi \Vdash \bot : \mathcal{B}}\ \textsc{TmFalse} \qquad \frac{\Psi \vdash_{\mathtt{ty}} \tau_1 \qquad \Psi; x : \tau_1 \Vdash t : \tau_2}{\Psi \Vdash \lambda(x : \tau_1).\, t : \tau_1 \to \tau_2}\ \textsc{TmAbs}$$

$$\frac{\Psi \Vdash t_1 : \tau_1 \to \tau_2 \qquad \Psi \Vdash t_2 : \tau_1}{\Psi \Vdash t_1\ t_2 : \tau_2}\ \textsc{TmApp} \qquad \frac{\Psi; \alpha \Vdash t : \sigma}{\Psi \Vdash \Lambda\alpha.\, t : \forall\alpha.\sigma}\ \textsc{TmTAbs}$$

$$\frac{\Psi \vdash_{\mathtt{ty}} \tau \qquad \Psi \Vdash t : \forall\alpha.\sigma}{\Psi \Vdash t\,[\tau] : [\tau/\alpha]\sigma}\ \textsc{TmTApp}$$

$\boxed{t \mapsto t'}$ Operational semantics

$$\frac{}{(\lambda(x : \tau).\, t_1)\ t_2 \mapsto [t_1/x]t_2}\ \textsc{OpApp} \qquad \frac{}{(\Lambda\alpha.\, t)\ \tau \mapsto [\tau/\alpha]t}\ \textsc{OpTApp}$$

$$\frac{t \mapsto t'}{E\,[t] \mapsto E\,[t']}\ \textsc{OpEC}$$

Figure 3.3: Target system

When applying a type τ to an abstraction of type $\forall\alpha.\sigma$, the universal quantifier is removed, and τ is substituted for α in σ (rule TmTApp).

As for the operational semantics, the direct combination of the elimination around the introduction form reduce for both term and type abstractions (Rule OpApp and rule OpTApp, respectively). Furthermore, by means of evaluation contexts (rule OpEC), reduction may happen under term and type application, where for term abstractions a call by value semantics is enforced by requiring the left term of an application be reduced to a value before the right term may be reduced (see Figure 3.1).

3.4 Declarative specification

Figure 3.4 displays the declarative specification. It is based on the declarative system presented in Chapter 2, with the simplifications proposed in Section 2.8

$\boxed{\Psi \vdash e : \tau \rightsquigarrow t}$ Term Typing

$$\frac{\Psi(x) = \sigma \qquad \Psi \vdash x : \sigma \geq t : \tau}{\Psi \vdash x : \tau \rightsquigarrow t}\ \textsc{DecVar}_{\rightsquigarrow} \qquad \frac{}{\Psi \vdash () : 1 \rightsquigarrow ()}\ \textsc{DecUnit}_{\rightsquigarrow}$$

$$\frac{\Psi \vdash_{\mathsf{ty}} \tau_1 \qquad \Psi; x : \tau_1 \vdash e : \tau_2 \rightsquigarrow t}{\Psi \vdash \lambda x.\, e : \tau_1 \to \tau_2 \rightsquigarrow \lambda(x : \tau_1).\, t}\ \textsc{DecAbs}_{\rightsquigarrow}$$

$$\frac{\Psi \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow t_1 \qquad \Psi \vdash e_2 : \tau_1 \rightsquigarrow t_2}{\Psi \vdash e_1\ e_2 : \tau_2 \rightsquigarrow t_1\ t_2}\ \textsc{DecApp}_{\rightsquigarrow}$$

$$\frac{A \# \Psi \qquad \Psi; A \vdash e_1 : \tau_1 \rightsquigarrow t_1 \qquad \Psi; x : \forall A.\, \tau_1 \vdash e_2 : \tau \rightsquigarrow t_2}{\Psi \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow [\Lambda A.\, t_1/x]t_2}\ \textsc{DecLet}_{\rightsquigarrow}$$

$\boxed{\Psi \vdash t_1 : \sigma_1 \geq t_2 : \sigma_2}$ Term and Type Subsumption

$$\frac{}{\Psi \vdash t : \tau \geq t : \tau}\ \textsc{SSMon}_{\rightsquigarrow} \qquad \frac{\Psi \vdash_{\mathsf{ty}} \tau_1 \qquad \Psi \vdash t_1\,[\tau_1] : [\tau_1/\alpha]\sigma \geq t_2 : \tau_2}{\Psi \vdash t_1 : \forall\alpha.\sigma \geq t_2 : \tau_2}\ \textsc{SSInst}_{\rightsquigarrow}$$

$$\frac{\Psi; (\alpha) \vdash t_1 : \sigma_1 \geq t_2 : \sigma_2}{\Psi; (\alpha) \vdash t_1 : \sigma_1 \geq \Lambda\alpha.\, t_2 : \forall\alpha.\sigma_2}\ \textsc{SSGen}_{\rightsquigarrow}$$

Figure 3.4: Typing of the Declarative Specification

(the inlining of the polymorphic judgments into their respective let judments, and the merging of existential and Skolem type variables) applied.

The expression () has type 1, and elaborates to () (rule $\textsc{DecUnit}_{\rightsquigarrow}$). Rule $\textsc{DecAbs}_{\rightsquigarrow}$ types term abstractions $\lambda x.\, e$. Given that e elaborates to t with τ_1 assigned to x, the abstraction elaborates to $\lambda(x : \tau_1).\, t$. Rule $\textsc{DecApp}_{\rightsquigarrow}$ types applications $e_1 e_2$. Given that e_1 and e_2 elaborate to t_1 and t_2, respectively, the application elaborates to $t_1\ t_2$.

Rule $\textsc{DecLet}_{\rightsquigarrow}$ types let expressions $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$. As stated it consists of the fully grounding let judgment with the polymorphic judgment (rule DecLet and rule DecGen, Figure 2.3) lined in, extended with elaboration. As we are implementing (well-scoped) let-polymorphism, we extend Ψ with a sequence of (fresh w.r.t. Ψ) type variables A in the derivation for e_1. We abstract both the type (in the derivation of e_2) as well as the term (in the elaborated term) over this A. As our elaboration target does not contain let expressions, we instead inline them by substituting t_1 (the elaborated term of e_1) generalized over A for

x in t_2 (the elaborated term of e_2). An alternative encoding of the let binding is to use abstraction and application (i.e. to set $(\lambda x.\, t_2)\ (\Lambda A.\, t_1)$ as the elaborated term for rule DecLet$_{\rightsquigarrow}$). This however would, by the operational semantics of the target system (Figure 3.3), step to the result produced by rule DecLet$_{\rightsquigarrow}$. We argue that there is no real difference in the complexity of the two options, so have opted for the further evaluated one.

To discuss the typing of variables x we must first discuss subsumption. Given a subsumption judgment of shape $\Psi \vdash \sigma \geq \tau$, such as the one in Figure 2.3, one might expect to extend this judgment with a coercion term that coerces terms of the more general type σ to terms of the more specific type τ (i.e. a term t_c of type $\Psi \Vdash t_c : \sigma \to \tau$). This, however, is not a valid HDM type because of the σ occurring to the left of $\to$. However, the problem lies in the fact that the coercion term t_c must first abstract over a term of type σ, and then instantiate it to τ. A workaround therefore is to already supply this term in the input, and to instantiate the passed term by applying a type for each quantifier (rule SSInst$_{\rightsquigarrow}$). As can be seen in Figure 3.4, this is the approach we chose. In the base case (rule SSMon$_{\rightsquigarrow}$) the (now fully applied) input is then carried to the output, after which it is passed back unaltered trough all recursive cases (rule SSInst$_{\rightsquigarrow}$). While this workaround is not as simple as a coercion term in the specification, it greatly simplifies the formalisation since it allows us to have one representation of types in the object language. Given an understanding of subsumption, the typing of variables (DecVar$_{\rightsquigarrow}$) is standard, and the elaborated term is the variable coerced into the desired monotype.

Finally, we define a types-only shorthand that compares types, defined in terms of the term and type subsumption.

Definition 3.4.1. $\Psi \vdash \sigma_1 \geq \sigma_2 := \exists t_1 t_2.\Psi \vdash t_1 : \sigma_1 \geq t_2 : \sigma_2$

3.4.1 Adapted declarative system

The system presented in Figure 3.4 has two drawbacks:

- It is unclear which variables are *in scope* for inferred types and elaborated terms. In other words, we cannot fully ground the derived types and terms.
- It does not allow for the introduction of fresh type variables at any point.

A better understanding of the correctness proof is needed to understand why these points need fixing. Therefore we will defer this motivation until

$\boxed{\Psi \vdash e : [A]\tau \rightsquigarrow t}$ Term Typing

$$\frac{\Psi(x) = \sigma \qquad \Psi \vdash x : \sigma \geq t : [A]\tau}{\Psi \vdash x : [A]\tau \rightsquigarrow t}\ \textsc{DecAVar}_{\rightsquigarrow} \qquad \frac{A\#\Psi}{\Psi \vdash () : [A]1 \rightsquigarrow ()}\ \textsc{DecAUnit}_{\rightsquigarrow}$$

$$\frac{A_1\#\Psi \qquad \Psi; A_1 \vdash_{\mathtt{ty}} \tau_1 \qquad \Psi; A_1; x : \tau_1 \vdash e : [A_2]\tau_2 \rightsquigarrow t}{\Psi \vdash \lambda x.\, e : [A_1, A_2]\tau_1 \to \tau_2 \rightsquigarrow \lambda(x : \tau_1).\, t}\ \textsc{DecAAbs}_{\rightsquigarrow}$$

$$\frac{\Psi \vdash e_1 : [A_1]\tau_1 \to \tau_2 \rightsquigarrow t_1 \qquad \Psi; A_1 \vdash e_2 : [A_2]\tau_1 \rightsquigarrow t_2}{\Psi \vdash e_1\ e_2 : [A_1, A_2]\tau_2 \rightsquigarrow t_1\ t_2}\ \textsc{DecAApp}_{\rightsquigarrow}$$

$$\frac{\Psi \vdash e_1 : [A_1]\tau_1 \rightsquigarrow t_1 \qquad \Psi; x : \forall A_1.\, \tau_1 \vdash e_2 : [A_1]\tau \rightsquigarrow t_2}{\Psi \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : [A_2]\tau \rightsquigarrow [\Lambda A_1.\, t_1/x]t_2}\ \textsc{DecALet}_{\rightsquigarrow}$$

$\boxed{\Psi \vdash t_1 : \sigma_1 \geq t_2 : [A]\sigma_2}$ Term and Type Subsumption

$$\frac{A\#\Psi}{\Psi \vdash t : \tau \geq t : [A]\tau}\ \textsc{SSAMon}_{\rightsquigarrow}$$

$$\frac{A_1\#\Psi \qquad \Psi \vdash_{\mathtt{ty}} \tau_1 \qquad \Psi; A_1 \vdash t_1\,[\tau_1] : [\tau_1/\alpha]\sigma \geq t_2 : [A_2]\tau_2}{\Psi \vdash t_1 : \forall\alpha.\sigma \geq t_2 : [A_1, A_2]\tau_2}\ \textsc{SSAInst}_{\rightsquigarrow}$$

$$\frac{\Psi; (\alpha) \vdash t_1 : \sigma_1 \geq t_2 : [A]\sigma_2}{\Psi; (\alpha) \vdash t_1 : \sigma_1 \geq \Lambda\alpha.\, t_2 : [A]\forall\alpha.\sigma_2}\ \textsc{SSAGen}_{\rightsquigarrow}$$

Figure 3.5: Typing of the Declarative Specification with A

Section 3.8.2, and proceed with the presentation of an altered system that does not exhibit these problems.

Figure 3.5 displays the altered declarative specification. Its main change is in the signature of the judgment: it now features a list of variables in scope, just like the algorithm presented in Chapter 2. Derived types and terms are well-formed w.r.t. the context appended with the list in scope:

Lemma 3.4.2. *If* $\Psi \vdash e : [A]\tau \rightsquigarrow t$ *and* $\mathtt{wf}(\Psi)$ *then* $\Psi; A \vdash_{\mathtt{ty}} \tau$ *and* $\Psi; A \vdash_{\mathtt{tm}} t$.

The main change in the rules themselves, apart from the change in signature, is that like non-well-scoped systems, fresh variables can be introduced at any point.

As far as subsumption is concerned, both subsumption rules allow to generate fresh variables, and return them as part of the output. Furthermore, we define another types-only shorthand.

Definition 3.4.3. $\Psi \vdash \sigma_1 \geq [A]\sigma_2 := \exists t_1 t_2.\Psi \vdash t_1 : \sigma_1 \geq t_2 : [A]\sigma_2$

Rule DecAVar$_{\rightsquigarrow}$ uses this altered subsumption judgment, setting the returned A as the list in scope for the instantiated type and elaborated term. For typing unit, rule DecAUnit$_{\rightsquigarrow}$ may output any (fresh w.r.t. Ψ) list A and set it as the list in scope.

For abstractions (rule DecAAbs$_{\rightsquigarrow}$) and applications (rule DecAApp$_{\rightsquigarrow}$), the context is extended with a fresh A_1 in the premises. In the result this generated A_1 is added to the recursive list in scope Ψ_2. Finally, rule DecALet$_{\rightsquigarrow}$ makes use of the list in scope to ground the type and term derived for e_1. The resulting list in scope is taken from the judgment over e_2.

The correctness proof will proceed with this altered declarative specification. The two systems correspond, in one direction up to equivalent terms, as can be seen in Theorem 3.4.7. We will define this equivalence in Section 3.7. One direction can be proven directly (Lemma 3.4.4). The other direction (Lemma 3.4.6) needs a property of the adapted declarative system (Lemma 3.4.5).

Lemma 3.4.4. *If* $\Psi \vdash e : [A]\tau \rightsquigarrow t$ *then* $\Psi; A \vdash e : \tau \rightsquigarrow t$.

Lemma 3.4.5. *If* $\Psi; A_1 \vdash e : [A_2]\tau \rightsquigarrow t$ *then there exists a* t' *such that* $\Psi \vdash e : [A_1, A_2]\tau \rightsquigarrow t'$ *and* $\Psi \Vdash t \approx t' : \tau$.

Lemma 3.4.6. *If* $\Psi \vdash e : \tau \rightsquigarrow t$ *then there exists a* t' *such that* $\Psi \vdash e : [\bullet]\tau \rightsquigarrow t'$ *and* $\Psi \Vdash t \approx t' : \tau$.

Theorem 3.4.7 (Correspondence between the declarative specifications)**.** $\bullet \vdash e : \tau \rightsquigarrow t \iff (\exists t', \bullet \vdash e : [\bullet]\tau \rightsquigarrow t' \wedge \Psi \Vdash t \approx t' : \tau)$

3.5 Elaboration algorithm

Figure 3.6 displays the algorithmic elaboration algorithm. It is based on the fully grounding algorithm presented in Chapter 2, once again with the proposed simplifications (Section 2.8).

Type instantiation, used by Var$_{\rightsquigarrow}$, utilizes the same "trick" as the declarative specification (Section 3.4): the judgment features an input term and an output term, where the output term is the coerced (instantiated) input term. The typing of unit (Unit$_{\rightsquigarrow}$) is unsurprising. For abstractions (Abs$_{\rightsquigarrow}$) we once again

$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$ Type Inference

$$\frac{(x : \sigma) \in \Psi \qquad \Psi \vdash x : \sigma \geq t : [A]\tau}{\Psi \vdash x : [A]\tau \rightsquigarrow t \dashv \Psi} \textsc{Var}_{\rightsquigarrow} \qquad \frac{}{\Psi \vdash () : [\bullet]1 \rightsquigarrow \Psi \dashv ()} \textsc{Unit}_{\rightsquigarrow}$$

$$\frac{\alpha \# \Psi_1 \qquad \Psi_1; \alpha; x : \alpha \vdash e : [A_2]\tau_2 \rightsquigarrow t \dashv \Psi_2; A_1; x : \tau_1}{\Psi_1 \vdash \lambda x.\, e : [A_1, A_2](\tau_1 \to \tau_2) \rightsquigarrow \lambda(x : \tau_1).\, t \dashv \Psi_2} \textsc{Abs}_{\rightsquigarrow}$$

$$\frac{\begin{array}{c}\Psi_1 \vdash e_1 : [A_1]\tau \rightsquigarrow t_1 \dashv \Psi_2 \\ \Psi_2; \{t_1 : [A_1]\tau\} \vdash e_2 : [A_2]\tau_1 \rightsquigarrow t_2 \dashv \Psi_3; \{t_1' : [A_1']\tau'\} \qquad \alpha \# \Psi_3; (A_1', A_2) \\ \Psi_3; (A_1', A_2, \alpha); \{t_1'\ t_2 : \alpha\} \vdash \tau' \sim \tau_1 \to \alpha \dashv \Psi_4; A; \{t : \tau_2\}\end{array}}{\Psi_1 \vdash e_1\ e_2 : [A]\tau_2 \rightsquigarrow \Psi_4 \dashv t} \textsc{App}_{\rightsquigarrow}$$

$$\frac{\begin{array}{c}\Psi_1 \vdash e_1 : [A_1]\tau \rightsquigarrow t_1 \dashv \Psi_2 \\ \Psi_2; x : \forall A_1.\, \tau_1; \{\Lambda A_1.\, t_1\}; \{A_1\} \vdash e_2 : [A_2]\tau \rightsquigarrow t_2 \dashv \Psi_3; x : \sigma; \{t_1'\}; \{A_1'\}\end{array}}{\Psi_1 \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [A_2]\tau \rightsquigarrow [t_1'/x]t_2 \dashv \Psi_3} \textsc{Let}_{\rightsquigarrow}$$

$\boxed{\Psi \vdash t_1 : \sigma \geq t_2 : [A]\tau}$ Polymorphic Type Instantiation

$$\frac{}{\Psi \vdash t : \tau \geq t : [\bullet]\tau} \textsc{InstMono}_{\rightsquigarrow}$$

$$\frac{\alpha \# \Psi \qquad \Psi; (\alpha) \vdash t_1\ \alpha : \sigma \geq t_2 : [A]\tau}{\Psi \vdash t_1 : \forall \alpha.\sigma \geq t_2 : [(\alpha), A]\tau} \textsc{InstPoly}_{\rightsquigarrow}$$

Figure 3.6: Typing of the algorithmic system

utilize full context to obtain the argument type τ_1, used not only in the resulting type, but also in the elaborated term as annotation.

When comparing the case for applications ($\textsc{App}_{\rightsquigarrow}$) with its fully grounding counterpart (App, Figure 2.5), we have an additional t_1 and t_2 obtained from elaborating e_1 and e_2. To ensure t_1 is applied all unifications made during the elaboration of e_2, we add t_1 to the object on its input context, and obtain a possibly further instantiated t_1' from its output context. We do the same with unification: we add the application $t_1'\ t_2$ to the input of unification, and obtain a possibly further instantiated term t from the output.

As for let expressions, like the declarative specification and as suggested in Section 2.8, $\textsc{Let}_{\rightsquigarrow}$ consists of the fully grounding let rule with the polymorphic rule (Let and Gen, Figure 2.3) lined in. Just like the declarative rule $\textsc{DecALet}_{\rightsquigarrow}$ the term t_1 derived during elaboration on e_1 will be ground over the list in scope of t_1. However, first it is appended to the input context of

$\boxed{\Psi_{in} \vdash \tau_1 \sim \tau_2 \dashv \Psi_{out}}$ $\boxed{\Psi_{in} \vdash \Omega_{in} \longrightarrow^* \Psi_{out} \vdash \Omega_{out}}$ Unification Algorithm

$$\frac{\Psi_{in} \vdash (\tau_1 \sim \tau_2) \longrightarrow^* \Psi_{out} \vdash \bullet}{\Psi_{in} \vdash \tau_1 \sim \tau_2 \dashv \Psi_{out}}\ \mathrm{U}$$

$$\frac{}{\Psi \vdash 1 \sim 1; \Omega \longrightarrow \Psi \vdash \Omega}\ \textsc{UssUnit} \qquad \frac{}{\Psi \vdash \alpha \sim \alpha; \Omega \longrightarrow \Psi \vdash \Omega}\ \textsc{UssVar}$$

$$\frac{}{\Psi \vdash \tau_1 \to \tau_2 \sim \tau_3 \to \tau_4; \Omega \longrightarrow \Psi \vdash \tau_1 \sim \tau_3; \tau_2 \sim \tau_4; \Omega}\ \textsc{UssArr}$$

$$\frac{\begin{array}{c}(\tau_1 = \alpha \wedge \tau_2 = \tau_3 \to \tau_4) \vee (\tau_1 = \tau_3 \to \tau_4 \wedge \tau_2 = \alpha) \\ \alpha \notin fv(\tau_3 \to \tau_4) \qquad (\alpha_1; \alpha_2) \# \Psi_1; (A_1; \alpha; A_2); \Psi_2\end{array}}{\begin{array}{l}\Psi_1; (A_1; \alpha; A_2); \Psi_2 \vdash \tau_1 \sim \tau_2; \Omega \longrightarrow \\ \quad [\alpha_1 \to \alpha_2/\alpha](\Psi_1; (A_1; \alpha_1; \alpha_2; A_2); \Psi_2 \vdash \Omega)\end{array}}\ \textsc{UssSplit}$$

$$\frac{(\tau_1 = \alpha \wedge \tau_2 = \beta) \vee (\tau_1 = \beta \wedge \tau_2 = \alpha) \qquad \alpha \in \Psi_1; A_1}{\Psi_1; (A_1; \beta; A_2); \Psi_2 \vdash \tau_1 \sim \tau_2; \Omega \longrightarrow [\alpha/\beta](\Psi_1; (A_1; A_2); \Psi_2 \vdash \Omega)}\ \textsc{UssSubEx}$$

$$\frac{(\tau_1 = \alpha \wedge \tau_2 = 1) \vee (\tau_1 = 1 \wedge \tau_2 = \alpha)}{\Psi_1; (A_1; \alpha; A_2); \Psi_2 \vdash \tau_1 \sim \tau_2; \Omega \longrightarrow [1/\alpha](\Psi_1; (A_1; A_2); \Psi_2 \vdash \Omega)}\ \textsc{UssSubUnit}$$

Figure 3.7: Unification Algorithm

the elaboration of e_2 such that it enjoys any unifications made. The possibly further instantiated t_1' is obtained from the output context and substituted for x in the returned elaborated term.

3.5.1 Unification

Figure 3.7 displays the unification algorithm of the elaboration algorithm. It implements the same algorithm as displayed our fully grounding algorithm (Figure 2.6), adapted in the expected manner to the change in syntax.

The presentation of Figure 3.7 matches the implementation, and therefore foregoes the notational niceties used in Chapter 2. First, here we do not use the hole notation introduced in Section 2.4.3. Instead, we match on the structure of context directly, and use regular membership $\in$ in the case of the double hole notation (rule UssSplit). Secondly, it essentially treats equalities $\tau_1 \sim \tau_2$ in Ω as *unordered* pairs, which avoids the duplication of the presentation in Chapter 2.

$\boxed{\Psi_{alg} \rightsquigarrow \Psi_{dec}}$ $\boxed{(\Psi, \theta_1) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta_2)}$ Context instantiation

$$\frac{(\bullet, \bullet) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta)}{\Psi_{alg} \rightsquigarrow \Psi_{dec}}\ \textsc{EInst}$$

$$\frac{}{(\Psi, \theta) \vdash \bullet \rightsquigarrow (\bullet, \bullet)}\ \textsc{EInstNil}$$

$$\frac{(\Psi, \theta_1) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta_2) \qquad A_{dec} \# \Psi_{dec} \qquad \Psi, \Psi_{dec}; A_{dec} \vdash A_{alg} \rightsquigarrow \theta_3}{(\Psi, \theta_1) \vdash \Psi_{alg}; A_{dec} \rightsquigarrow (\Psi_{dec}; A_{dec}, \theta_3 \circ \theta_2 \circ \theta_1)}\ \textsc{EInstA}$$

$$\frac{(\Psi, \theta_1) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta_2)}{(\Psi, \theta_1) \vdash \Psi_{alg}; x : \sigma \rightsquigarrow (\Psi_{dec}; x : (\theta_2 \circ \theta_1)\sigma, \theta_2)}\ \textsc{EInstX}$$

$$\frac{\begin{array}{c}(\Psi, \theta_1) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta_2) \\ A_{dec} \# \Psi_{dec} \qquad \Psi, \Psi_{dec}; A_{dec} \vdash A_{alg} \rightsquigarrow \theta_3\end{array}}{\begin{array}{l}(\Psi, \theta_1) \vdash \Psi_{alg}; \{t : [A_{alg}]\sigma\} \rightsquigarrow \\ \qquad (\Psi_{dec}; \{(\theta_3 \circ \theta_2 \circ \theta_1)t : [A_{dec}](\theta_3 \circ \theta_2 \circ \theta_1)\sigma\}, \theta_2)\end{array}}\ \textsc{EInstObj}$$

$\boxed{\Psi \vdash A \rightsquigarrow \theta}$ A Instantiation

$$\frac{}{\Psi \vdash \bullet \rightsquigarrow \bullet}\ \textsc{AInstNil} \qquad \frac{\Psi \vdash A \rightsquigarrow \theta \qquad \Psi \vdash_{\mathtt{ty}} \tau}{\Psi \vdash A; \alpha \rightsquigarrow [\tau/\alpha] \circ \theta}\ \textsc{AInstCons}$$

Figure 3.8: Context instantiation

3.6 Context instantiation

Figure 3.8 displays the judgments for context instantiation, which are used to instantiate variables in contexts.

The top-level judgment of shape $\Psi_{alg} \rightsquigarrow \Psi_{dec}$ instantiates Ψ_{alg} into Ψ_{dec}. In Chapter 2 the types of in- and outputcontext were disjoint: the rules in Figure 2.7 take an algorithmic context as input, and produce an declarative context by instantiating all remaining holes (i.e. existential type variables). As the systems presented in this chapter share a grammar for contexts, the type of in- and output contexts are the same. What remains is the purpose of instantiation: to instantiate algorithmic context (which always try to assign the most general type possible) into more specific types. The top-level instantiation

is defined (rule EInst) in the intermediate context instantiation judgment, which in turn uses a helper judgment to instantiate As. Notable that the shape of Ψ_{alg} and Ψ_{dec} is the same: sequences A may change in length and members and terms and types may be substituted, but no element may ever be appended or removed from the context.

Intermediate context instantation of shape $(\Psi, \theta_1) \vdash \Psi_{alg} \rightsquigarrow (\Psi_{dec}, \theta_2)$ adds θ, which is the result of instantiating Ψ_{alg}, to the result. θ contains mappings from typte variables to types. The domain of θ matches the type variables present in Ψ_{alg}, and all types in the range of θ are well-formed w.r.t. Ψ_{dec}. Therefore, any type or term well-formed w.r.t. Ψ_{alg} that has θ applied to it is well-formed w.r.t. Ψ_{dec}.

Ψ_{in} and θ_{in} represent the result of instantiating some other (algorithmic) context. These allow for the formulation of the following two lemmas which greatly helps reasoning inductively over instantation judgments:

Lemma 3.6.1 (Context instantation splitting)**.** *If* $(\Psi_{in}, \theta_{in}) \vdash \Psi_{alg1}; \Psi_{alg2} \rightsquigarrow (\Psi_{dec}, \theta)$ *then* $\exists\ \Psi_{dec1}\ \Psi_{dec2}\ \theta_1\ \theta_2$ *such that* $\Psi_{dec} = \Psi_{dec1}; \Psi_{dec2}$, $\theta = \theta_2; \theta_1$, $(\Psi_{in}, \theta_{in}) \vdash \Psi_{alg1} \rightsquigarrow (\Psi_{dec1}, \theta_1)$ *and* $(\Psi_{in}; \Psi_{alg1}, \theta_1; \theta_{in}) \vdash \Psi_{alg2} \rightsquigarrow (\Psi_{dec2}, \theta_2)$.

Lemma 3.6.2 (Context instantation merging)**.** *If* $(\Psi_{in}, \theta_{in}) \vdash \Psi_{alg1} \rightsquigarrow (\Psi_{dec1}, \theta_1)$ *and* $(\Psi_{in}; \Psi_{alg1}, \theta_1; \theta_{in}) \vdash \Psi_{alg2} \rightsquigarrow (\Psi_{dec2}, \theta_2)$ *then* $(\Psi_{in}, \theta_{in}) \vdash \Psi_{alg1}; \Psi_{alg2} \rightsquigarrow (\Psi_{dec1}; \Psi_{dec2}, \theta_2; \theta_1)$.

As for the rules themselves, the helper judgment $\Psi \vdash A \rightsquigarrow \theta$ converts the input A to θ with a domain equal to A and a range well-formed w.r.t. Ψ. Its only trivial rule, rule AInstCons, generates a τ well-formed w.r.t. Ψ, and adds it to θ, to be mapped to α. The intention is that any further occurrence of α will be replaced by τ.

Rule EInstA utilizes this helper judgment to instantiate A_{alg}. The sequence A_{alg} is replaced by A_{dec} (maintaining the shape), and the substitutions threaded trough are extended with substitutions for $exEnv[alg]$. Rule EInstX utilizes the threaded trough substitution to instantiate σ. Note that σ is applied both θ_1 (the contextual substitution, reflecting a previously instantiated context that Ψ_{alg} would have been appended to) and θ_2 (the result of instantiating Ψ_{alg}).

Finally, rule EInstObj features elements from the previous two rules. Firstly, the A_{alg} included in the object is instantiated similar to what happens in rule EInstA. Then, term t and type σ are instantiated similar to what happens in rule EInstX. Finally, the instantiated object is returned, but crucially, *substitution* θ_3 *is dropped*. As it is the result of instantiating A_{alg}, and its scope does not extend the object, its resulting substation should not either.

3.7 Term equivalence

As already hinted at before, when dealing with elaborated terms it is necessary to reason about the equivalence of terms. For example, Lemma 3.4.5 only holds up to equivalence, because in order to move variables from the context to the list in scope, if the derivation contains a let binding, the same would have to be done for the derivation of e_1 (rule DecALet$_{\rightsquigarrow}$). This means that the term t_1 derived for expression e_1 will be generalized over a potentially bigger list of variables. This term will be more general, so the derivation is still possible. Yet, the term resulting term will not be equal to the original. However, it will be *equivalent* in the sense that it has the same behaviour.

Another example is given in Figure 3.9, where three derivations of a let binding are displayed. They differ in the type chosen for x: the top derivation assigns the type $\forall\alpha.\alpha \to \alpha$, the middle assigns $\forall\alpha.\forall\beta.(\alpha, \beta) \to (\alpha, \beta)$, and the bottom $\forall\alpha.(\alpha, \mathcal{B}) \to (\alpha, \mathcal{B})$. Since the algorithmic system always assigns the most general type possible, only the top one reflects the algorithmic derivation. Yet, in this case all selections are valid, resulting in equal types and *equivalent* terms.

The reason these problems show up only when extending the system with elaboration is that subsumption on types works by substitution, whereas for terms it works by abstraction and application. This difference makes it so that in both examples above and in general changes that affect the generality of types part of the derivation result in a derivation that derives an *equal* type, but *equivalent* term. The reason substitution is not possible for terms is that, at the time of subsumption, the actual term that has the to-be-instantiated type is not known yet. Instead, we have to work with a placeholder: a term variable. Substitutions on this are nonsensical, because the type variables substitution attempts to replace will only be present *after* the placeholder has been replaced with a concrete term, i.e. in the case of application (rule DecAApp$_{\rightsquigarrow}$) or let bindings (rule DecALet$_{\rightsquigarrow}$).

When the concrete term has been supplied, the combination of abstraction and application results in the desired substitutions that in the case of types already happen during subsumption. However, since consuming the abstractions and applications reduces the term, these substitutions now happen as a result of evaluation! To reason about classes of terms that “evantually” evaluate to equal terms utilize logical relations.

$$\dfrac{\dfrac{\dfrac{[\alpha]\#\Psi \qquad [\alpha] \vdash_{\mathtt{ty}} \alpha}{[\alpha]; x : \alpha \vdash x : [\bullet]\alpha \rightsquigarrow x}}{\bullet \vdash \lambda x.\, x : [\alpha]\alpha \to \alpha \rightsquigarrow \lambda(x : \alpha).\, x} \qquad \dfrac{f : \forall\alpha.\alpha \to \alpha \vdash f : [\bullet](1, \mathcal{B}) \to (1, \mathcal{B}) \rightsquigarrow f\,[(1, \mathcal{B})]}{f : \forall\alpha.\alpha \to \alpha; \bullet \vdash ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow ((), \top)}}{f : \forall\alpha.\alpha \to \alpha \vdash f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow f\,[(1, \mathcal{B})]\ ((), \top)}$$

$$\bullet \vdash \mathbf{let}\ f = \lambda x.\, x\ \mathbf{in}\ f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow (\Lambda\alpha.\, \lambda(x : \alpha).\, x)\,[(1, \mathcal{B})]\ ((), \top)$$

$$\dfrac{\dfrac{\dfrac{[\alpha;\beta]\#\Psi \qquad [\alpha;\beta] \vdash_{\mathtt{ty}} (\alpha, \beta)}{[\alpha;\beta]; x : (\alpha, \beta) \vdash x : [\bullet](\alpha, \beta) \rightsquigarrow x}}{\bullet \vdash \lambda x.\, x : [\alpha;\beta](\alpha, \beta) \to (\alpha, \beta) \rightsquigarrow \lambda(x : (\alpha, \beta)).\, x} \qquad \dfrac{f : \forall\alpha.\forall\beta.(\alpha, \beta) \to (\alpha, \beta) \vdash f : [\bullet](1, \mathcal{B}) \to (1, \mathcal{B}) \rightsquigarrow f\,[1]\,[\mathcal{B}]}{f : \forall\alpha.\forall\beta.(\alpha, \beta) \to (\alpha, \beta); \bullet \vdash ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow ((), \top)}}{f : \forall\alpha.\forall\beta.(\alpha, \beta) \to (\alpha, \beta) \vdash f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow f\,[1]\,[\mathcal{B}]\ ((), \top)}$$

$$\bullet \vdash \mathbf{let}\ f = \lambda x.\, x\ \mathbf{in}\ f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow (\Lambda\alpha.\, \Lambda\beta.\, \lambda(x : (\alpha, \beta)).\, x)\,[1]\,[\mathcal{B}]\ ((), \top)$$

$$\dfrac{\dfrac{\dfrac{[\alpha]\#\Psi \qquad [\alpha] \vdash_{\mathtt{ty}} (\alpha, \mathcal{B})}{[\alpha]; x : (\alpha, \mathcal{B}) \vdash x : [\bullet](\alpha, \mathcal{B}) \rightsquigarrow x}}{\bullet \vdash \lambda x.\, x : [\alpha;\beta](\alpha, \beta) \to (\alpha, \beta) \rightsquigarrow \lambda(x : (\alpha, \beta)).\, x} \qquad \dfrac{f : \forall\alpha.(\alpha, \mathcal{B}) \to (\alpha, \mathcal{B}) \vdash f : [\bullet](1, \mathcal{B}) \to (1, \mathcal{B}) \rightsquigarrow f\,[1]}{f : \forall\alpha.(\alpha, \mathcal{B}) \to (\alpha, \mathcal{B}); \bullet \vdash ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow ((), \top)}}{f : \forall\alpha.(\alpha, \mathcal{B}) \to (\alpha, \mathcal{B}) \vdash f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow f\,[1]\ ((), \top)}$$

$$\bullet \vdash \mathbf{let}\ f = \lambda x.\, x\ \mathbf{in}\ f\ ((), \top) : [\bullet](1, \mathcal{B}) \rightsquigarrow (\Lambda\alpha.\, \lambda(x : (\alpha, \mathcal{B})).\, x)\,[1]\,[\mathcal{B}]\ ((), \top)$$

Figure 3.9: Three equivalent derivations

$\rho_1,\ \rho_2,\ \rho_R,\ \gamma_1,\ \gamma_2$ Projections

$$
\begin{aligned}
Rel[\tau_1,\tau_2] =^1\ & \{R \in \mathcal{P}(Val \times Val) \mid \forall (v_1,v_2) \in R.\ \bullet \Vdash v_1 : \tau_1 \wedge \bullet \Vdash v_1 : \tau_2\} \\
\mathcal{V}[\![\sigma]\!]_\rho =^2\ & \{(v_1,v_2) \mid \bullet \Vdash v_1 : \sigma \wedge \bullet \Vdash v_2 : \sigma \wedge \ldots\} \\
\mathcal{V}[\![\alpha]\!]_\rho =^3\ & \{(v_1,v_2) \mid (v_1,v_2) \in \rho_R(\alpha)\} \\
\mathcal{V}[\![1]\!]_\rho =^4\ & \{((),())\} \\
\mathcal{V}[\![\mathcal{B}]\!]_\rho =^5\ & \{(\top,\top),(\bot,\bot)\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho =^6\ & \{(\lambda(x:\rho_1(\tau_1)).\,t_1, \lambda(x:\rho_2(\tau_1)).\,t_2) \mid \forall (v_1,v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho. \\
& \qquad ([v_1/x]t_1, [v_2/x]t_2) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\forall\alpha.\sigma]\!]_\rho =^7\ & \{(\Lambda\alpha.\,t_1, \Lambda\alpha.\,t_2) \mid \forall \tau_1,\tau_2,R \in Rel[\tau_1,\tau_2].\ \bullet \vdash_{\mathtt{ty}} \tau_1 \wedge \bullet \vdash_{\mathtt{ty}} \tau_2 \\
& \qquad \wedge (t_1,t_2) \in \mathcal{V}[\![\sigma]\!]_{\alpha \mapsto (\tau_1,\tau_2,R);\rho}\} \\
\mathcal{E}[\![\sigma]\!]_\rho =^8\ & \{(t_1,t_2) \mid \bullet \Vdash t_1 : \sigma \wedge \bullet \Vdash t_2 : \sigma \wedge \exists v_1,v_2.\, t_1 \mapsto^* v_1 \wedge \\
& \qquad t_2 \mapsto^* v_2 \wedge (v_1,v_2) \in \mathcal{V}[\![\forall\alpha.\sigma]\!]_\rho\} \\
\mathcal{G}[\![\bullet]\!] =^9\ & \{(\emptyset,\emptyset)\} \\
\mathcal{G}[\![\Psi; x:\sigma]\!] =^{10}\ & \{(x \mapsto (v_1,v_2);\gamma,\rho) \mid (\gamma,\rho) \in \mathcal{G}[\![\Psi]\!] \wedge x \notin dom(\gamma) \wedge \\
& \qquad (v_1,v_2) \in \mathcal{V}[\![\sigma]\!]_\rho\} \\
\mathcal{G}[\![\Psi;\bullet]\!] =^{11}\ & \{(\gamma,\rho) \mid (\gamma,\rho) \in \mathcal{G}[\![\Psi]\!]\} \\
\mathcal{G}[\![\Psi; A; \alpha]\!] =^{12}\ & \{(\gamma, \alpha \mapsto (\tau_1,\tau_2,R);\rho) \mid (\gamma,\rho) \in \mathcal{G}[\![\Psi]\!] \wedge \bullet \vdash_{\mathtt{ty}} \tau_1 \wedge \bullet \vdash_{\mathtt{ty}} \tau_2 \\
& \qquad \wedge R \in Rel[\tau_1,\tau_2]\}
\end{aligned}
$$

Figure 3.10: Logical relations

3.7.1 Logical relations

Logical relations are a proof technique that revolves around an inductively defined relation over a number of terms, two (binary) terms or more, set up such that from membership of this relation interesting properties can be inferred. For example, strong normalization can be proven by means of an unary logical relation. In this chapter however we employ a binary logical relation that relates two terms if they are equivalent, *given equivalent arguments*. This last part is crucial, because as established, the substitutions that settle the difference between terms when generalizing the type of an argument in the context only happens as part of the operational semantics. In order for the reduction to happen, arguments must be supplied.

Figure 3.10 displays our logical relations. It is more general than simple equivalence: it parametrizes over the exact relation and instead relates terms given related arguments. The two main logical relations $\mathcal{V}[\![-]\!]_-$ and $\mathcal{E}[\![-]\!]_-$

are defined in terms of each other, and define when two closed values and expressions, respectively, are related.

The rules in Figure 3.10 are based from Skorstengaard's summary [73] of Achmed's lectures on logical relations [2], adapted to fit our shape of contexts Ψ (which includes the scope of type variables in Ψ instead of having a separate Δ) and our HDM types σ (instead of the non-stratified System F types). Since we base ourselves on (not peer reviewed) lectures, we verify its soundness in Section 3.7.2.

Equation 2 asserts that both values are closed and of the parameter type. Equation 2 is a superrule, meaning that it is meant to be prepended to all other equations of $\mathcal{V}[\![-]\!]_-$.

The rules for unit 1 (equation 4) and booleans $\mathcal{B}$ (equation 5) are base cases and straightforward, and simply relate equal values. Two terms of type $\tau_1 \to \tau_2$ are related (equation 6) firstly if they are abstractions abstracting over the same term variable, annotated with the left and right substitutions in ρ, respectively (by means of the $-_1$ and $-_2$ projections). Secondly, their bodies, when supplied arguments of the argument type (i.e. values related by $\mathcal{V}[\![\tau_1]\!]_\rho$) must be expressions of the return type (i.e. expressions related by $\mathcal{E}[\![\tau_2]\!]_\rho$).

Both $\mathcal{V}[\![-]\!]_-$ and $\mathcal{E}[\![-]\!]_-$ are indexed by a ρ. This ρ encapsulates the related *when given related arguments* aspect by containing mappings of shape $\alpha \mapsto (\tau_1, \tau_2, R)$. The intention here is that α is substituted by τ_1 and τ_2 in the left and right related term, respectively, and that $R \in Rel[\tau_1, \tau_2]$. R being in $Rel[\tau_1, \tau_2]$ means (equation 1) that R relates values, and that for any pair of values related by R the left value has type τ_1, and the right τ_2. This way it is ensured that whichever terms are supplied as arguments have a consistent type (since a binding for α in ρ is unique) whilst still parametrising over the exact way they are related. When an universal quantifier is encountered the indexed ρ must be extended with a binding for this quantifier. As can be seen (equation 7), the logical relation admits any relation R as long as it is in $Rel[\tau_1, \tau_2]$. Therefore, two type abstractions are related if their bodies are under the extended ρ. The rule for type variables α (equation 3) utilizes the indexed ρ by looking up the relation corresponding to α in ρ (by means of the $-_R$ projection) and asserting any values in this relation are related values.

The relation $\mathcal{E}[\![\sigma]\!]_\rho$ is defined by a singular equation (equation 8), and states that two terms are related if they both have the parameter type and step to values in $\mathcal{V}[\![\sigma]\!]_\rho$. The formalisation inlines equation 8 in equations 6 and 7 avoiding a mutually recursive definition.

Finally, the $\mathcal{G}[\![-]\!]$ relation, while at first glance looking like a binary relation, is actually a unary relation over a pair (γ, ρ), and combines the $\mathcal{G}$ and $\mathcal{D}$ relations

from [73]. The second component, ρ, is used to substitute type variables and explained above. The first component, γ, contains mappings for *term variables*, of shape $x \mapsto (v_1, v_2)$. However, instead of relating v_1 and v_2 with some relation R to be freely chosen, we instead fix the relation to be the value logical relation $\mathcal{V}[\![-]\!]_-$ defined above. Together, when $(\gamma, \rho) \in \mathcal{G}[\![\Psi]\!]$, γ and ρ can be used to close off terms well-formed w.r.t. Ψ, which is used in our definition of Logical Equivalence below. Equations 9 and 11 deal with empty contexts and context with an empty A as head. Equations 10 and 12 extend γ with a binding for x and α, respectively, in the expected way.

3.7.2 Logical Equivalence

We can now define Logical Equivalence, taken directly from [73], adapted in the expected way.

Definition 3.7.1 (Logical Equivalence)**.** *Two (open) terms t_1 and t_2 are logically equivalent at type σ under Ψ (denoted as $\Psi \Vdash t_1 \approx t_2 : \sigma$) if they (1) both have type σ under Ψ and (2) are, when closed off by ρ and γ, related by $\mathcal{E}[\![\sigma]\!]_\rho$, for all $(\gamma, \rho) \in \mathcal{G}[\![\Psi]\!]$:*

$$\Psi \Vdash t_1 \approx t_2 : \sigma := \\ \quad \Psi \Vdash t_1 : \sigma \wedge \Psi \Vdash t_2 : \sigma \wedge \forall (\gamma, \rho) \in \mathcal{G}[\![\Psi]\!].\, (\rho_1(\gamma_1(t_1)), \rho_2(\gamma_2(t_2))) \in \mathcal{E}[\![\sigma]\!]_\rho$$

Based on this, we can give the Fundamental Property, which states that if a term t can be typed at type σ under Ψ, it is equivalent with itself at σ under Ψ.

Theorem 3.7.2 (Fundamental Property)**.** *$\Psi \Vdash t : \sigma$ then $\Psi \Vdash t \approx t : \sigma$.*

Soundness of the Logical Relation

We prove the soundness of the logical relation by showing it implies contextual equivalence. We base its definition Skorstengaard's [73]. However, we found Skorstengaard's typing of holes lacklustre and not powerful enough. Therefore, we define a typing judgment per production of C, displayed in Figure 3.11.

Using this hole typing we can define contextual equivalence.

Definition 3.7.3 (Contextual Equivalence)**.** *Two open terms t_1 and t_2 are contextually equivalent at some type σ under Ψ (denoted as $\Psi \Vdash t_1 \approx t_2 : \sigma$) if they (1) both have type σ under Ψ and (2) are, when closed off by ρ and γ, related by $\mathcal{E}[\![\sigma]\!]_\rho$, for all $(\gamma, \rho) \in \mathcal{G}[\![\Psi]\!]$:*

$$\Psi \Vdash t_1 \approx^{ctx} t_2 : \sigma := \forall C : (\Psi \Vdash \tau_1) \Longrightarrow (\bullet \Vdash \tau_2).\, \exists v. t_1 \mapsto^* v \iff t_2 \mapsto^* v$$

$\boxed{C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \sigma_2)}$ Hole typing

$$\frac{}{\bullet : (\Psi \Vdash \sigma) \Longrightarrow (\Psi \Vdash \sigma)}\ \textsc{CTyHole}$$

$$\frac{\Psi_2 \vdash_{\mathtt{ty}} \tau_1 \qquad C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2; x : \tau_1 \Vdash \tau_2)}{\lambda(x : \tau_1).\, C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \tau_1 \to \tau_2)}\ \textsc{CTyAbs}$$

$$\frac{C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \tau_1 \to \tau_2) \qquad \Psi_2 \Vdash t : \tau_1}{C\ t : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \tau_2)}\ \textsc{CTyAppL}$$

$$\frac{\Psi_2 \Vdash t : \tau_1 \to \tau_2 \qquad C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \tau_1)}{t\ C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \tau_2)}\ \textsc{CTyAppR}$$

$$\frac{C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2; (\alpha) \Vdash \sigma_2)}{\Lambda\alpha.\, C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \sigma_2)}\ \textsc{CTyTAbs}$$

$$\frac{\Psi_2 \vdash_{\mathtt{ty}} \tau \qquad C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \forall\alpha.\sigma_2)}{C\,[\tau] : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash [\tau/\alpha]\sigma_2)}\ \textsc{CTyTApp}$$

Figure 3.11: Hole typing

Based on this we can now define the Soundness of the Logical Relation, which states that Logical Equivalence implies Contextual Equivalence. While this seems obvious (and the proof is relatively straightforward, given the Fundamental Property and the compatibility lemmas used to prove the Fundamental Property), it is useful to verify nevertheless.

Theorem 3.7.4 (Soundness of the Logical Relation)**.** *If* $\Psi \Vdash t_1 \approx t_2 : \sigma$ *then* $\Psi \Vdash t_1 \approx^{ctx} t_2 : \sigma$.

3.8 Correctness

Like in Chapter 2, we prove the algorithm correct in two parts, by first proving soundness, and then completeness. The proof is inspired by both the mechanisation Chapter 2 as well as Vaughan's paper proof [84] of a type inference algorithm for the HDM system.

Since the closed forms of soundness and completeness are corollaries of the (more interesting) their open forms, in this section we only present the open forms.

3.8.1 Soundness

Our definition of Soundness of the elaboration algorithm below (Theorem 3.8.1) is an extension of the (open) soundness statement in Chapter 2 (Lemma 2.5.4). Since we have inlined the polymorphic judgment in the rule for let bindings (as proposed in Section 2.8), there is no need for mutual induction. Apart from that, the extension is straightforward.

Theorem 3.8.1 (Soundness of the elaboration algorithm)**.** *If* $\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}$*,* $\mathtt{wf}(\Psi_{in})$*, and* $\Psi_{out}; \{t : [A]\tau\} \rightsquigarrow \Psi_{dec}; \{t_{dec} : [A_{dec}]\tau_{dec}\}$*, then* $\Psi_{dec} \vdash e : [A_{dec}]\tau_{dec} \rightsquigarrow t_{dec}$*.*

3.8.2 Completeness

Before we present our formulation of Completeness of the elaboration algorithm, we must discuss environment weakening. Specifically, we formulate the following lemma that deals with replacing the type assigned to a term variable in the context with a more general one.

Lemma 3.8.2 (Weakening of a binding in the declarative system)**.** *If* $\Psi; x : \sigma \vdash e : [A]\tau \rightsquigarrow t_1$ *and* $\Psi \vdash t_2' : \sigma' \geq t_2 : \sigma$ *then there exists* A' τ' t_1' *such that* $\Psi; x : \sigma' \vdash e : [A']\tau' \rightsquigarrow t_1'$*,* $\forall A'.\tau' = \forall A.\tau$*, and* $\Psi \Vdash \Lambda A'.\,[t_2'/x]t_1' \approx \Lambda A.\,[t_2/x]t_1 : \forall A.\tau$*.*

As discussed in Section 3.7 (and displayed in Figure 3.9), generalizing the type of a binding in the environment impacts the elaborated term. Where types are instantiated by means of substitution, terms are instantiated by means of *(type) application*. As established, this results in an equivalent, but not equal term. This is what Lemma 3.8.2 captures: generalizing σ to σ' in a derivation of e resulting in type τ and term t_1 with A in scope results in an alternate derivation. Since σ' may contain variables that occur in A, the alternate derivation generates A'. The alternate type, τ', is equal to the original τ when both are closed over their respective list in scope. Finally, the original term t_1, when supplied with t_2, is equivalent to the alternate term t_1' when supplied with t_2'. The subsumption judgment forms the final piece of the puzzle, connecting t_2 with t_2'.

This lemma also motivates the need for the alternative declarative system as described in Section 3.4.1. The terms and types "generated" by weakening are only related once closed off by their respective lists in scope. This requires being able to determine what this list in scope is, which is why we need to keep track of it explicitly (as motivated by the previous chapter).

Given an understanding of logical equivalence, and the need for and definition of weakening, our completeness statement below (Theorem 3.8.3) is a straightforward extension of Lemma 2.5.7.

Theorem 3.8.3 (Completeness of the elaboration algorithm)**.** *If* $\Psi_{dec} \vdash e : [A_{dec}]\tau_{dec} \rightsquigarrow t_{dec}$, $\mathtt{wf}(\Psi_{in})$, *and* $\Psi_{in} \rightsquigarrow \Psi_{dec}$, *then there exists* A'_{dec} τ'_{dec} t'_{dec} A_{alg} τ_{alg} t_{alg} Ψ_{out} *such that* $\Psi_{in} \vdash e : [A_{alg}]\tau_{alg} \rightsquigarrow t_{alg} \dashv \Psi_{out}$, $\Psi_{out}; \{t_{alg} : [A_{alg}]\tau_{alg}\} \rightsquigarrow \Psi_{dec}; \{t'_{dec} : [A'_{dec}]\tau'_{dec}\}$, $\forall A'_{dec}.\tau'_{dec} = \forall A_{dec}.\tau_{dec}$, *and* $\Psi_{dec} \Vdash \Lambda A'_{dec}.\, t'_{dec} \approx \Lambda A_{dec}.\, t_{dec} : \sigma$

3.9 Implementation

While we did not fully formalize soundness and completeness like in Chapter 2, we did formalize all definitions. The full implementation is available at `https://github.com/rogerbosman/hdm-elaboration`; we cover the most interesting aspects here.

3.9.1 Locally nameless & the Ott/LNgen stack

The formalisation of Chapter 2, which we build on, uses Ott [71] to convert the grammar and judgments, written in Ott's DSL, to both Coq and LaTeX. The Coq output uses the locally nameless representation, [14], and LNgen [3] is used to automatically generate a set of lemmas related to the locally nameless representation, and its interaction with for example the Ott-generated substitution and free variable functions. Since we build on this work, we have generally adopted the technology stack. However, as we already described in Chapter 2, while using this technology stack offers some benefits, the Ott DSL can be cumbersome to write in, and is not always as expressive as required. Therefore, additions tend to have been defined manually, especially additional judgments (since the main benefit of the Ott/LNgen stack lies in the datatypes).

3.9.2 Implementation of the logical relation in Coq

We encode the relations as given in Figure 3.10 as $\overline{X \rightarrow} Tm \rightarrow Tm \rightarrow Prop$, where $\overline{X \rightarrow}$ are the indexes of the logical relation, and Tm the type of terms. For example, the type of $\mathcal{V}$ is $Sc \rightarrow Rho \rightarrow Tm \rightarrow Tm \rightarrow Prop$, where Sc is the (coq) type of (object-level) polytypes, and Rho the (coq) type of (object-level) ρ.

Closed values

The closed values requirement of equation 2 is captured by the definition below.

```
Definition closed_vals (σ:Sc) (ρ:rho) (v1 v2:Tm) :=
    is_val v1
  ∧ is_val v2
  ∧ • ⊢t v1 : (π1 ρ) σ
  ∧ • ⊢t v2 : (π2 ρ) σ.
```

Value & Expression relation

The main logical relation is defined using the Equations library [74], in part because of its support for well-founded recursion. Every case of `logrel_val` below uses the `closed_vals` definition above to assert the values are closed and have the desired type. For sobriety, we will omit these lines below,

As a first example, the $\mathcal{V}[\![\mathcal{B}]\!]_\rho$ case looks as follows.

```
Equations logrel_val (σ:Sc) (ρ:rho) (t1 t2:Tm) : Prop :=
...
logrel_val (S_T T_Bool) ρ v1 v2 :=
    (v1 = t_True ∧ v2 = t_True)
  ∨ (v1 = t_False ∧ v2 = t_False).
```

As can be seen, `logrel_val` is defined over terms, which are then manually asserted to be values by usage of the `is_val` predicate, defined in the expected way. We have found this approach simpler to use than for example sigma types, which require constant (un)folding, which becomes cumbersome quickly. That being said, part of this perceived inconvenience can be explained by the authors' inexperience working with (and automating) sigma types.

Locally nameless logical relation Since terms and types are implemented locally namelessly, we need to convert the equations from Figure 3.10. We show the term and type abstractions here.

The first case, for term abstractions, is unsurprising, except for the argument to `logrel_exp`: since we receive $t1$ and $t2$ under a binder (`t_Lam`), we must open $t1$ and $t2$ before using them. However, we then immediately substitute $v1'$ and $v2'$ for x (see equation 6), which is equal to opening with $v1'$ and $v2'$ directly. Opening with

```
Equations logrel_val (σ:Sc) (ρ:rho) (t1 t2:Tm) : Prop :=
...
logrel_val (S_T (T_Fun τ1 τ2)) ρ v1 v2 :=
    ∃ t1 t2,
    v1 = t_Lam ((π1 ρ) τ1) t1
  ∧ v2 = t_Lam ((π2 ρ) τ1) t2
  ∧ ∀ (v1' v2':Tm),
    logrel_val (S_T τ1) ρ v1' v2'
  → logrel_exp τ2 ρ (open_Tm_wrt_Tm t1 v1')
      (open_Tm_wrt_Tm t2 v2')
```

For term abstractions, we observe that we do not have an type variable α. We require a variable, because we go under the universal binder (`S_Forall`), and we need a variable to extend ρ with. Therefore, we quantify over this variable co-finitely: when showing two terms are related (i.e. when using `logrel_val` as an introductory form), one can specify a "blacklist" L that α is not in. When deriving a property from the fact that two terms are related (i.e. when using `logrel_val` as a lamination form), one open the type and terms up with any α, as long as it is not in L.

```
Equations logrel_val (σ:Sc) (ρ:rho) (t1 t2:Tm) : Prop :=
...
logrel_val (S_Forall σ) ρ v1 v2 :=
    ∃ t1 t2,
    v1 = t_TLam t1
  ∧ v2 = t_TLam t2
  ∧ ∀ τ1 τ2 R, ∃ L, ∀ α,
      α ∉ L
    → ( • ⊢wft τ1
      ∧ • ⊢wft τ2
      ∧ Rel R τ1 τ2
      )
    → logrel_exp (open_Sc_wrt_T σ (T_Var_f α))
        (((τ1, τ2, R), α) :: ρ) (open_Tm_wrt_T t1 τ1)
        (open_Tm_wrt_T t2 τ2)
```

Well-founded recursion As can be seen in equation 8, $\mathcal{E}[\![-]\!]_-$ is defined in terms of $\mathcal{V}[\![-]\!]_-$ *at the same type*. This means that effectively in the implementation of the case for an universal quantifier `S_Forall` σ, we recurse not on the structurally smaller σ, but on `open_Sc_wrt_T` σ `(T_Var_f` α`)`. Since this is not a syntactic subterm (i.e. the method by which Coq can automatically

determine recursion to be well-founded) we have to manually show our recursion is well-founded.

The equations library lets us derive the well-founded `Subterm` relation for any inductive type. We do so for monotypes `T`. →

```
Equations Derive NoConfusion Subterm for T.
Definition well_founded_subterm : well_founded T_subterm.
  eapply well_founded_T_subterm. Qed.
(* well_founded_T_subterm comes from deriving Subterm for T *)
```

We then define the inductive type `foralls_or_subterm` on type schemes that either asserts that the left scheme has less universal quantifiers than the right, or that both types are monotypes, and the left is a subterm of the right.

```
Inductive foralls_or_subterm : Sc → Sc → Prop :=
  | decr_foralls : ∀ (σ1 σ2:Sc),
      lt (foralls σ1) (foralls σ2)
    → foralls_or_subterm σ1 σ2
  | decr_subterm : ∀ (τ1 τ2:T),
      T_subterm τ1 τ2
    → foralls_or_subterm (S_T τ1) (S_T τ2).
```

We then manually prove this type well-founded such that we can recurse over it for our logical relation.

Compositionality

The locally nameless representation also affects lemmas such as Compositionality listed below. For the most part, the translation from Skorstengaard [73] is straightforward.

Lemma 3.9.1 (Compositionality)**.** $(v_1, v_2) \in \mathcal{V}[\![\sigma^\tau]\!]_\rho \iff (v_1, v_2) \in \mathcal{V}[\![\sigma^\alpha]\!]_{\alpha \mapsto (\rho_1(\tau), \rho_2(\tau), \mathcal{V}[\![\tau]\!]_\rho);\rho}$

3.9.3 Soundness of the logical relation & Filling holes

The formulation of Logical Equivalence uses programs with holes C and a fill function that takes a program with a hole C and term t, and places t at the hole, yielding a term t' without holes. In our observation, this seems to be incompatible with a locally nameless representation.

To understand this, let us first look at the named case. If we have an program with a hole $\lambda(x : \tau).\bullet$ and we want to obtain the identity function $\lambda(x : \tau).x$, we can obtain this by filling in $(\lambda(x : \tau).\bullet)[x] = \lambda(x : \tau).x$, given that $(\lambda(x : \tau).C)[t] = \lambda(x : \tau).(C[t])$. Omitting clauses for the well-formedness of τ, we can type this program with a hole using the following rule.

$$\frac{C : (\Psi_1; x : \tau_1 \Vdash \sigma) \Longrightarrow (\Psi_2 \Vdash \tau_2)}{\lambda(x : \tau_1).C : (\Psi_1 \Vdash \sigma) \Longrightarrow (\Psi_2 \Vdash \tau_1 \to \tau_2)}$$

An obvious lemma about filling named holes would be that if we fill a hole with a type fitting the holes left-hand signature a term with the hole's right-hand signature is obtained:

Lemma 3.9.2. *If* $C : (\Psi_1 \Vdash \sigma_1) \Longrightarrow (\Psi_2 \Vdash \sigma_2)$ *and* $\Psi_1 \Vdash t : \sigma_1$ *then* $\Psi_2 \Vdash C[t] : \sigma_2$

In the case of a locally nameless representation, the identity function looks like $\lambda\tau.0$. Replacing the body of the function with a hole yields $\lambda\tau.\bullet$. What should be filled into this program with a hole, to obtain the identity function? That depends on the fill function. If it does not alter the term (i.e. $(\lambda\tau.C)[t] = \lambda\tau.(C[t])$), then we should fill with the term 0: $(\lambda\tau.\bullet)[0] = \lambda\tau.0$. The problem with this is that the term 0 is not locally closed, and therefore not able to be typed, which makes typing holes impossible.

The solution is to forego the nameless aspect of programs with holes, and instead annotate a name that the to-be-filled term will be closed off with by the fill function: $(\lambda(x : \tau).C)[x] = \lambda\tau.({}^{\backslash x}(C[t]))$. Given this, we get that $(\lambda(x : \tau).\bullet)[x] = \lambda\tau.({}^{\backslash x}(\bullet[x])) = \lambda\tau.({}^{\backslash x}x) = \lambda\tau.0$.

This does mean however that our holes are not typed locally namelessly, and instead fully named, shown as the `CTy_Abs` constructor of `CTy` below.

```
Inductive CTy : Cont → E → Sc → E → Sc → Prop :=
  ...
  | CTy_Abs :
    ∀ (C:Cont) (Ψin:E) (σin:Sc) (Ψ:E) (τ1:T) (τ2:T) (x:var)
      (WFT: (Ψ  ⊢wfτ τ1))
      (CT: CTy C Ψin σin (E_Var Ψ x (S_T τ1)) (S_T τ2)),
    CTy (C_Lam x τ1 C) Ψin σin Ψ (S_T (T_Fun τ1 τ2))
```

This is inherently incompatible with a cofinite locally nameless interpretation of rule TmAbs (Figure 3.3), shown as the `TmTy_Abs` constructor of `TmTy_Abs` below.

```
Inductive TmTy : E → Tm → Sc → Prop :=
  ...
  | TmTy_Abs :
    ∀ (L:vars) (Ψ:E) (τ1:T) (t:Tm) (τ2:T)
      (WFT: (Ψ ⊢wfτ τ1))
      (TMTY: (∀ x, x ∉ L → TmTy (E_Var Ψ x (S_T τ1))
       (open_Tm_wrt_Tm t (t_Var_f x)) (S_T τ2))),
    TmTy Ψ (t_Lam τ1 t) (S_T (T_Fun τ1 τ2))
```

Since the proof of Lemma 3.9.2, which inducts over the given program hole, consists of essentially replacing the constrictors of `CTy` with `TmTy`, we run into problems when dealing with cofinitely quantified variables. Here, `TmTy_Abs` requires something stronger than what is given by the induction hypothesis: `TmTy_Abs` requires the inner term be a valid derivation the context extended with any variable not in `L`, whereas the induction hypothesis only gives this derivation for a specific variable. The solution is to define an existentially quantified version of `TmTy`, shown as `TmTy'` below, which corresponds with `TmTy`.

```
Inductive TmTy' : E → Tm → Sc → Prop :=
  ...
  | TmTy_Abs' :
    ∀ (Ψ:E) (τ1:T) (t:Tm) (τ2:T) (x:var)
      (WFT: (Ψ ⊢wfτ τ1))
      (TMTY:  TmTy' (E_Var Ψ x (S_T τ1)) t (S_T τ2)),
    TmTy' Ψ (t_Lam τ1 (close_Tm_wrt_Tm x t)) (S_T (T_Fun τ1 τ2))
```

3.10 Conclusion & Future Work

In this chapter we have incorporated elaboration to System F into the fully grounding type inference algorithm for the Hindley- Damas-Milner system of Chapter 2. The main changes to the algorithm are to return an elaborated term alongside the type information, and to include target terms in the context objects in order to benefit from substitutions. To our knowledge this is the first formalised type-inference-with-elaboration algorithm of its kind.

The most immediate avenue for future work is the actual mechanisation of the proofs that were sketched in this chapter. The authors have already made significant efforts towards this goal, but much work still remains to be done. The locally nameless representation especially caused significant complications. Even though the representation promises to lighten the need for substitution

lemmas and other freshness constraints, we conjecture that a fully nameless representation might be the more suitable for this task, but this remains to be demonstrated.

A subsequent research direction would be to extend our algorithm with additional elaboration-based language features such as typeclasses. A fully formalised compiler for a language like Haskell (along the lines of CompCert for C [43]) would be the ultimate goal. So far, we have only laid the groundwork, and we expect many challenges will have to be overcome to make this a reality, even for the more limited and well-defined Haskell 98 standard.

Part II

Scoped Effects

Chapter 4

A Calculus For Scoped Effects & Handlers

4.1 Introduction

While monads [51, 52, 89] have long been the go-to approach for modelling effects, *algebraic effects & handlers* [59, 58] are steadily gaining more traction. They offer a more structured and modular approach to composing effects, based on an algebraic model. The approach consists of two parts: effects denote the syntax of operations, and handlers interpret them by means of structural recursion. By composing handlers that each interpret only a part of the syntax in the desired order, one can modularly build an interpretation for the entire program. Algebraic effects & handlers have been adopted in several libraries (e.g., fused-effects [66], extensible-effects [38], Eff in OCaml [39]) and languages (e.g., Eff [63], Links [30], Koka [42], Effekt [12]).

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not always applicable. In particular, those effects that have or introduce a delimited scope (e.g., exceptions, concurrency, local state) are not algebraic. Essentially, these so-called *scoped effects* [93] split the program into two: a part *in* the scope of the effect (called the scoped computation) and a part *out* of the scope (called the continuation).

This separation breaks algebraicity, which states that operations commute with sequencing. Modeling scoped effects as handlers [59] has been proposed as a way of encoding scoped effects in an algebraic framework. However, this comes

at the cost of modularity [98] (and thus expressiveness). Wu et al. proposes to treat scoped effects as separate, built-in operations, and present their solution in a denotational setting. However, their solution of modular composition of scoped handlers is rather ad-hoc, and not a generalization of the algebraic case. Therefore, a treatment of scoped effects & handlers in an operational setting with full support for modular composition is desirable. The growing interest in scoped effects & handlers, evidenced by their adoption at GitHub [82] and in Haskell libraries (e.g., eff [37], polysemy [48], fused-effects [66]), motivates the need for such a calculus, all the more because they use the same ad-hoc approach of modular composition as suggested by Wu et al.

This chapter aims to fill this gap in the literature: we cover scoped effects & handlers, which in previous work has only been covered in a denotational setting, in an operational setting by presenting λ_{sc}, a calculus with scoped effects & handlers. Our main source of inspiration is Eff [5, 6, 63], a calculus for algebraic effects & handlers, effectively easing programming with those features. Although Eff is an appropriate starting point, the extension to support scoped effects & handlers is non-trivial, for two reasons. First, scoped effects require polymorphic handlers, which we support by let-polymorphism and F_ω-style type operators. Second, we need to be able to forward unknown operations in order to keep the desired modularity. Whereas algebraic effects & handlers have a uniform (and implicit) forwarding mechanism, scoped effects & handlers need explicit forwarding clauses in order to allow sufficient freedom in their implementation.

In what follows, after introducing the appropriate background (Section 4.2) and informally motivating the challenges and design choices of our calculus (Section 4.3), we formalize λ_{sc}. We make the following contributions:

- We present our solution for the modular composition of scoped handlers, called forwarding (Section 4.3.3), and generalize it later (Section 4.7).
- We design formal syntax for λ_{sc} terms, types and contexts (Section 4.4).
- We provide an operational semantics (Section 4.5).
- We define a type-and-effect system for λ_{sc} (Section 4.6).
- We formulate and prove λ_{sc}'s metatheoretical properties (Section 4.8).
- We show the usability of our calculus on a range of examples (Section 4.9).
- We provide an interpreter of our calculus with type inference in which we implement all our examples (supplementary material).

4.2 Background & Motivation

This section provides the necessary background and motivates our goal. We review *algebraic effects & handlers* as a modular approach to composing side-effects in effectful programs. Next, we present *scoped effects & handlers*: effects that have or create a delimited scope (such as `once` for nondeterminism [56, 93]), and motivate the need for a calculus with built-in support for these scoped effects.

4.2.1 Algebraic effects & handlers

Algebraic effects & handlers consist of *operations*, denoting their syntax, and *handlers*, denoting their semantics. This separation gives us modular composition, which has intrinsic value *and* allows controlling effect interaction.

Algebraic Operations

Effects are denoted by a name (or *label*) and characterized by a *signature* $A \twoheadrightarrow B$, taking a value of type A and producing a value of type B. For example, $\texttt{choose} : () \twoheadrightarrow \mathsf{Bool}$ takes a unit value and produces a boolean (e.g., nondeterministically). *Operations* invoke effects, combining the **op** keyword, an effect to invoke, a *parameter* passed to the effect, and a *continuation*, containing the rest of the program. For brevity of presentation, we follow the syntax of Eff [63] to have an explicit continuation for every operation.

$c_{\mathsf{ND1}} = \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)$

In accordance with its signature, `choose` is passed (), and in the supplied continuation b has type Bool. As a result, c_{ND1} is a computation that returns either 1 or 2.

Some operations commute with sequencing. For example:

$$\begin{aligned} &\mathbf{do}\ x \leftarrow \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)\,;\mathbf{return}\ x^2 \\ \equiv\ &\mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{do}\ x \leftarrow \mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2\,;\mathbf{return}\ x^2) \end{aligned}$$

This equivalence is an instance of the *algebraicity property*, and operations are *algebraic* if they satisfy this property. Algebraicity states that the sequencing of a computation c_2 after an operation $\mathbf{op}\ \ell\ v\ (y\,.\,c_1)$ is equivalent to sequencing the same computation after the *continuation* of this operation:

$$\mathbf{do}\ x \leftarrow \mathbf{op}\ \ell\ v\ (y\,.\,c_1)\,;c_2\ \equiv\ \mathbf{op}\ \ell\ v\ (y\,.\,\mathbf{do}\ x \leftarrow c_1\,;c_2)$$

Handlers

Handlers give meaning to operations. For example, handler h_{ND} interprets `choose` by collecting all the choices in a list:

$$\begin{aligned} h_{\mathsf{ND}} = \mathbf{handler}\,\{&\mathbf{return}\ x \mapsto \mathbf{return}\ [x] \\ ,&\mathbf{op}\ \texttt{choose}\ \_\ k \mapsto \mathbf{do}\ xs \leftarrow k\ \mathsf{true}\,;\mathbf{do}\ ys \leftarrow k\ \mathsf{false}\,;xs +\!\!+ ys\} \end{aligned}$$

This handler has two clauses. The first clause returns a singleton list in case a value x is returned. The second clause, which interprets `choose`, executes both branches by applying the continuation k to both true and false, and concatenates their resulting lists with the $(+\!\!+)$-operator. We apply h_{ND} to c_{ND1} to obtain both of its results:

$$\begin{aligned} &\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}} \\ \rightsquigarrow^*\ &[1,2] \end{aligned}$$

The algebraicity property plays a critical role in giving semantics to algebraic effects and handlers. For example, consider the following program which uses h_{ND} to handle a computation that sequences two `choose` operations using the **do** syntax.

$$c_{\mathsf{ND2}} = \mathbf{do}\ p \leftarrow \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\,;\mathbf{op}\ \texttt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))$$

In order to handle the first `choose` operation, the handler h_{ND} needs to get access to the continuation of it. Thus, we use the algebraicity property of `choose` to put the second `choose` operation in the explicit continuation of the first `choose` operation.

$$\begin{aligned} &\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND2}} \\ \rightsquigarrow\ &\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{op}\ \texttt{choose}\ ()\ (b\,. \\ &\quad \mathbf{do}\ p \leftarrow \mathbf{return}\ b\,;\mathbf{op}\ \texttt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))) \\ \rightsquigarrow^*\ &[(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false}),(\mathsf{false},\mathsf{true}),(\mathsf{false},\mathsf{false})] \end{aligned}$$

Algebraic effects & handlers bring several interesting advantages. Most interestingly, their separation of syntax and semantics allows a modular composition of different effects, which in turn allows for altering the meaning of a program by different effect interactions.

For those familiar with the category-theoretical backings of folds, handler application can be seen as a fold over an abstract syntax tree, where operations are nodes in that tree, and handlers are the fold algebra. The result type of the fold is the carrier of the algebra.

Modular Composition

Effects can be composed by combining different primitive operations. For example, computation $c_{\mathsf{c,g}}$ below uses $\texttt{get} : () \twoheadrightarrow \mathsf{String}$ in addition to `choose`.

$$c_{\mathsf{c,g}} = \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ op}\ \texttt{get}\ ()\ (x\,.\,\mathbf{return}\ x))$$

Instead of having to write a handler for each combination of effects, algebraic effects & handlers allow us to write a handler specific for the effect `get`, and to compose it with the existing handler h_{ND}.

$$h_{\mathsf{get}} = \mathbf{handler}\ \{\mathbf{return}\ x \mapsto \mathbf{return}\ x, \mathbf{op}\ \texttt{get}\ \_\ k \mapsto k\ 2\}$$

When composing handlers $\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ (\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ c_{\mathsf{c,g}})$, h_{get} is applied first, and handles `get`. Since h_{get} does not contain a clause for `choose`, it leaves (we say *"forwards"*, see below) the `choose` operation to be handled by another handler. This forwarding behavior is key to the modular reuse and composition of handlers. Handler h_{ND} then takes care of the remaining effects.

$$\begin{aligned} & \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ (\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ c_{\mathsf{c,g}}) \\ \rightsquigarrow^* & \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}} \\ \rightsquigarrow^* & [1, 2] \end{aligned}$$

Forwarding

Critical to modular composition is the possibility to apply partial handlers, i.e. to apply handlers to computations where the handlers may only handle a subset of the effects present in the computations. As discussed in Section 4.2.1, handler application can be seen as a fold, where handlers form the folding algebra. Therefore, the partial algebra formed by partial handlers must be supplemented by a *forwarding* algebra that interprets the effects not handled by the handler such that they can be handled by another handler.

Generally, calculi for algebraic effects require that handler clauses leave computations as their output. The forwarding algebra must "reinterpret" computations into computations, which is essentially the identity function. Therefore, for these calculi, forwarding can be and is done uniformly, which means handlers do not need to give specialised forwarding clauses for effects they do not handle. For example, `choose` is forwarded by h_{get} by simply leaving the effect with the handled computation.

$$\begin{aligned} & \mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle\ op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b) \\ \rightsquigarrow\ & \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle\ return}\ b) \end{aligned}$$

In the calculus we will present we cannot forward generically, and instead utilize explicit forwarding clauses. This is not novel: it already shows up in the algebraic case when generalizing handler clauses when they may return types that are not computations. Since the return type of handlers corresponds to the carrier of the fold, when generalizing handler clauses like this we get a fold with a carrier type that is not a computation. In this case, we can no longer uniformly forward the unhandled operations. Instead, we must specify, per carrier, how to forward unhandled operations by means of an explicit forwarding clause [69]. Requirement of explicit forwarding algebras for algebraic effects and handlers often happens in libraries, but not in languages, because languages with primitive support for algebraic effects and handlers always require the carriers of handlers to be computations.

As stated, in the case of scoped effects, we are required to specify such an explicit forwarding clause even when the carrier is a computation. For now, we would like to observe that this forwarding clause is *not* a consequence of implementing scoped effects, but merely a consequence of moving away from a (very specific) instance of algebraic effects.

Effect Interaction

One of the valuable features of the modular composition of algebraic effects & handlers is that effects can interact differently by applying their handlers in a different order. Consider the effect $\texttt{inc} : () \multimap \mathsf{Int}$, which produces an (incremented) integer. The handler h_{inc} turns computations into state-passing functions.

$$\begin{aligned} h_{\mathsf{inc}} = \mathbf{handler}\,\{&\mathbf{return}\ x \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{return}\ (x, s)) \\ ,&\mathbf{op}\ \texttt{inc}\ \_\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ s' \leftarrow s + 1\,; \\ &\qquad\qquad\qquad\qquad\quad \mathbf{do}\ k' \leftarrow k\ s'\,; \\ &\qquad\qquad\qquad\qquad\quad k'\ s')\} \end{aligned}$$

The state s represents the current counter value. On every occurrence of `inc`, the incremented value is passed to the continuation twice: (1) for updating the counter value and (2) for returning the result of the operation. The latter is for the continuation and the former for serving the next `inc` operation. We use the folowing syntactic sugar:

$$run_{\mathsf{inc}}\ s\ c \ \equiv\ \mathbf{do}\ c' \leftarrow \mathbf{with}\ h_{\mathsf{inc}}\ \mathbf{handle}\ c\,;\,c'\ s$$

The computation c_{inc} combines `choose` and `inc`:

$$\begin{aligned} c_{\mathsf{inc}} = \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,&\mathbf{if}\ b\ \mathbf{then}\ \mathbf{op}\ \texttt{inc}\ ()\ (x\,.\,x + 5) \\ &\mathbf{else}\ \ \mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,y + 2)) \end{aligned}$$

When handling `inc` first, each `choose` branch gets the same initial counter value 0.

$$
\begin{aligned}
& \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}} \\
\rightsquigarrow^* & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle op}\ \mathtt{choose}\ ()\ (b\,.\\
& \quad \textbf{do}\ p' \leftarrow \textbf{if}\ b\ \textbf{then with}\ h_{\mathsf{inc}}\ \textbf{handle}\ (\textbf{op}\ \mathtt{inc}\ ()\ (x\,.\,x+5)) \\
& \qquad\qquad\qquad \textbf{else with}\ h_{\mathsf{inc}}\ \textbf{handle}\ (\textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,y+2))\,; \\
& \quad p'\ 0) \\
\rightsquigarrow^* & [(6,1),(3,1)]
\end{aligned}
$$

In contrast, when handling `choose` first, the counter value is threaded through the successive branches, showing that with algebraic effects & handlers, the manner in which effects interact can be controlled by the order in which handlers are applied.

$$
\begin{aligned}
& run_{\mathsf{inc}}\ 0\ (\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ c_{\mathsf{inc}}) \\
\rightsquigarrow^* & run_{\mathsf{inc}}\ 0\ (\textbf{do}\ xs \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle op}\ \mathtt{inc}\ ()\ (x\,.\,x+5)\,; \\
& \qquad\qquad \textbf{do}\ ys \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle op}\ \mathtt{inc}\ ()\ (y\,.\,y+2)\,; \\
& \qquad\qquad \textbf{return}\ xs +\!\!+ ys) \\
\rightsquigarrow^* & ([6,4],2)
\end{aligned}
$$

4.2.2 Scoped effects

Not all effects are algebraic. There is a wide class of higher-order effects [7] which do not satisfy the algebraicity property including scoped effects [93], latent effects [83], and parallel effects [96]. In this chapter, we focus on scoped effects. Consider extending nondeterminism with an effect that takes a computation that contains nondeterministic choice and returns only its first result. This is known as the `once` operation [56]. Since effect operations syntactically already receive the continuation as and argument, which looks like a computation, we might be tempted to use it for this purpose, i.e. to use it to denote the scope of `once`. Subscripting ✗ to indicate an erroneous example, we could attempt to syntactically write this as the algebraic operation $\mathtt{once}_{\boldsymbol{\times}} : () \twoheadrightarrow ()$ by extending h_{ND} with a clause for $\mathtt{once}_{\boldsymbol{\times}}$ that executes the continuation, and then takes the head of the result if possible. Otherwise it just returns the empty list. We then try to prune the first `choose` of c_{ND2} by using `once`.

$$
\begin{aligned}
h_{\mathsf{once}_{\boldsymbol{\times}}} = \textbf{handler}\ \{\ldots, \textbf{op}\ \mathtt{once}_{\boldsymbol{\times}}\ \_\ k \mapsto\ & \textbf{do}\ ts \leftarrow k\ ()\,; \\
& \textbf{do}\ b \leftarrow ts = [\,]\,; \\
& \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else}\ \mathsf{head}\ ts\}
\end{aligned}
$$

$$
\begin{aligned}
c_{\mathsf{once}_{\boldsymbol{\times}}} = \ & \textbf{do}\ p \leftarrow \textbf{op}\ \mathtt{once}_{\boldsymbol{\times}}\ ()\ (\_\,.\,\textbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\textbf{return}\ b))\,; \\
& \textbf{op}\ \mathtt{choose}\ ()\ (q\,.\,\textbf{return}\ (p,q))
\end{aligned}
$$

We intend for **with** $h_{\mathsf{once}\boldsymbol{\chi}}$ **handle** $c_{\mathsf{once}\boldsymbol{\chi}}$ to return $[(\mathsf{true}, \mathsf{true}), (\mathsf{true}, \mathsf{false})]$ as the first `choose` is pruned by $\mathtt{once}_{\boldsymbol{\chi}}$ to only return the first alternative. The second `choose` is out of scope of $\mathtt{once}_{\boldsymbol{\chi}}$, so should still return both results. However, algebraicity pulls the second `choose` inside the scope of $\mathtt{once}_{\boldsymbol{\chi}}$:

$$
\begin{aligned}
& \textbf{with } h_{\mathsf{once}\boldsymbol{\chi}} \textbf{ handle } (\textbf{do } p \leftarrow \textbf{op } \mathtt{once}_{\boldsymbol{\chi}} \ () \ (\_ \,.\, \textbf{op } \mathtt{choose} \ () \ (b \,.\, \textbf{return } b)) \,; \\
& \qquad\qquad\qquad\qquad \textbf{op } \mathtt{choose} \ () \ (q \,.\, \textbf{return } (p, q))) \\
\rightsquigarrow \ & \textbf{with } h_{\mathsf{once}\boldsymbol{\chi}} \textbf{ handle } \mathtt{once}_{\boldsymbol{\chi}} \ () \ (\_ \,.\, \textbf{do op } \mathtt{choose} \ () \ (b \,.\, \textbf{return } b) \,; \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{op } \mathtt{choose} \ () \ (q \,.\, \textbf{return } (p, q))) \\
\rightsquigarrow^* \ & [(\mathsf{true}, \mathsf{true})]
\end{aligned}
$$

There are many more examples of operations that have a scope; some of which we present in Section 4.9:

- `catch` for catching exceptions that are raised during program execution;
- `local` for creating local variables (local state);
- `call` for creating a scope in a nondeterministic program, where branches can be cut using the algebraic `cut` operation;
- `depth` for bounding the depth in the depth-bounded search strategy;

Following Wu et al. [93], we call them *scoped operations*. Plotkin and Power [59] have already realised that algebraic effects are unable to represent so-called effect deconstructors (e.g. scoped effects) and propose to model them as handlers. Although this solution is used (for example by Thompson et al. [82]), it merges syntax and semantics at the cost of modularity [93, 98]. In Section 4.9 we revisit this issue, showing attempts at encoding scoped effects as handlers, their problems, and how λ_{sc} remedies the situation.

The goal of this work is to implement scoped effects while maintaining a separation between syntax and semantics, and thus preserve modular composition and control over effect interaction. It follows a line of research [56, 93, 98] that has developed denotational semantic domains, backed by categorical models. However, they did not consider the modular composition of scoped effects and handlers in their denotional semantics. What is lacking from the literature is a calculus that allows programming with both algebraic and scoped operations and their handlers as well as supports the modular composition between them.

4.3 Design Decisions & Challenges

This section informally discusses the design of λ_{sc}, a novel calculus with support for scoped effects & handlers as built-in features. We present the main challenges and design choices, and address how to forward unknown operations.

4.3.1 Scoped Effects as Built-in Operations

λ_{sc} supports scoped effects as built-in operations, signalled by the **sc** keyword.

$\mathbf{sc}\ \mathtt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,c_2)$

Like algebraic operations, scoped operations feature a label `once` to identify the effect, a parameter, in this case (), and a continuation $(z\,.\,c_2)$. However, unlike algebraic operations, scoped operations feature an additional *scoped computation* $(y\,.\,c_1)$. For `once`, the scoped computation entails the computation to be restricted to the first result (i.e. the computation in scope). The result from the scoped computation c_1 is passed to the continuation c_2 by means of argument z.

Scoped handler clauses feature an additional argument when compared to algebraic handler clauses, corresponding to the scoped computation. For example, below we define a handler for `once`, which extends h_{ND}.

$$\begin{aligned} h_{\mathsf{once}} = \mathbf{handler}\ \{\ \ldots,\ & \\ \mathbf{sc}\ \mathtt{once}\ \_\ p\ k \mapsto\ & \mathbf{do}\ ts \leftarrow p\ ()\,; \\ & \mathbf{do}\ b \leftarrow ts = [\,]\,; \\ & \mathbf{if}\ b\ \mathbf{then\ return}\ [\,]\ \mathbf{else\ do}\ t \leftarrow \mathsf{head}\ ts\,;\, k\ t\} \end{aligned}$$

Adding scoped operations gives rise to a variant of the algebraicity property, which models the desired behavior of sequencing for scoped operations: scoped operations commute with sequencing in the continuation, but leave the scoped computation intact.

$$\mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\,;\,c_3\ \equiv\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,\mathbf{do}\ x \leftarrow c_2\,;\,c_3)$$

Using `once` as a scoped operation correctly restrict only the first `choose`, as can be seen when applying h_{once} to c_{once}, which is like $c_{\mathsf{once}\not\mathsf{x}}$ defined in Section 4.2.2 but with the scoped effect `once` instead of the algebraic effect $\mathtt{once}_{\not\mathsf{x}}$.

$$\begin{aligned} c_{\mathsf{once}} = \mathbf{sc}\ \mathtt{once}\ ()\ (\_\,.\,\mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)) & \\ (p\,.\,\mathbf{do}\ q \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\,;\mathbf{return}\ (p, q)) & \end{aligned}$$

$$\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle}\ c_{\mathsf{once}} \rightsquigarrow^{*} [(\mathsf{true}, \mathsf{true}), (\mathsf{true}, \mathsf{false})]$$

Signatures Scoped effects, like algebraic effects, have signatures $A \twoheadrightarrow B$. Like algebraic effects, the left-hand side of this signature refers to the type of the value passed to the operation (() in the case of `once`). However, where the right-hand side of the signature in the case of algebraic effects refers to the argument of the *continuation*, in the case of scoped effects it refers to the *scoped computation*. Whilst this may seem strange from this point of view, it makes sense from the point of view of writing handler clauses: in both cases, a value of type A is given, and a value of type B is produced by the handler clause. The difference comes from what computation this produced value of type B is passed to: algebraic effect handler clauses pass this value to the continuation, whereas scoped effect handler clauses pass this value to the scoped computation (whose result is passed to the continuation).

Therefore, the signature of `once` is $() \twoheadrightarrow ()$ since the handler clause calls the scoped computation with (). The scoped result type (Section 4.3.3) varies depending on the computation `once` is applied to, and is therefore not a part of the signature.

4.3.2 Eff with Row Typing

Our calculus is based on Eff [63, 5, 6], an existing calculus for algebraic effects & handlers. However, instead of using the subtyping-based effect system of Eff, we use a row-based effect system in the style of Koka [42] for simplicity. Therefore, computations have types of shape $A \,!\, \langle E \rangle$, where A is the type of the value returned by the computation, and E is a collection (row) of effects that *may* occur during its evaluation. For example, $(\mathsf{Bool}, \mathsf{Bool}) \,!\, \langle \texttt{choose}, \texttt{once} \rangle$ is the type of c_{once}.

Handler types, of shape $A \,!\, \langle E \rangle \Rightarrow B \,!\, \langle F \rangle$, reflect that handlers turn one computation (of type $A \,!\, \langle E \rangle$) into another (of type $B \,!\, \langle F \rangle$). For example, h_{once} handles `choose` and `once`.

$h_{\mathsf{once}} : (\mathsf{Bool}, \mathsf{Bool}) \,!\, \langle \texttt{choose}, \texttt{once} \rangle \Rightarrow \mathsf{List}\ (\mathsf{Bool}, \mathsf{Bool}) \,!\, \langle \rangle$

4.3.3 Scoped result type

As we have seen before, the two computations taken by a scoped operation $\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ are not just two irrelevant terms; they are connected in the sense that the result of the scoped computation $y\,.\,c_1$ is passed to the continuation $z\,.\,c_2$ and exactly bound as z in c_2. Therefore, they must agree on the type of this result, which we name the *scoped result type*. For example, consider

c_{once} with overall type $(\mathsf{Bool}, \mathsf{Bool})\,!\,\langle \mathtt{once}\,;\mathtt{choose}\rangle$. Its scoped result type is (a singular) Bool: it is the type that is *produced* by the scoped computation, and *consumed* by the continuation.

$$\mathbf{sc}\ \mathtt{once}\ ()\ (\underbrace{\_\,.\,\mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)}_{()\to \mathsf{Bool}\,!\,\langle \mathtt{once}\,;\mathtt{choose}\rangle})\ (\underbrace{p\,.\,\mathbf{do}\ q \leftarrow \mathbf{op}\ldots\ \mathbf{return}\ (p,q)}_{\mathsf{Bool}\to(\mathsf{Bool},\mathsf{Bool})\,!\,\langle \mathtt{once}\,;\mathtt{choose}\rangle})$$

Dealing with the presence of this scoped result type is the most non-trivial part for designing a calculus with composable scoped effects. There are two main complications. First, since the type does not occur in the computation's overall type, *polymorphic handlers* are required to handle scoped effects. Second, the scoped result type describes a dependency between the scoped computation and continuation: if one changes its type, the other must match this. This makes generic *forwarding* impossible: it alters the type of the scoped computation, but does not make up for it in the continuation.

The rest of this section covers these two complications. While understanding these issues is important for fully understanding the semantics presented in Section 4.5, it may be hard to fully comprehend the remainder section without having seen the type system presented in Section 4.6. We recommend the reader to skim over parts of the remainder of this section that are unclear, and revisit them after having seen the type system.

Polymorphic Handlers

Applying a handler to a computation involves recursively applying the handler to the computation's subcomputations as well. In the case of algebraic effects, these subcomputations always have the same type as the operation itself, as witnessed by the algebraicity property. This means that calculi that only support algebraic effects & handlers, such as Eff, allow typing handlers monomorphically, without limitations in expressivity.

However, typing scoped effect handlers monomorphically *does* limit their implementation freedom: it only allows scoped operations of which the scoped result type matches the operation's overall type. For example, consider the type $(\mathsf{Bool}, \mathsf{Bool})\,!\,\langle \mathtt{choose}\,;\mathtt{once}\rangle \Rightarrow \mathsf{List}\ (\mathsf{Bool}, \mathsf{Bool})\,!\,\langle\rangle$ we previously assigned to h_{once}. This monomorphic type requires the scoped result type to be $(\mathsf{Bool}, \mathsf{Bool})$ as well, as it is the only type of computation monomorphic h_{once} can handle. This is not the case for c_{once}: as established, its scoped result type is Bool. Therefore, scoped computations such as c_{once}, cannot be handled by monomorphic handlers. The solution is to let handlers abstract over the value type of computations, allowing for the handling of scoped operations with *any* scoped result type. This way, h_{once} can be typed as follows:

$h_{\mathsf{once}} : \forall\, \alpha\,.\, \alpha\,!\,\langle \mathtt{choose}\,;\mathtt{once}\rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle\rangle$

With this polymorphic typing in place, **with** h_{once} **handle** c_{once} can now be evaluated by *polymorphic recursion*. To support this, λ_{sc} features **let**-polymorphism, F_ω-style type operators, and requires all handlers for scoped effects to be polymorphic.

Forwarding Unknown Operations

In order to retain the modularity of composing different effects, as discussed in Section 4.2.1, we allow for partial handlers (i.e. handlers that handle only a subset of the effects they are applied to) to obtain dedicated handlers that interpret only their part of the syntax. This requires that all remaining operations be *forwarded* to other handlers. As established, this forwarding can, in the case for algebraic effects and in some specific situations, be done in a canonical way.

One might hope to forward scoped effects in a similar way, where the inner computations are handled, and the effect is left as-is for future handlers to handle. For example, consider the following computation which uses a scoped operation $\mathtt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}$ for catching exceptions.

$$c_{\mathsf{catch}} = \mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)\ (x\,.\,\mathbf{return}\ x)$$
$$\quad \mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle}\ c_{\mathsf{catch}}$$

Applying the handler h_{once} to it gives us

$$\begin{aligned} & \mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle}\ c_{\mathsf{catch}} \\ \rightsquigarrow_{\not\mathsf{x}}\ & \mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ (b\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ if}\ b \ldots) \\ & \qquad\qquad\qquad (x\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ return}\ x) \end{aligned}$$

Unfortunately, this does not work: again, the hurdle is in the scoped result type, in combination with the fact that applying a handler to a computation changes its type. In our example, h_{once} applies the type operator List when handling a computation, resulting in a handled scoped computation of type $\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle\mathtt{catch}\rangle$, and a handled continuation of type $\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle\mathtt{catch}\rangle$.

$$\mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ \overbrace{(b\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else} \ldots)}^{\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle\mathtt{catch}\rangle}\ \underbrace{(x\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ return}\ x)}_{\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle\mathtt{catch}\rangle}$$

This introduced a type mismatch, as the return type of the scoped computation has changed ($\mathsf{List\ Int}$), whereas the continuation still expects the original type

(Int). In other words: the scoped computation and continuation no longer agree on a scoped result type. Thus, scoped effects cannot be forwarded uniformly. Therefore, we require that every handler is equipped with an explicit forwarding clause for unknown scoped operations, specific to the handler's carrier (i.e. the type constructor is applied to a computation when handled). This forwarding clause is used to resolve the type mismatch between the result type of the handled scoped computation and the parameter type of continuation.

Considering the above example, the goal of applying a function of type $\mathsf{Int} \to \mathsf{List\ Int}!\langle E \rangle$ to a value of type $\mathsf{List\ Int}$ naturally reminds us of monadic binding. Indeed, for most carriers, this forwarding clause is very similar to that of the monadic bind $\gg\!=$. In fact, for now we will assume this to be the case for all effects, and introduce λ_{sc} based on these bind-like forwarding clauses. In Section 4.7 we will revisit this firstly by giving examples where bind-like clauses are not expressive enough, secondly by generalizing the forwarding clause, and lastly by showing how to desugar bind-like clauses to generalized forwarding clauses. Therefore, forwarding clauses are—for now—of shape $\mathbf{bind}\ x\ k$, where x represents the result of the scoped computation, and k represents the continuation, yielding a type similar to $\gg\!=$.

What should we write for a forwarding clause? The answer depends on what we expect when we want to connect the results of a handler to outside. For example, consider the handler h_{once}. The x here is the list of results we get from the nondeterministic program in the scope after handled by h_{once}. The k here is what remains to do for each result. The most intuitive way to connect x and k is to run k on every element in x and connect their results together. This is exactly what the concatMap function in Haskell does.

$$h_{\mathsf{once}} = \mathbf{handler}\ \{\ \ldots, \mathbf{bind}\ x\ k \mapsto \mathsf{concatMap}\ x\ k\}$$

We define concatMap in λ_{sc} as follows.

$$\begin{aligned}
&\mathsf{concatMap}\ : \forall\ \alpha\ \beta\ \mu\,.\,\mathsf{List}\ \beta \to^{\mu} (\beta \to^{\mu} \mathsf{List}\ \alpha) \to^{\mu} \mathsf{List}\ \alpha \\
&\mathsf{concatMap}\ [\,]\ \ \ f = \mathbf{return}\ [\,] \\
&\mathsf{concatMap}\ (b : bs)\ f = \mathbf{do}\ as \leftarrow f\ b\,;\, as' \leftarrow \mathsf{concatMap}\ bs\ f\,;\, as +\!\!+ as'
\end{aligned}$$

Note that there is no unique correct forwarding clause for every handler; it is up to the programmers to decide which kind of forwarding behaviours they want. For example, for h_{once} we can also use a reversed version of concatMap which does the same thing except for traversing the list from right to left. Alternatively, we can only pass the first element of x to k, or even invoke other effects like printing some information when forwarding. Comparing these different styles of forwarding, we prefer the original version using a left-to-right concatMap since we usually do not want other scoped effects to unexpectedly change the results of h_{once} by discarding some results or reversing the order of results. From our

$$
\begin{array}{rcl}
\text{values } v & ::= & () \mid (v_1, v_2) \mid x \mid \boldsymbol{\lambda} x \,.\, c \mid h \\
\text{handlers } h & ::= & \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r \\
 & & \quad , \mathit{oprs} \\
 & & \quad , \mathbf{bind}\ x\ k \mapsto c_f\,\} \\
\text{operation clauses } \mathit{oprs} & ::= & \cdot \\
 & \mid & \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs} \\
 & \mid & \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, \mathit{oprs} \\
\text{computations } c & ::= & \mathbf{return}\ v \\
 & \mid & \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y \,.\, c) \\
 & \mid & \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y \,.\, c_1)\ (z \,.\, c_2) \\
 & \mid & \mathbf{with}\ v\ \mathbf{handle}\ c \\
 & \mid & \mathbf{do}\ x \leftarrow c_1\,;\, c_2 \\
 & \mid & v_1\ v_2 \\
 & \mid & \mathbf{let}\ x = v\ \mathbf{in}\ c
\end{array}
$$

Figure 4.1: Terms of λ_{sc}.

experience of implementing a range of examples of scoped effects in Section 4.9, we found that a simple and intuitive forwarding clause usually yields the expect semantics.

In what follows, we put our calculus on formal footing, discussing its syntax, operational semantics and type-and-effect system.

4.4 Term Syntax

As stated, λ_{sc} is based on Eff [5, 6]. Before adding support for scoped effects, we have altered Eff from its presentation in [5, 6] in two ways. Firstly, we have made a number of cosmetic changes that arguably improve the readability of the calculus. Secondly, we adopt row-based typing in the style of Koka [42]. The row types will be introduced in Section 4.6.

Like Eff we implement fine-grained call-by-value semantics [44]. Therefore, terms are split into inert values and computations that can be reduced.

4.4.1 Computations

For computations, **return** can be used to return values. Handlers can be applied to values with the **with** and **handle** keywords. As seen before, computations may be sequenced by means of do-statements ($\mathbf{do}\ x \leftarrow c_1\,;\, c_2$). Applications reduce, so they are computations. As discussed in Section 4.3.3, to support polymorphic handlers we support let-polymorphism and thus let-bindings. Finally, a computation may be the invocation of an effect by means of an operation.

To be able to differentiate between algebraic and scoped effects, we add the effect keyword **sc** to model scoped effects. Consequently, **op** now ranges over algebraic effects only. Furthermore, we annotate labels with either op or sc to indicate if they are the label of an algebraic or scoped effect, respectively. We implicitly assume any label ℓ occurs either as an algebraic or scoped effect label. Like their algebraic counterparts, scoped effect operations feature a label ℓ^{sc}, argument v and continuation $(z\,.\,c_2)$. In addition, scoped effect operations feature a scoped computation $(y\,.\,c_1)$. This way, the scope of effect ℓ^{sc} is delimited: (scoped) computation $(y\,.\,c_1)$ is in scope, continuation $(z\,.\,c_2)$ is not.

4.4.2 Values

Values consist of the unit value $()$, value pairs (v_1, v_2), variables x, functions $\boldsymbol{\lambda} x\,.\,c$ and handlers h. Handlers h have three kinds of clauses: one return clause, zero or more operation clauses, and a forwarding clause.

The return clause $\mathbf{return}\ x \mapsto c$ denotes that the result x of a computation is processed by computation c.

Algebraic operation clauses $\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c$ specify that handling an effect with label ℓ^{op}, parameter x and continuation k is processed by computation c (e.g., h_{ND}, h_{get}, h_{inc}). For scoped effect clauses the extension is analogous to the operation case: we take the algebraic clause and add support for a scoped computation, which in the case for the clause has the form of parameter p.

Finally, as motivated in Section 4.3.3, we have forwarding clauses of shape $\mathbf{bind}\ x\ k \mapsto c$ that deal with forwarding unknown scoped operations with some label ℓ^{sc}. We will generalize these in Section 4.7.

4.5 Operational Semantics

Figure 4.2 displays the small-step operational semantics of λ_{sc}. Here, relation $c \rightsquigarrow c'$ denotes that computation c steps to computation c', with $\rightsquigarrow^*$ its reflexive, transitive closure. The highlighted rules deal with the extensions that support scoped effects. The following discussion of the semantics is exemplified by snippets of derivations of computations[1] used in Section 4.2. We refer to Appendix A for the full version of these derivations.

Rules E-AppAbs and E-Let deal with function application and let-binding, respectively, and are standard. The rest of the rules consist of two parts: sequencing and handling.

Sequencing For sequencing computations $\mathbf{do}\ x \leftarrow c_1\,;c_2$, we distinguish between the situation where c_1 can take a step (E-Do), and where c_1 is in normal form (**return**, **op**, or **sc**). First, if c_1 returns a value v, we substitute v for x in c_2 (E-DoRet). Second, if c_1 is an algebraic operation, we rewrite the computation using the algebraicity property (E-DoOp), bubbling up the algebraic operation to the front of the computation. Third, the new case, where c_1 is a scoped operation, is analogous: the generalization of the algebraicity property (Section 4.3.1) is used to rewrite the computation (E-DoSc).

Handling For handling computations with a handler of the form $\mathbf{with}\ h\ \mathbf{handle}\ c$, we distinguish six situations. First, if possible, c takes a step (E-Hand); in the other cases, c is in normal form. If c returns a value v, we use the handler's return clause $\mathbf{return}\ x \mapsto c_r$, switching evaluation to c_r with x replaced by v (E-HandRet).

If computation c is an algebraic operation $\mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1)$, its label is looked up in the handler h. If the handler contains an algebraic clause with this label, evaluation switches to the clause's computation c (E-HandOp), with v substituted for parameter x and continuation k replaced by a function that, given the original argument y, contains the already-handled[2] continuation. For example, $\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}}$ (p. 64) reduces as follows.

$$
\begin{aligned}
&\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}} \\
\rightsquigarrow\ &\mathbf{do}\ xs \leftarrow (\boldsymbol{\lambda} b\,.\,\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 2)\ \mathsf{true}
\end{aligned}
$$

[1] Following convention, these examples may contain elements not present in our calculus, such as integers and if-then-else statements. These may be viewed as syntactic sugar for their Church encodings.

[2] Like eff, λ_{sc} implements *deep* handlers, as opposed to *shallow* handlers.[31]

$\boxed{c \rightsquigarrow c'}$ Computation reduction

$$\frac{}{(\boldsymbol{\lambda} x\,.\,c)\ v \rightsquigarrow c\,[\,v\,/\,x\,]}\ \textsc{E-AppAbs} \qquad \frac{}{\mathbf{let}\ x = v\ \mathbf{in}\ c \rightsquigarrow c\,[\,v\,/\,x\,]}\ \textsc{E-Let}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\mathbf{do}\ x \leftarrow c_1\,;\,c_2 \rightsquigarrow \mathbf{do}\ x \leftarrow c_1'\,;\,c_2}\ \textsc{E-Do}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{return}\ v\,;\,c_2 \rightsquigarrow c_2\,[\,v\,/\,x\,]}\ \textsc{E-DoRet}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1)\,;\,c_2 \rightsquigarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,\mathbf{do}\ x \leftarrow c_1\,;\,c_2)}\ \textsc{E-DoOp}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\,;\,c_3 \rightsquigarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,\mathbf{do}\ x \leftarrow c_2\,;\,c_3)}\ \textsc{E-DoSc}$$

$$\frac{c \rightsquigarrow c'}{\mathbf{with}\ h\ \mathbf{handle}\ c \rightsquigarrow \mathbf{with}\ h\ \mathbf{handle}\ c'}\ \textsc{E-Hand}$$

$$\frac{(\mathbf{return}\ x \mapsto c_r)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{return}\ v \rightsquigarrow c_r\,[\,v\,/\,x\,]}\ \textsc{E-HandRet}$$

$$\frac{(\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) \rightsquigarrow c\,[\,v\,/\,x, (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\,/\,k\,]}\ \textsc{E-HandOp}$$

$$\frac{(\mathbf{op}\ \ell^{\mathsf{op}}\ \_\ \_)\ \notin\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) \rightsquigarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)}\ \textsc{E-FwdOp}$$

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c)\ \in\ h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \\ \quad c\,[\,v\,/\,x, (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\,/\,p, (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\,/\,k\,]\end{array}}\ \textsc{E-HandSc}$$

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_)\ \notin\ h \qquad (\mathbf{bind}\ x\ k \mapsto c)\ \in\ h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \\ \quad \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1) \\ \qquad (z\,.\,c\,[\,z\,/\,x, (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\,/\,k\,])\end{array}}\ \textsc{E-Bind}$$

Figure 4.2: Operational semantics of λ_{sc}.

$$
\begin{array}{ll}
 & \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{false} \\
 & \quad xs +\!\!+ ys \\
\rightsquigarrow^* & \textbf{return}\ [1,2]
\end{array}
$$

In case h does not contain a clause for label ℓ^{op}, the effect is forwarded (E-FwdOp). Algebraic effects can be forwarded generically: we re-invoke the operation and recursively apply the handler to continuation c_1. For example, during the application of run_{inc} in **with** h_{ND} **handle** $run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}}$ (p. 68), `choose` is forwarded:

$$
\begin{array}{ll}
 & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}} \\
\rightsquigarrow & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle} \\
 & \quad \textbf{if}\ b\ \textbf{then op}\ \texttt{inc}\ ()\ (x\,.\,x+5)\ \textbf{else op}\ \texttt{inc}\ ()\ (y\,.\,y+2))\,;\,p'\ 0 \\
\rightsquigarrow^* & \textbf{return}\ [(6,1),(3,1)]
\end{array}
$$

If computation c is a scoped operation $\textbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, we again distinguish two situations: the case where h contains a clause for ℓ^{sc}, and where it does not. If h contains a clause for label ℓ^{sc}, evaluation switches to the clause's computation c (E-HandSc), with v substituted for parameter x. Both the scoped computation and the continuation are replaced by a function that contains the already-handled computations c_1 and c_2. For example, this happens for the scoped operation `once` in **with** h_{once} **handle** c_{once} (p. 70).

$$
\begin{array}{ll}
 & \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ c_{\mathsf{once}} \\
\rightsquigarrow & \textbf{do}\ ts \leftarrow (\boldsymbol{\lambda}\_\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \texttt{choose}\ (b\,.\,\textbf{return}\ b))\ () \\
 & \textbf{do}\ t\ \leftarrow \mathsf{head}\ ts\ (\boldsymbol{\lambda} p\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose} \\
 & \qquad\qquad (b\,.\,\textbf{return}\ b)\,;\,\textbf{return}\ (p,q)))\ t \\
\rightsquigarrow^* & \textbf{return}\ [(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false})]
\end{array}
$$

When h does not contain a clause for label ℓ^{sc}, we must forward the effect. As we showed in Section 4.3.3, forwarding scoped effects cannot happen canonically, but (for now) rather proceeds via the handler's forwarding clause $\textbf{bind}\ x\ k \mapsto c$. Similar to the forwarding of algebraic effects, we recursively handle the subcomputations. However, in the case of **bind**, we set the continuation of the to-be-forwarded effect to the forwarding clause's body c, substituted with the relevant arguments.

For a computation, consider again the example described in Section 4.3.3. Even though we have not given any semantics to `catch` yet (we will do so in Section 4.9.1), we know h_{once} does not contain a clause for `catch`. As described, `catch` must be forwarded, addressing the type mismatch between the handled scoped computation and handled continuation with concatMap.

$$
\begin{array}{rcl}
\text{value types } A,\, B,\, M & ::= & () \mid (A, B) \mid A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D} \mid \\
& & \alpha \mid \lambda\, \alpha\,.\, A \mid M\ A \\
\text{type schemes } \sigma & ::= & A \mid \forall\, \mu\,.\, \sigma \mid \forall\, \alpha\,.\, \sigma \\
\text{computation types } \underline{C},\, \underline{D} & ::= & A\,!\,\langle E \rangle \\
\text{effect rows } E,\, F & ::= & \cdot \mid \mu \mid \ell\,;\, E \\
\text{signature contexts } \Sigma & ::= & \cdot \mid \Sigma, \ell^{\mathsf{op}} : A \twoheadrightarrow B \mid \Sigma, \ell^{\mathsf{sc}} : A \twoheadrightarrow B \\
\text{type contexts } \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \mid \Gamma, \mu \mid \Gamma, \alpha
\end{array}
$$

Figure 4.3: Types of λ_{sc}.

$$
\begin{array}{l}
\quad \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ c_{\mathsf{catch}} \\
\rightsquigarrow\ \textbf{sc}\ \texttt{catch "err"}\ (b\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2) \\
\qquad\qquad\quad (x\,.\,\mathsf{concatMap}\ x\ (\boldsymbol{\lambda} z\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ z))
\end{array}
$$

4.6 Type-and-Effect System

This section presents the type-and-effect system of λ_{sc}. As before, we distinguish between values, computations and handlers.

4.6.1 Grammar

Figure 4.3 displays the grammar of the types of λ_{sc}. Like terms, types are split: values have value types A, B, computations have computation types $\underline{C}, \underline{D}$. Value types consist of the unit type $()$, pair types (A, B), function types $A \to \underline{C}$, handler types $\underline{C} \Rightarrow \underline{D}$, type variables α and abstraction over them $\boldsymbol{\lambda}\alpha\,.\,A$, and type application $M\ A$. Following convention and to allow for meaningful examples, we may add base value types to the calculus, such as String, Int and Bool. Functions take a value of type A as argument and return a computation of type $\underline{C}$; handlers take a computation of type $\underline{C}$ as argument and return a computation of type $\underline{D}$.

A computation type $A\,!\,\langle E \rangle$ consists of a value type A, representing the type of the value the computation evaluates to, and an effect type E, representing the effects that *may* be called during this evaluation. Different from Eff, we implement effect types as effect rows using row polymorphism [40] in the style of Koka [42]. Therefore, rows E are represented as collections of the previously discussed atomic labels ℓ^{op} and ℓ^{sc}, optionally terminated by a row variable μ.

$\boxed{\Gamma \vdash v : \sigma}$ Value Typing

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash () : ()}\ \text{T-Unit} \qquad \frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : (A, B)}\ \text{T-Pair}$$

$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \boldsymbol{\lambda} x \,.\, c : A \to \underline{C}}\ \text{T-Abs} \qquad \frac{\Gamma \vdash v : A \qquad A \equiv B}{\Gamma \vdash v : B}\ \text{T-EqV}$$

$$\frac{\Gamma \vdash v : \forall\, \alpha \,.\, \sigma \qquad \Gamma \vdash A}{\Gamma \vdash v : [A \,/\, \alpha]\, \sigma}\ \text{T-Inst} \qquad \frac{\Gamma, \alpha \vdash v : \sigma \qquad \alpha \notin \Gamma}{\Gamma \vdash v : \forall\, \alpha \,.\, \sigma}\ \text{T-Gen}$$

$$\frac{\Gamma \vdash v : \forall\, \mu \,.\, \sigma \qquad \Gamma \vdash E}{\Gamma \vdash v : [E \,/\, \mu]\, \sigma}\ \text{T-InstEff} \qquad \frac{\Gamma, \mu \vdash v : \sigma \qquad \mu \notin \Gamma}{\Gamma \vdash v : \forall\, \mu \,.\, \sigma}\ \text{T-GenEff}$$

Figure 4.4: Value typing.

Finally, we can abstract over both type and row variables, giving rise to type schemes σ.

4.6.2 Value typing

Figure 4.4 displays the typing rules for values. Rules T-Var, T-Unit, T-Pair and T-Abs type variables, units, pairs and term abstractions, respectively, and are standard.

Rule T-EqV expresses that typing holds up to equivalence of types. The full type equivalence relation ($A \equiv B$), which also uses the equivalence of rows ($E \equiv_{\langle\rangle} F$), is included in Appendix B. However, these relations can be described as the congruence closure of the following two rules.

$$\frac{}{(\lambda\, \alpha \,.\, A)\ B \ \equiv\ A\,[B \,/\, \alpha]}\ \text{Q-AppAbs} \qquad \frac{\ell_1 \neq \ell_2}{\ell_1 \,;\, \ell_2 \,;\, E \ \equiv_{\langle\rangle}\ \ell_2 \,;\, \ell_1 \,;\, E}\ \text{R-Swap}$$

Rule Q-AppAbs captures type application, following F_ω, and R-Swap captures the insignificance of the order in effect rows, following Koka's row typing approach.

The final four value typing rules deal with generalization and instantiation of type variables and row variables. Rule T-Inst instantiates the type variables α in a type scheme with a value type A. Rule T-Gen is its dual, abstracting over a type variable. The rules for row variables are similar: T-InstEff instantiates

$\boxed{\Gamma \vdash c : \underline{C}}$ Computation Typing

$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \underline{C}}\ \text{T-App}$$

$$\frac{\Gamma \vdash c_1 : A!\langle E\rangle \qquad \Gamma, x : A \vdash c_2 : B!\langle E\rangle}{\Gamma \vdash \mathbf{do}\ x \leftarrow c_1\,;\, c_2 : B!\langle E\rangle}\ \text{T-Do} \qquad \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \equiv \underline{D}}{\Gamma \vdash c : \underline{D}}\ \text{T-EqC}$$

$$\frac{\Gamma \vdash v : \sigma \qquad \Gamma, x : \sigma \vdash c : \underline{C}}{\Gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}}\ \text{T-Let} \qquad \frac{\Gamma \vdash v : A \qquad \Gamma \vdash E}{\Gamma \vdash \mathbf{return}\ v : A!\langle E\rangle}\ \text{T-Ret}$$

$$\frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathbf{with}\ v\ \mathbf{handle}\ c : \underline{D}}\ \text{T-Hand}$$

$$\frac{(\ell^{\mathsf{op}} : A_1 \twoheadrightarrow A_2) \in \Sigma \qquad \Gamma \vdash v : A_1 \qquad \Gamma, y : A_2 \vdash c : A_3!\langle E\rangle \qquad \ell^{\mathsf{op}} \in E}{\Gamma \vdash \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c) : A_3!\langle E\rangle}\ \text{T-Op}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{sc}} : A_1 \twoheadrightarrow A_2) \in \Sigma \\ \Gamma \vdash v : A_1 \qquad \Gamma, y : A_2 \vdash c_1 : A_3!\langle E\rangle \qquad \Gamma, z : A_3 \vdash c_2 : A_4!\langle E\rangle \qquad \ell^{\mathsf{sc}} \in E\end{array}}{\Gamma \vdash \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A_4!\langle E\rangle}\ \text{T-Sc}$$

Figure 4.5: Computation typing.

row variable with an effect row E; T-GenEff abstracts over a row variable. The definition of well-scopedness for types $\Gamma \vdash \sigma$ and effect rows $\Gamma \vdash E$ is straightforward (Appendix C).

4.6.3 Computation typing

Figure 4.5 shows the rules for computation typing. Rules T-App and T-Do capture application and sequencing, and are standard. Like value typing, typing of computations holds up to equivalence of types (T-EqC). Rule T-Let is part of our extension of Eff, as scoped effects require introducing let-polymorphism.

Rule T-Ret assigns a computation type to a return statement. This type consists of the value v in the return, together with a effect row E. Notice that, as in Koka, this row can be freely chosen.

Rule T-Hand types handler application. The typing rules for handlers and their clauses are discussed in Section 4.6.4. A handler of type $C \Rightarrow D$ denotes a

handler that transforms computations of type C to a computation of type D.

Rule T-Op types algebraic effects. Looking up label ℓ^{op} in Σ yields a signature $A_1 \twoheadrightarrow A_2$, where A_1 is the type of the operation's parameter v, and A_2 is the type of argument y of the continuation. The resulting effect row includes ℓ^{op}. Indeed, $\ell \in E$ means that there is some E' such that $E \equiv_{\langle\rangle} \ell\,; E'$. Finally, the operation's type equals that of continuation c.

Similarly, rule T-Sc types scoped effects. Again, looking up label ℓ^{sc} in Σ yields signature $A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}$ where A_{sc} corresponds to the type of the operation's parameter v. However, where B_{op} in the algebraic case refers to the *continuation's* argument, B_{sc} now describes the *scoped computation's* argument. This leaves the the scoped result type undescribed by the signature, but as discussed in Section 4.3.3, this freedom is exactly what we want. As for the effect rows, T-Sc requires the rows of the scoped computation to match.

4.6.4 Handler typing

The typing rules for handlers and handler clauses are shown in Figure 4.6. They consist of four judgments. Judgment $\Gamma \vdash \mathbf{return}\ x \mapsto c_r : M\ A!\langle E\rangle$ types return clauses, $\Gamma \vdash \mathit{oprs} : M\ A!\langle E\rangle$ types operation clauses, and $\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c : M\ A!\langle E\rangle$ types forwarding clauses. Finally, $\Gamma \vdash h : \forall\,\alpha\,.\,\alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle$ types handlers, using the first three judgments.

Return Clauses Rule T-Return types return clauses of the form $\mathbf{return}\ x \mapsto c_r$. It binds variable x to type A, adds it to the context, and returns the type $M\ A!\langle E\rangle$ of c_r as the type of the return clause.

Operation Clauses The judgment $\Gamma \vdash \mathit{oprs} : M\ A!\langle E\rangle$ denotes that all operations in the sequence of operations *oprs* have type $M\ A!\langle E\rangle$. The base case T-Empty types the empty sequence. The other two cases require the head of the sequence (either **op** or **sc**) to have the same type as the tail.

Rule T-OprOp types algebraic operation clauses $\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c$. Looking up label ℓ^{op} in Σ yields signature $A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}$, where A_{op} describes the type of parameter x, and B_{op} the type of the argument of continuation k. In order for an operation **op** to have type $M\ A!\langle E\rangle$, c should have the same type.

Once again, the case for typing a scoped clause $\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c$ (T-OprSc) is similar to its algebraic equivalent, extended to include the scoped computation. Notice the type of p and k when typing c. First, as λ_{sc} allows freedom in the scoped result type, the type variable β is used for this type. Second, as shown

$\boxed{\Gamma \vdash \mathbf{return}\ x \mapsto c_r : \underline{C}}$ $\boxed{\Gamma \vdash \mathit{oprs} : \underline{C}}$ $\boxed{\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c_f : \underline{C}}$
Handler clause typing

$$\frac{\Gamma, x : A \mapsto c_r : \underline{C}}{\Gamma \vdash \mathbf{return}\ x \mapsto c_r : \underline{C}}\ \text{T-Return} \qquad \frac{}{\Gamma \vdash \cdot : \underline{C}}\ \text{T-Empty}$$

$$\frac{\Gamma \vdash \mathit{oprs} : \underline{C} \qquad (\ell^{\mathsf{op}} : A_1 \twoheadrightarrow A_2) \in \Sigma \qquad \Gamma, x : A_1, k : A_2 \to \underline{C} \vdash c : \underline{C}}{\Gamma \vdash \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs} : \underline{C}}\ \text{T-OprOp}$$

$$\frac{\Gamma \vdash \mathit{oprs} : M\ A_1\,!\langle E\rangle \qquad (\ell^{\mathsf{sc}} : A_2 \twoheadrightarrow A_3) \in \Sigma \qquad \Gamma, \beta, x : A_2, p : A_3 \to M\ \beta\,!\langle E\rangle, k : \beta \to M\ A_1\,!\langle E\rangle \vdash c : M\ A_1\,!\langle E\rangle}{\Gamma \vdash \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, \mathit{oprs} : M\ A_1\,!\langle E\rangle}\ \text{T-OprSc}$$

$$\frac{\Gamma, \alpha, x : M\ \alpha, k : \alpha \to M\ A\,!\langle E\rangle \vdash c_f : M\ A\,!\langle E\rangle}{\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c_f : M\ A\,!\langle E\rangle}\ \text{T-Bind}$$

$\boxed{\Gamma \vdash h : \forall\ \alpha\,.\,\alpha\,!\langle E\rangle \Rightarrow M\ \alpha\,!\langle F\rangle}$ Handler typing

$$\text{T-Handler}_{\text{Bind}}\ \frac{\langle F\rangle \equiv_{\langle\rangle} \langle \mathit{labels}(\mathit{oprs})\,; E\rangle \qquad \Gamma, \alpha \vdash \mathbf{return}\ x \mapsto c_r : M\ \alpha\,!\langle E\rangle \qquad \Gamma, \alpha \vdash \mathit{oprs} : M\ \alpha\,!\langle E\rangle \qquad \Gamma, \alpha \vdash \mathbf{bind}\ x\ k \mapsto c_f : M\ \alpha\,!\langle E\rangle}{\Gamma \vdash \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{bind}\ x\ k \mapsto c_f\} : \forall\ \alpha\,.\,\alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle}$$

Figure 4.6: Handler typing.

in the operational semantics (rule E-HandSc), for a clause $\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c$, computation c uses the *already-handled* subcomputations p and k. Therefore, type operator M occurs in the scoped result type as well as in the continuation's result type:

$$p : B_{\mathsf{sc}} \to M\ \beta\,!\langle E\rangle \qquad\qquad k : \beta \to M\ A\,!\langle E\rangle$$

This means that, even though our focus on mitigating the type mismatch between scoped computation and continuation so far has been on forwarding *unknown* scoped effects, the same applies when handling *known* scoped effects, where computation c accounts for this discrepancy.

Forwarding Clause In accordance with Section 4.3.3, when an application of a handler to a scoped effect the handler does not contain a clause for is encountered, the effect must be forwarded. Like the algebraic case, the effect is left in-place with handled sub-computations. However, as described in Section 4.3.3, simply handling the sub-computations would result in a type mismatch. It is clear from the typing rule that $\mathbf{bind}\ x\ k \mapsto c_f$ basically deals with this type mismatch between the type $M\ \alpha$ of x and the parameter type α of k.

Handler Rule T-Handler types handlers with a polymorphic type of the form $\forall\ \alpha\,.\,\alpha\,!\,\langle F\rangle \Rightarrow M\ \alpha\,!\,\langle E\rangle$. A handler consists of a **return**-clause (T-Return), zero or more operation clauses (T-Empty, T-OprOp and T-OprSc), and a forwarding clause (T-Fwd). All clauses should agree on their result type $M\ \alpha\,!\,\langle E\rangle$. Notice that E denotes a collection with at least the labels of the present algebraic and scoped operation clauses in the handler (computed by the *labels*-function).

4.7 Generalized Forwarding

In Section 4.3.3 we showed how **bind** can be used to forward certain scoped effects. Here we also noted that not all effects can be forwarded by **bind**, and that some require a more general forwarding construct. In this section we revisit this issue. First, we show how in some situations, we need a more expressive forwarding clause. We will then present a generalized forwarding clause, and show how the **bind**-based forwarding clauses can be straightforwardly desugared into generalized forwarding clauses. Finally, we will amend the calculus with rules for its grammar, semantics and typing.

4.7.1 *Inc*, revisited

We introduced h_{inc} in an algebraic setting, and have not returned to it. We have not yet given a forwarding clause for h_{inc} (with carrier $(\mathsf{Int} \to (\alpha, \mathsf{Int})\,!\,\langle E\rangle)\,!\,\langle E\rangle$ for some effect type E). We could implement the forwarding clause of it using **bind** as follows.

$$
\begin{aligned}
h_{\mathsf{inc}\cancel{\mathsf{X}}} = \mathbf{handler}\ & \\
& \{\mathbf{return}\ x \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{return}\ (x, s)) \\
& ,\ \mathbf{op}\ \mathtt{inc}\ \_\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ s' \leftarrow s + 1\,;\mathbf{do}\ k' \leftarrow k\ s'\,;k'\ s') \\
& ,\ \mathbf{bind}\ x\ k \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ (y, s') \leftarrow x\ s\,;k' \leftarrow k\ y\,;k'\ s')\}
\end{aligned}
$$

However, this implementation of forwarding would actually result in unexpected behaviours. Consider the derivation displayed in Figure 4.7. The chosen example has an occurrence of `choose` inside an occurrence of `inc`, both of them inside an occurrence of `once`. Since `choose` is in the scope of `once`, we would expect that the `choose` is pruned by the `once` to only keep the first branch. However, the derivation shows that by using the **bind**-based forwarding clause of $h_{\mathsf{inc}\checkmark}$ the `choose` escapes the scope of `once`. As can be seen, when `head` (i.e. the embodiment of the `once` effect) is evaluated, `choose` has not even been handled yet; it is captured by a function in the result list.

For the brevity of presentation, we omitted the unimportant part of terms in the derivation, and we inline the guard that checks for the empty list in the if statement. We underlined parts that will reduce in the next step(s).

The essence of this problem comes from the fact that the carrier of h_{inc} contain computations. As a result, other operations in scope might be captured in these computations and then be carried out of the scope. The **bind**-based forwarding of h_{inc} is not expressive enough to avoid this kind of unexpected behaviour, because it has no control over what is returned by the scoped computation of scoped operations. For the example above, to correctly put `choose` in the scope, we need to evaluate the result function of the scoped computation of `once` that is handled by h_{inc}, instead of directly returning this function which unexpectedly captures `choose`. We will show how to solve this problem using a generalized version of forwarding clauses in the remaining of this section.

In general, this problem shows up for any handler whose carrier suspends computations, such as the carrier of h_{inc} which is a function. The effects captured in the suspend computation can escape their scope without being handled. The generalized forwarding clause fixes this problem by providing us the flexibility to early execute the suspend computation before leaving the scope.

4.7.2 Adding generalized forwarding clauses

Whereas in the case of **bind**-like forwarding clauses the remittal of the unknown, to-be-forwarded scoped effect is dealt with by the semantics (i.e., the reinvocation of scoped effect is embedded in the rule E-Bind), in the case of generalized forwarding clauses, denoted by **fwd**, this responsibility is deferred to the forwarding clause. This allows generalized forwarding clauses to execute logic *before* remitting the forwarded effect, and to alter not only the continuation but also the scoped computation of the forwarded effect. An intuitive implementation

$$
\begin{array}{ll}
 & \textbf{with } h_{\mathsf{once}} \textbf{ handle} \\
 & \quad \textbf{do } f \leftarrow \underline{\textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle}} \\
 & \qquad \textbf{sc } \underline{\mathtt{once}}\ ()\ (\_ .\ \textbf{op } \mathtt{inc}\ ()\ (\_ .\ \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b))) \\
 & \qquad\qquad (z .\ \textbf{return } z)\,; \\
 & \quad f\ 0 \\
\rightsquigarrow^* & \textbf{with } h_{\mathsf{once}} \textbf{ handle} \\
 & \quad \underline{\textbf{do } f} \leftarrow \\
 & \qquad \textbf{sc } \mathtt{once}\ ()\ (\_ .\ \textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle} \\
 & \qquad\qquad\qquad (\textbf{op } \mathtt{inc}\ ()\ (\_ .\ \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b)))) \\
 & \qquad\qquad (z .\ \ldots) \\
 & \quad \underline{f\ 0} \\
\rightsquigarrow & \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}} \\
 & \quad \textbf{sc } \underline{\mathtt{once}}\ ()\ (\_ .\ \textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle} \\
 & \qquad\qquad\qquad (\textbf{op } \mathtt{inc}\ ()\ (\_ .\ \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b))))) \\
 & \qquad\qquad (z .\ \ldots) \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle} \\
 & \qquad\qquad \underline{\textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle}}\ (\textbf{op } \underline{\mathtt{inc}}\ ()\ (\_ .\ \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b))) \\
 & \quad \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}} \\
 & \qquad\qquad \underline{\textbf{return}}\ (\boldsymbol{\lambda} s .\ \textbf{do } s' \leftarrow s + 1\,; \\
 & \qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle} \\
 & \qquad\qquad\qquad\qquad\quad \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b)\,; \\
 & \qquad\qquad\quad k'\ s') \\
 & \quad \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } \underline{ts} \leftarrow \underline{\textbf{return}}\ [\boldsymbol{\lambda} s .\ \textbf{do } s' \leftarrow s + 1\,; \\
 & \qquad\qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle} \\
 & \qquad\qquad\qquad\qquad\quad\quad \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b)\,; \\
 & \qquad\qquad\qquad\quad k'\ s']\,; \\
 & \quad \underline{\textbf{if }}\ (ts = [\,])\ \underline{\textbf{ then }}\ \textbf{return } [\,]\ \underline{\textbf{ else }}\ \textbf{do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } t \leftarrow \mathsf{head}\ [\boldsymbol{\lambda} s .\ \ \textbf{do } s' \leftarrow s + 1\,; \\
 & \qquad\qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\boldsymbol{\lightning}} \textbf{ handle} \\
 & \qquad\qquad\qquad\qquad\quad\quad \textbf{op } \mathtt{choose}\ ()\ (b .\ \textbf{return } b)\,; \\
 & \qquad\qquad\qquad\quad k'\ s']\,; \\
 & \quad \ldots \\
\rightsquigarrow^* & [(\mathsf{true}, 1), (\mathsf{false}, 1)]
\end{array}
$$

Figure 4.7: `once` escaping its scope when forwarded by **bind**

of **fwd** may look like this:

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_) \notin h \qquad (\mathbf{fwd}'\ \ell^{\mathsf{sc}\prime}\ x\ p\ k \mapsto c_f) \in h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \\ \quad c_f\ [\,\ell^{\mathsf{sc}}\ /\ \ell^{\mathsf{sc}\prime}, v\ /\ x, (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ p, \\ \qquad (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k]\end{array}}\ \text{E-FwdSc'}$$

The forwarding clause has form $\mathbf{fwd}'\ \ell^{\mathsf{sc}\prime}\ x\ p\ k \mapsto c_f$, where x, p, k represents for the parameter, (deeply handled) scoped computation, and (deeply handled) continuation, respectively. While this would give us our desired expressivity, it requires us to extend the calculus with the ability to abstract and substitute operation labels.

To avoid unnecessary complication, we opt for another design choice. Because of parametricity, there is only one thing a forwarding clause can do with an unknown label: invoke it with some (possibly altered) scoped computation and continuation. Below we utilize this property by, instead of supplying the forwarding clause with the label, we supply it with what is essentially a partial application of the **sc** operator to the unknown label. This partial application can be used by the forwarding clause in exactly the same way as the original label was. Therefore, this simplifies the type system, but does not affect the expressivity of forwarding clauses.

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_) \notin h \qquad (\mathbf{fwd}\ f\ p\ k \mapsto c_f) \in h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \\ \quad c_f\ [\,\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ p, (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k, \\ \qquad (\boldsymbol{\lambda}(p', k')\,.\,\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z))\ /\ f]\end{array}}\ \text{E-FwdSc}$$

4.7.3 Syntax & typing

We replace the **bind** syntax in handlers with **fwd**, and implement **bind** as syntactic sugar for **fwd**.

$$\begin{array}{rcl}\text{handlers } h & ::= & \mathbf{handler}\ \{\ldots \\ & & \qquad ,\mathbf{fwd}\ f\ p\ k \mapsto c_f\} \\ \mathbf{bind}\ x\ k \mapsto c & \equiv & \mathbf{fwd}\ f\ p\ k \mapsto f\ (p, (\boldsymbol{\lambda} x\,.\,c))\end{array}$$

Furthermore, we add a typing rule for typing **fwd** clauses, and a new rule for typing handlers containing **fwd** clauses.

$$\frac{\begin{array}{c} A_p = \alpha \to M\ \beta\,!\,\langle E\rangle \\ A'_p = \alpha \to \gamma\,!\,\langle E\rangle \qquad A_k = \beta \to M\ A\,!\,\langle E\rangle \qquad A'_k = \gamma \to \delta\,!\,\langle E\rangle \\ \Gamma, \alpha, \beta, p : A_p, k : A_k, f : \forall\ \gamma\ \delta\,.\,(A'_p, A'_k) \to \delta\,!\,\langle E\rangle \vdash c_f : M\ A\,!\,\langle E\rangle \end{array}}{\Gamma \vdash \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A\,!\,\langle E\rangle}\ \text{T-FWD}$$

$$\text{T-HANDLER}\ \frac{\begin{array}{c} \langle F\rangle \equiv_{\langle\rangle} \langle \mathit{labels}\,(\mathit{oprs})\,; E\rangle \qquad \Gamma, \alpha \vdash \mathbf{return}\ x \mapsto c_r : M\ \alpha\,!\,\langle E\rangle \\ \Gamma, \alpha \vdash \mathit{oprs} : M\ \alpha\,!\,\langle E\rangle \qquad \Gamma, \alpha \vdash \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha\,!\,\langle E\rangle \end{array}}{\Gamma \vdash \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{fwd}\ f\ p\ k \mapsto c_f\} : \forall\ \alpha\,.\,\alpha\,!\,\langle F\rangle \Rightarrow M\ \alpha\,!\,\langle E\rangle}$$

Rule E-BIND and T-HANDLER$_{\text{BIND}}$ can be derived (and are therefore superseded by) E-FWD and T-HANDLER.

4.7.4 h_{inc}, revisited again

Using generalized forwarding clauses, we can implement the forwarding of h_{inc} correctly as follows.

$$h_{\mathsf{inc}} = \mathbf{handler}\ \{\ldots, \mathbf{fwd}\ f\ p\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,\mathbf{do}\ p' \leftarrow p\ y\,; p'\ s, \\ \boldsymbol{\lambda}(z, s')\,.\,\mathbf{do}\ k' \leftarrow k\ z\,; k'\ s'))\}$$

Instead of directly returning the function p' like the **bind**-based forwarding clause, the **fwd**-based forwarding clause applies the initial state s to p' to avoid the escaping of the `choose` operation captured in p'. Then, to connect the results inside the scope with outside, we apply the continuation k to the result z and the updated state s'. The derivation is shown in Figure 4.8. We only get the first result of the `choose` operation.

4.8 Metatheory

4.8.1 Syntax-directed version of λ_{sc}

Appendix D contains a syntax-directed version of λ_{sc}. We prove the type safety of λ_{sc} by connecting the typing judgement of the declarative version of λ_{sc} to those of the syntax-directed version of λ_{sc} in Appendix E. The type inference algorithm implemented in our interpreter is also built on this syntax-directed version. The syntax-directed version was obtained by the following three transformations.

$$
\begin{array}{ll}
 & \textbf{with } h_{\mathsf{once}} \textbf{ handle} \\
 & \quad \textbf{do } f \leftarrow \underline{\textbf{with } h_{\mathsf{inc}} \textbf{ handle}} \\
 & \quad\quad \textbf{sc } \underline{\texttt{once}}\ () \quad (\_ . \textbf{op } \texttt{inc}\ ()\ (\_ . \textbf{op } \texttt{choose}\ ()\ (b . \textbf{return } b))) \\
 & \quad\quad\quad\quad\quad\quad (z . \textbf{return } z)\,; \\
 & \quad f\ 0 \\
\rightsquigarrow^* & \textbf{with } h_{\mathsf{once}} \textbf{ handle} \\
 & \quad \textbf{do } \underline{f} \leftarrow \underline{\textbf{return}}\ (\boldsymbol{\lambda} s . \textbf{sc } \texttt{once}\ () \\
 & \quad\quad (\_ .\ \textbf{do } p' \leftarrow (\textbf{with } h_{\mathsf{inc}} \textbf{ handle} \\
 & \quad\quad\quad\quad\quad\quad (\textbf{op } \texttt{inc}\ ()\ (\_ . \textbf{op } \texttt{choose}\ ()\ (b . \textbf{return } b)))\,; \\
 & \quad\quad\quad p'\ s)\ (z . \ldots)) \\
 & \quad \underline{f\ 0} \\
\rightsquigarrow^* & \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}}\ \textbf{sc } \underline{\texttt{once}}\ () \\
 & \quad\quad (\_ .\ \textbf{do } p' \leftarrow (\textbf{with } h_{\mathsf{inc}} \textbf{ handle} \\
 & \quad\quad\quad\quad\quad\quad (\textbf{op } \texttt{inc}\ ()\ (\_ . \textbf{op } \texttt{choose}\ ()\ (b . \textbf{return } b)))\,; \\
 & \quad\quad\quad p'\ 0)\ (z . \ldots) \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle } (\\
 & \quad \textbf{do } p' \leftarrow (\underline{\textbf{with } h_{\mathsf{inc}} \textbf{ handle}} \\
 & \quad\quad\quad (\textbf{op } \underline{\texttt{inc}}\ ()\ (\_ . \textbf{op } \texttt{choose}\ ()\ (b . \textbf{return } b)))\,; \\
 & \quad p'\ 0) \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle } (\\
 & \quad \textbf{do } \underline{p'} \leftarrow \underline{\textbf{return}}\ (\boldsymbol{\lambda} s . \textbf{do } s' \leftarrow s + 1\,; \\
 & \quad\quad\quad\quad\quad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}} \textbf{ handle} \\
 & \quad\quad\quad\quad\quad\quad\quad\quad \textbf{op } \texttt{choose}\ ()\ (b . \textbf{return } b)\,; \\
 & \quad\quad\quad\quad\quad\quad k'\ s') \\
 & \quad \underline{p'\ 0}) \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle } (\\
 & \quad \textbf{do } \underline{s'} \leftarrow \underline{0 + 1}\,; \\
 & \quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}} \textbf{ handle op } \texttt{choose}\ ()\ (b . \textbf{return } b)\,; \\
 & \quad k'\ \underline{s'}) \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle } (\\
 & \quad\quad \textbf{op } \texttt{choose}\ ()\ (b . \textbf{do } \underline{k'} \leftarrow \underline{\textbf{return}}\ (\boldsymbol{\lambda} s . (b, s))\,; \underline{k'}\ 1))\,; \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}}\ (\\
 & \quad\quad \textbf{op } \underline{\texttt{choose}}\ ()\ (b . \textbf{return } (b, 1)))\,; \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & \textbf{do } ts \leftarrow \textbf{return } [(\mathsf{true}, 1), (\mathsf{false}, 1)]\,; \\
 & \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots \\
\rightsquigarrow^* & [(\mathsf{true}, 1)]
\end{array}
$$

Figure 4.8: `once` staying in its scope forwarded by **fwd**

First we removed rule T-EqV, which re-types expressions to some equivalent type. This rule is used to make types line up exactly at the site of applications, for example by changing the order of the labels in effect rows. As a consequence, in the syntax directed version we essentially inline T-EqV wherever it is needed.

Secondly, we removed the rules dealing with generalisation and instantiation (T-Inst, T-InstEff, T-Gen and T-GenEff). Instead, whenever rules insist on some kind of polymorphism on some subderivation, we extend the environment with fresh type variables, and generalize over them locally, instead of via axillary rules.

Finally, as dealing with higher-kinded polymorphism is orthogonal to our work (and real programming languages like Haskell and OCaml already have their solutions for higher-kinded polymorphism [97]), we avoid higher-order unification by annotating handlers with the type operator they apply (e.g. $\mathbf{handler}_M\ \{\ldots\}$ instead of $\mathbf{handler}\ \{\ldots\}$).

4.8.2 Safety

The type-and-effect system of λ_{sc} is type safe, which we show by proving Subject Reduction and Progress. Here we briefly state the theorems; the proofs and used lemmas can be found in Appendix E.

As values are inert, these theorems range over computations only. The formulation of Subject Reduction (Theorem 4.8.1) is standard. Furthermore, apart from an additional normal form $\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, Progress (Theorem 4.8.2) is standard as well. Note that Progess implies effect safety, which intuitively means that only operations tracked in the effect type can be invoked.

Theorem 4.8.1 (Subject Reduction)**.** *If* $\Gamma \vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$*, then there exists a* $\underline{C}'$ *such that* $\underline{C} \equiv \underline{C}'$ *and* $\Gamma \vdash c' : \underline{C}'$*.*

Theorem 4.8.2 (Progress)**.** *If* $\cdot \vdash c : A!\langle E\rangle$*, then either:*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is in a* normal form*, which means it is in one of the following forms: (1)* $c = \mathbf{return}\ v$*, (2)* $c = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c')$ *where* $\ell^{\mathsf{op}} \in E$*, or (3)* $c = \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ *where* $\ell^{\mathsf{sc}} \in E$*.*

4.9 Examples & Implementation

Now that we have formalized the calculus we can cover some examples. This serves two purposes. First, we will highlight how the conventional encoding of scoped effects as handlers, the solution proposed in Plotkin and Power [59], is not expressive enough, even though it is applied in the real world [82]. We have postponed doing so, because now that we have formally introduced a calculus, we can immediately show how λ_{sc} addresses these issues. The first two examples in this section (exceptions with catch and reader with local) therefore contain both an attempt at encoding them as an handler, as well as a proper encoding as a **sc** in λ_{sc}. Secondly, the examples exemplify the expressivity of λ_{sc}.

Syntactic sugar To enhance readability, we write the examples in a higher-level syntax following Eff's conventions: we use top-level definitions, coalesce values and computations, implicitly sequence steps and insert **return** where needed. Furthermore, we drop trivial **return** continuations of operations:

$$\begin{aligned} \mathbf{op}\ \ell^{\mathsf{op}}\ x &\equiv \mathbf{op}\ \ell^{\mathsf{op}}\ x\ (y\,.\,\mathbf{return}\ y) \\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ (y\,.\,c_1) &\equiv \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ (y\,.\,c_1)\ (z\,.\,\mathbf{return}\ z) \end{aligned}$$

Implementation A prototype implementation of an interpreter for λ_{sc} is available at `https://github.com/thwfhk/lambdaSC`. This implementation contains a Hindley-Milner [50] type inference algorithm for λ_{sc}, which is mostly an extension of the type inference of Koka [41]. There are two main things that we need to additionally deal with for scoped effects. One is that we explicit force all handlers to be polymorphic. Note that there is no requirement for undecidable polymorphic recursion; deep handlers are implicitly recursive and always given polymorphic types. The other is that we require handlers to be annotated with the type operators that they use for their carriers in order to avoid higher-order unification. With explicit annotations for type operators, our unification algorithm just needs to reduce types first before unifying. The details can be found in the code.

Each of the examples covered in the rest of this section have been implemented. See the readme of the implementation's repository for more information.

4.9.1 Exceptions

Wu et al. [93] have shown how to catch exceptions with a scoped operation. Raising an exception is an algebraic operation $\texttt{raise} : \mathsf{String} \twoheadrightarrow \mathsf{Empty}$, and

catching an exception is a scoped operation $\texttt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}$. For example, consider that we are dealing with a counter with a maximum value of 10. The following computation increases the counter by 1 and raises an exception when the counter exceeds 10:

$$\begin{aligned} \mathit{incr} = {} & \mathbf{do}\ x \leftarrow \mathbf{op}\ \texttt{inc}\ ()\,; \\ & \mathbf{if}\ x > 10\ \mathbf{then}\ \mathbf{op}\ \texttt{raise}\ \texttt{"Overflow"}\ (y\,.\,\mathsf{absurd}\ y)\ \mathbf{else}\ \mathbf{return}\ x \end{aligned}$$

Clearly, if we start with a state of 8 and call `inc` thrice, we end up with an exception. We want to define a `catch` operation that executes an alternative computation when an exception is thrown inside its scope.

Catch as handler

One might attempt to write catch as a handler. However, as we will see, this method does not have the same modularity and expressivity as our calculus because it cannot achieve the local update semantics [93].

$$\begin{aligned} h_{\mathsf{except}\text{✗}} &= \mathbf{handler}\ \{\mathbf{return}\ x \mapsto \mathsf{right}\ x, \mathbf{op}\ \texttt{raise}\ e\ \_ \mapsto \mathsf{left}\ e\} \\ \texttt{catch}\text{✗}\ c_1\ c_2 \ &\equiv\ \mathbf{with\ handler}\ \{\mathbf{return}\ x \quad \mapsto \mathbf{return}\ x \\ &\qquad\qquad\qquad\qquad\qquad\qquad , \mathbf{op}\ \texttt{raise}\ \_\ \_ \mapsto c_2 \qquad\quad \}\ \mathbf{handle}\ c_1 \\ c_{\mathsf{catch}\text{✗}} &= \mathbf{do}\ \mathit{incr}\,; \texttt{catch}\text{✗}\ (\mathbf{do}\ \mathit{incr}\,; \mathbf{do}\ \mathit{incr}\,; \mathbf{return}\ \texttt{"success"}) \\ &\qquad\qquad\qquad\qquad (\mathbf{return}\ \texttt{"fail"}) \end{aligned}$$

The $\texttt{catch}\text{✗}$ implements the `catch` operation as a handler. The $h_{\mathsf{except}\text{✗}}$ interprets exceptions using a sum type. It is used to handle potential exceptions not captured by $\texttt{catch}\text{✗}$. By handling exceptions before state we obtain global update semantics where the state updates are not discarded when an error is raised:

$$\mathit{run}_{\mathsf{inc}}\ 8\ (\mathbf{with}\ h_{\mathsf{except}\text{✗}}\ \mathbf{handle}\ c_{\mathsf{catch}\text{✗}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$$

When handling exceptions *after* state, we would expect *local* update semantics, where we discard the state updates when an error is raised. Thus, the expected result should be $\mathsf{right}\ (\texttt{"fail"}, 9)$. However, we again get the global update semantics:

$$\mathbf{with}\ h_{\mathsf{except}\text{✗}}\ \mathbf{handle}\ (\mathit{run}_{\mathsf{inc}}\ 8\ c_{\mathsf{catch}\text{✗}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$$

How can this be? By implementing $\texttt{catch}\text{✗}$ as a handler, we have lost the separation between syntax and semantics: $\texttt{catch}\text{✗}$ is supposed to denote syntax, but it contains semantics in the form of a handler. Since we apply $\texttt{catch}\text{✗}$ to a computation ($c_{\mathsf{catch}\text{✗}}$), any containing `raise` will have already been handled by

$\texttt{catch}_{\cancel{x}}$ before h_{inc} is applied. In other words, we have lost modular composition because the handler of `raise` always come before the handler of `inc`. As a result, we cannot change the semantics of the interactions of effects by swaping the order of their handlers, which is certainly harmful to expressivity.

Catch as scoped effect

Let us implement `catch` as a scoped operation in λ_{sc}.

$$
\begin{aligned}
c_{\mathsf{catch}} = \mathbf{do}\ \mathit{incr}\,;\mathbf{sc}\ \texttt{catch}\ \texttt{"Overflow"}\ (b\,.\\
\quad \mathbf{if}\ b\ \mathbf{then}\ (\mathbf{do}\ \mathit{incr}\,;\mathbf{do}\ \mathit{incr}\,;\mathbf{return}\ \texttt{"success"})\\
\quad \mathbf{else}\ \mathbf{return}\ \texttt{"fail"})
\end{aligned}
$$

The scoped computation's true branch is the program that may *raise* exceptions, while the false branch *deals with* the exception. Our handler interprets exceptions in terms of a sum type $\mathbf{data}\ \alpha + \beta = \mathsf{left}\ \alpha \mid \mathsf{right}\ \beta$, where $\mathsf{left}\ v$ denotes an exception and $\mathsf{right}\ v$ a result.

$$
\begin{aligned}
&h_{\mathsf{except}} : \forall\, \alpha\ \mu\,.\,\alpha\,!\,\langle \texttt{raise}\,;\texttt{catch}\,;\mu\rangle \Rightarrow \mathsf{String} + \alpha\,!\,\langle\mu\rangle\\
&h_{\mathsf{except}} = \mathbf{handler}\\
&\quad \{\mathbf{return}\ x \mapsto \mathsf{right}\ x\\
&\quad ,\mathbf{op}\ \texttt{raise}\ e\ \_ \mapsto \mathsf{left}\ e\\
&\quad ,\mathbf{sc}\ \texttt{catch}\ e\ p\ k \mapsto \mathbf{do}\ x \leftarrow p\ \mathsf{true}\,;\\
&\qquad\qquad\qquad\qquad \mathbf{case}\ x\ \mathbf{of}\ \mathsf{left}\ e' \mid e' = e \to \mathsf{exceptMap}\ (p\ \mathsf{false})\ k\\
&\qquad\qquad\qquad\qquad\qquad\qquad \_ \qquad\qquad \to \mathsf{exceptMap}\ x\ k\\
&\quad ,\mathbf{bind}\ x\ k \mapsto \mathsf{exceptMap}\ x\ k\}
\end{aligned}
$$

The return clause and algebraic operation clause for `raise` construct a return value and raise an exception e by calling the right and left constructors, respectively. The scoped operation clause for `catch` catches an exception e. If the scoped computation in $p\ \mathsf{true}$ raises an exception e, it is caught by `catch` and replaced by the scoped computation $(p\ \mathsf{false})$. Otherwise, it continues with $p\ \mathsf{true}$ and its results are passed to the continuation k. The forwarding clause follows the same principle as that of how we write the forwarding clause for h_{inc} in Section 4.3.3. We just need to think about how to continue with k when we have the result x that potentially fails. The most intuitive way is to return the exception if x fails ($\mathsf{left}\ e$), and we run the continuation k with the result if x succeeds ($\mathsf{right}\ y$).

$$
\begin{aligned}
&\mathsf{exceptMap} : \forall\, \alpha\ \beta\ \mu\,.\,\mathsf{String} + \beta \to^{\mu} (\beta \to^{\mu} \mathsf{String} + \alpha) \to^{\mu} \mathsf{String} + \alpha\\
&\mathsf{exceptMap}\ x\ k = \mathbf{case}\ x\ \mathbf{of}\ \mathsf{left}\ e \to \mathsf{left}\ e\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{right}\ y \to k\ y
\end{aligned}
$$

Given an initial counter value 8, we can handle the program c_{catch} with h_{except} and h_{inc}. Different orders of the application of handlers give us different semantics of the interaction of effects [93]. Handling exceptions before increments gives us global updates:

$$run_{\mathsf{inc}}\ 8\ (\mathbf{with}\ h_{\mathsf{except}}\ \mathbf{handle}\ c_{\mathsf{catch}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$$

Although an exception is raised and caught, the final value is still updated to 11 by the two `inc` operations and exceeds the maximum value of our counter. When handling exceptions after increments, we obtain the expected local update semantics:

$$\mathbf{with}\ h_{\mathsf{except}}\ \mathbf{handle}\ (run_{\mathsf{inc}}\ 8\ c_{\mathsf{catch}}) \rightsquigarrow^* \mathsf{right}\ (\texttt{"fail"}, 9)$$

The state updates introduced by the two invocations of `inc` inside the scope of `catch` are discarded. This correctly reflects our intuition when the handler of `inc` comes before the handler of `raise`.

4.9.2 Reader with Local

Reader entails an `ask` operation that lets one read the (integer) state that is passed around. The scoped effect `local` takes a function f which alters the state, and a computation for which the state should be altered, after which the state should be returned to its original state.[3] For example, in $\mathbf{sc}\ \texttt{local}\ (\boldsymbol{\lambda} i\,.\,i*2)\ (\mathbf{op}\ \texttt{ask}\ ())\ (\mathbf{op}\ \texttt{ask}\ ())$, the first ask receives a state that is doubled, whereas the second ask receives the original state. To exemplify the problems that arise when implementing `local` as a handler, our example uses effect `foo`, which is simply mapped to `ask` by h_{foo}:

$$\begin{aligned} h_{\mathsf{foo}} = \mathbf{handler}\ \{&\mathbf{return}\ x \quad \mapsto \mathbf{return}\ x \\ ,&\mathbf{op}\ \texttt{foo}\ \_\ k \mapsto \mathbf{do}\ x \leftarrow \mathbf{op}\ \texttt{ask}\ ()\ (y\,.\,k\ y) \\ ,&\mathbf{bind}\ x\ k \quad \mapsto k\ x\} \end{aligned}$$

Local as a handler

Whereas the lack of effect interaction control in example of `catch` as a handler could be described as unfortunate, in the case for ask there is arguably only

[3] The operation signature of `local` requires polymorphic parameter types like $\texttt{local} : (\forall\ \mu\,.\,\mathsf{Int} \to^{\mu} \mathsf{Int}) \to\!\triangleright\ ()$, which we do not support. It is easy to extend operations in λ_{sc} with prenex polymorphic parameter types without any need of other mechanism for higher-rank polymorphism.

one correct interaction, which is not the one that arises from scoped effects as handlers. Consider c_{local} below, which includes `foo`, which is mapped to `ask` by h_{foo}.

$$
\begin{aligned}
&\texttt{local}_{\textbf{✗}}\ f\ c\ \equiv\ \textbf{with handler}\ \{\textbf{return}\ x \mapsto \textbf{return}\ x \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \textbf{op}\ \texttt{ask}\ \_ \mapsto x \leftarrow \texttt{ask}\,;f\ x\}\ \textbf{handle}\ c \\
&h_{\mathsf{read}\textbf{✗}} = \textbf{handler}\ \{\textbf{return}\ x \mapsto \boldsymbol{\lambda} s\,.\,x, \textbf{op}\ \texttt{ask}\ \_\ k \mapsto \boldsymbol{\lambda} s\,.\,k\ s\ s\} \\
&\mathit{run}_{\mathsf{read}\textbf{✗}}\ s\ c\ \equiv\ \textbf{do}\ c' \leftarrow \textbf{with}\ h_{\mathsf{read}\textbf{✗}}\ \textbf{handle}\ c\,;c'\ s \\
&c_{\mathsf{local}\textbf{✗}} = \textbf{do}\ x \leftarrow \textbf{op}\ \texttt{ask}\ ()\,;\textbf{do}\ y \leftarrow \textbf{op}\ \texttt{foo}\ () \\
&\qquad\quad \texttt{local}_{\textbf{✗}}\ (\boldsymbol{\lambda} a \to 2 * a) \\
&\qquad\qquad\qquad (z \leftarrow \textbf{op}\ \texttt{ask}\ ()\,;u \leftarrow \textbf{op}\ \texttt{foo}\ ()\,;\textbf{return}\ (x,y,z,u))
\end{aligned}
$$

Since h_{foo} introduces `ask`, we must (re)apply h_{read} after applying h_{foo}. Since `foo` is mapped to `ask`, in $c_{\mathsf{local}\textbf{✗}}$ we expect x to be equal to y, and z equal to u. Starting with the reader state set to 1, we expect the result $(1,1,2,2)$. Instead, we get:

$$\mathit{run}_{\mathsf{read}}\ 1\ (\textbf{with}\ h_{\mathsf{foo}}\ \textbf{handle}\ c_{\mathsf{local}\textbf{✗}}) \rightsquigarrow^{*} \textbf{return}\ (1,1,2,1)$$

Again, how can this be? The cause is the same as the example with `catch`: since we encode the semantics of $\texttt{local}_{\textbf{✗}}$ in its definition, we are forced to perform the handling at the moment of application. Notice that `foo` is not caught by $\texttt{local}_{\textbf{✗}}$! Therefore, f is only applied to `ask`. When `foo` is mapped to `ask` by h_{foo}, $\texttt{local}_{\textbf{✗}}$'s effect will already have been triggered, which is why f is not applied to it.

Local as a scoped effect

Using a scoped effect we can properly encode `local`:

$$
\begin{aligned}
&h_{\mathsf{read}}\ :\ \forall\ \alpha\ \mu\,.\,\alpha\,!\,\langle \texttt{ask}\,;\texttt{local}\,;\mu\rangle \Rightarrow (\mathsf{Int} \to^{\mu} \alpha)\,!\,\langle\mu\rangle \\
&h_{\mathsf{read}} = \textbf{handler}\ \{\textbf{return}\ x \qquad\quad \mapsto \boldsymbol{\lambda} s\,.\,x \\
&\qquad\qquad\qquad , \textbf{op}\ \texttt{ask}\ \_\ k \qquad \mapsto \boldsymbol{\lambda} s\,.\,k\ s\ s \\
&\qquad\qquad\qquad , \textbf{sc}\ \texttt{local}\ f\ p\ k \mapsto \boldsymbol{\lambda} s\,.\,\textbf{do}\ x \leftarrow p\ ()\ (f\ s)\,;k\ x\ s \\
&\qquad\qquad\qquad , \textbf{fwd}\ f\ p\ k \qquad \mapsto \boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ s, \boldsymbol{\lambda} z\,.\,k\ z\ s)\} \\
&\mathit{run}_{\mathsf{read}}\ s\ c\ \equiv\ \textbf{do}\ c' \leftarrow \textbf{with}\ h_{\mathsf{read}}\ \textbf{handle}\ c\,;c'\ s \\
&c_{\mathsf{local}} = \textbf{do}\ x \leftarrow \textbf{op}\ \texttt{ask}\ ()\,;\textbf{do}\ y \leftarrow \textbf{op}\ \texttt{foo}\ ()\,; \\
&\qquad\quad \textbf{sc}\ \texttt{local}\ (\boldsymbol{\lambda} a \to 2 * a) \\
&\qquad\qquad\qquad (\textbf{do}\ z \leftarrow \textbf{op}\ \texttt{ask}\ ()\,;\textbf{do}\ u \leftarrow \textbf{op}\ \texttt{foo}\ ()\,;\textbf{return}\ (x,y,z,u))
\end{aligned}
$$

Note that the forwarding clause of h_{read} is similar to that of h_{inc} in Section 4.7.4 except for passing the same state to the scoped computation p and continuation

k. This makes natural sense because we definitely do not want the scopes of other irrelevant operations to change what can be read. Since `local` is now purely syntactic, we can apply h_{foo} before h_{read}, and have h_{read} handle the `ask` that h_{foo} outputs:

$$
\begin{aligned}
& \mathit{run}_{\mathsf{read}}\ 1\ (\mathbf{with}\ h_{\mathsf{foo}}\ \mathbf{handle}\ c_{\mathsf{local}}) \\
\rightsquigarrow\ & \mathit{run}_{\mathsf{read}}\ 1\ (x \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,;\, y \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \\
& \qquad \mathbf{sc}\ \mathtt{local}\ (\boldsymbol{\lambda} a \to 2 * a)\ (\mathbf{do}\ z \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \\
& \qquad\quad \mathbf{do}\ u \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \mathbf{return}\ (x, y, z, u))) \\
\rightsquigarrow\ & \mathbf{return}\ (1, 1, 2, 2)
\end{aligned}
$$

4.9.3 Nondeterminism with Cut

The algebraic operation $\mathtt{cut} : () \twoheadrightarrow ()$ provides a different flavor of pruning nondeterminism that has its origin as a Prolog primitive. The idea is that `cut` prunes all remaining branches and only allows the current branch to continue. Typically, we want to keep the effect of `cut` local. This is achieved with the scoped operation $\mathtt{call} : () \twoheadrightarrow ()$, as proposed by Wu et al. [93]. To handle `cut` and `call`, we use the CutList datatype [57].

$$\mathbf{data}\ \mathsf{CutList}\ \alpha = \mathsf{opened}\ (\mathsf{List}\ \alpha)\ \mid\ \mathsf{closed}\ (\mathsf{List}\ \alpha)$$

We can think of $\mathsf{opened}\ v$ as a list that may be extended and $\mathsf{closed}\ v$ as a list that may not be extended with further elements. This intention is captured in the $\mathsf{append}_{\mathsf{CutList}}$ function, which discards the second list if the constructor of the first list is closed.

$$
\begin{aligned}
& \mathsf{append}_{\mathsf{CutList}} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
& \mathsf{append}_{\mathsf{CutList}}\ (\mathsf{opened}\ xs)\ (\mathsf{opened}\ ys) = \mathsf{opened}\ (xs +\!\!+ ys) \\
& \mathsf{append}_{\mathsf{CutList}}\ (\mathsf{opened}\ xs)\ (\mathsf{closed}\ ys)\ \ = \mathsf{closed}\ \ (xs +\!\!+ ys) \\
& \mathsf{append}_{\mathsf{CutList}}\ (\mathsf{closed}\ xs)\ \ \_\qquad\qquad\ = \mathsf{closed}\ \ xs
\end{aligned}
$$

The handler for nondeterminism with cut is defined as follows:

$$
\begin{aligned}
& h_{\mathsf{cut}}\ :\ \forall\ \alpha\ \mu\,.\,\alpha\,!\,\langle \mathtt{choose}\,;\mathtt{fail}\,;\mathtt{cut}\,;\mathtt{call}\,;\mu\rangle \Rightarrow \mathsf{CutList}\ \alpha\,!\,\langle\mu\rangle \\
& h_{\mathsf{cut}} = \mathbf{handler}\ \{\mathbf{return}\ x \qquad\quad \mapsto \mathsf{opened}\ [x] \\
& \qquad\qquad ,\mathbf{op}\ \mathtt{fail}\ \_\ \_ \quad \mapsto \mathsf{opened}\ [\,] \\
& \qquad\qquad ,\mathbf{op}\ \mathtt{choose}\ x\ k \mapsto \mathbf{do}\ xs \leftarrow k\ \mathsf{true}\,; \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\ \mathsf{isclose}\ xs\ \mathbf{then}\ xs \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathsf{append}_{\mathsf{CutList}}\ xs\ (k\ \mathsf{false}) \\
& \qquad\qquad ,\mathbf{op}\ \mathtt{cut}\ \_\ k \quad \mapsto \mathsf{close}\ (k\ ())
\end{aligned}
$$

$$
\begin{array}{ll}
, \mathbf{sc}\ \texttt{call}\ \_\ p\ k & \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{open}\ (p\ ()))\ k \\
, \mathbf{bind}\ x\ k & \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ x\ k\}
\end{array}
$$

The operation clause for `cut` closes the cutlist and the clause for `call` (re-)opens it when coming out of the scope.

$$
\begin{array}{ll}
\mathsf{close} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha & \mathsf{open} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{close}\ (\mathsf{closed}\ as) = \mathsf{closed}\ as & \mathsf{open}\ (\mathsf{closed}\ as) = \mathsf{opened}\ as \\
\mathsf{close}\ (\mathsf{opened}\ as) = \mathsf{closed}\ as & \mathsf{open}\ (\mathsf{opened}\ as) = \mathsf{opened}\ as
\end{array}
$$

The function isclose checks whether a cutlist is closed. It is used in the `choose` clause for efficiency; we do not need to execute k false when k true returns a closed list.

$$
\begin{array}{l}
\mathsf{isclose} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{isclose}\ (\mathsf{closed}\ \_) = \mathsf{true} \\
\mathsf{isclose}\ (\mathsf{opened}\ \_) = \mathsf{false}
\end{array}
$$

The forwarding of h_{cut} uses the function $\mathsf{concatMap}_{\mathsf{CutList}}$, the cutlist counterpart of concatMap which takes the extensibility of CutList (signalled by opened and closed) into account when concatenating.

$$
\begin{array}{l}
\mathsf{concatMap}_{\mathsf{CutList}} : \forall\ \alpha\ \beta\ \mu\,.\,\mathsf{CutList}\ \beta \to^{\mu} (\beta \to^{\mu} \mathsf{CutList}\ \alpha) \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ [\,])\ f = \mathbf{return}\ (\mathsf{opened}\ [\,]) \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ [\,])\ f = \mathbf{return}\ (\mathsf{closed}\ [\,]) \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ (b : bs))\ f = \mathbf{do}\ as \leftarrow f\ b\,; \\
\qquad\qquad as' \leftarrow \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ bs)\ f\,; \\
\qquad\qquad \mathsf{append}_{\mathsf{CutList}}\ as\ as' \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ (b : bs))\ f = \mathbf{do}\ as \leftarrow f\ b\,; \\
\qquad\qquad as' \leftarrow \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ bs)\ f\,; \\
\qquad\qquad \mathsf{append}_{\mathsf{CutList}}\ as\ as'
\end{array}
$$

In Section 4.9.5, we give an example usage of `cut` to improve parsers.

4.9.4 Depth-Bounded Search

The handler h_{ND} for nondeterminism shown in Section 4.2 implements the *depth-first search* (DFS) strategy. However, with scoped effects and handlers we can implement other search strategies, such as *depth-bounded search* (DBS) [98], which uses the scoped operation $\texttt{depth} : \mathsf{Int} \rightarrowtriangle ()$ to bound the depth of the branches in the scoped computation. The handler uses return type

$\mathsf{Int} \to^{\mu} \mathsf{List}\ (\alpha, \mathsf{Int})$. Here, the Int parameter is the current depth bound, and the result is a list of (α, Int) pairs, where α denotes the result and Int reflects the remaining global depth bound.[4]

$$
\begin{aligned}
&h_{\mathsf{depth}}\ :\ \forall\ \alpha\ \mu\ .\ \alpha\ !\ \langle \texttt{choose}\,;\texttt{fail}\,;\texttt{depth}\,;\mu\rangle \Rightarrow (\mathsf{Int} \to^{\mu} \mathsf{List}\ (\alpha, \mathsf{Int}))\,!\,\langle\mu\rangle\\
&h_{\mathsf{depth}} = \mathbf{handler}\\
&\quad \{\mathbf{return}\ x \qquad\quad \mapsto \boldsymbol{\lambda} d\ .\ [(x, d)]\\
&\quad , \mathbf{op}\ \texttt{fail}\ \_\ \_ \quad \mapsto \boldsymbol{\lambda} \_\ .\ [\,]\\
&\quad , \mathbf{op}\ \texttt{choose}\ x\ k \ \mapsto \boldsymbol{\lambda} d\ .\ \mathbf{if}\ d = 0\ \mathbf{then}\ [\,]\\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\ k\ \mathsf{true}\ (d-1)\ +\!\!+ k\ \mathsf{false}\ (d-1)\\
&\quad , \mathbf{sc}\ \texttt{depth}\ d'\ p\ k \mapsto \boldsymbol{\lambda} d\ .\ \mathsf{concatMap}\ (p\ ()\ d')\ (\boldsymbol{\lambda}(v, \_)\ .\ k\ v\ d)\\
&\quad , \mathbf{fwd}\ f\ p\ k \qquad \mapsto \boldsymbol{\lambda} d\ .\ f\ (\boldsymbol{\lambda} y\ .\ p\ y\ d, \boldsymbol{\lambda} vs\ .\ \mathsf{concatMap}\ vs\ (\boldsymbol{\lambda}(v, d)\ .\ k\ v\ d))\}
\end{aligned}
$$

For the `depth` operation, we locally use the given depth bound d' for the scoped computation p and go back to using the global depth bound d for the continuation k. In case of an unknown scoped operation, the forwarding clause just threads the depth bound through, first into the scoped computation and from there into the continuation. It is similar to a combination of the forwarding of h_{once} and h_{inc}.

For example, the following program (Figure 4.9) has a local depth bound of 1 and a global depth bound of 2. It discards the results 2 and 3 in the scoped computation as they appear after the second `choose` operation, and similarly, the results 5 and 6 in the continuation are ignored.

$$
\begin{aligned}
c_{\mathsf{depth}} = \mathbf{sc}\ \texttt{depth}\ 1\\
\quad (\_\ .\ &\mathbf{do}\ b_1 \leftarrow \mathbf{op}\ \texttt{choose}\ ()\,;\mathbf{if}\ b_1\ \mathbf{then\ return}\ 1\ \mathbf{else}\\
&\quad \mathbf{do}\ b_2 \leftarrow \mathbf{op}\ \texttt{choose}\ ()\,;\mathbf{if}\ b_2\ \mathbf{then\ return}\ 2\ \mathbf{else\ return}\ 3)\\
\quad (x\ .\ &\mathbf{do}\ b_1 \leftarrow \mathbf{op}\ \texttt{choose}\ ()\,;\mathbf{if}\ b_1\ \mathbf{then\ return}\ x\ \mathbf{else}\\
&\quad \mathbf{do}\ b_2 \leftarrow \mathbf{op}\ \texttt{choose}\ ()\,;\mathbf{if}\ b_2\ \mathbf{then\ return}\ 4\ \mathbf{else}\\
&\quad \mathbf{do}\ b_3 \leftarrow \mathbf{op}\ \texttt{choose}\ ()\,;\mathbf{if}\ b_3\ \mathbf{then\ return}\ 5\ \mathbf{else\ return}\ 6)
\end{aligned}
$$

$$
\begin{aligned}
&\gg (\mathbf{with}\ h_{\mathsf{depth}}\ \mathbf{handle}\ c_{\mathsf{depth}})\ 2\\
&[(1,1),(4,0)]
\end{aligned}
$$

The result is $[(1,1),(4,0)]$, where the tuple's second parameter represents the global depth bound. Notice that `choose` operations in the scoped computation `depth` do not consume the global depth bound in the handler. We also show an alternative implementation of the handler clause for `depth` in the Supplementary Material.

[4] The pair type (α, Int) differs from Yang et al. [98]'s α in order to enable forwarding.

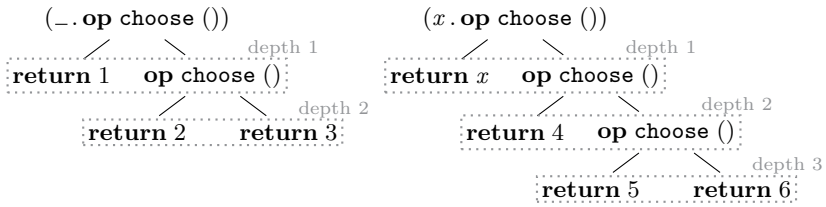

Figure 4.9: Visual representation of c_{depth}.

4.9.5 Parsers

A parser effect can be achieved by combining the nondeterminism-with-cut effect and a token-consuming effect [93]. The latter features the algebraic operation $\mathtt{token} : \mathsf{Char} \twoheadrightarrow \mathsf{Char}$ where $\mathbf{op}\ \mathtt{token}\ t$ consumes a single character from the implicit input string; if it is t, it is passed on to the continuation; otherwise the operation fails. The token handler has result type $\mathsf{String} \to^{\langle \mathtt{fail};\mu\rangle} (\alpha, \mathsf{String})$: it threads through the remaining part of the input string. Observe that the function type signals it may `fail`, in case the token does not match.

$$
\begin{aligned}
&h_{\mathsf{token}} \ :\ \forall\, \alpha\ \mu\,.\,\alpha\,!\,\langle \mathtt{token};\mathtt{fail};\mu\rangle \Rightarrow (\mathsf{String} \to^{\langle \mathtt{fail};\mu\rangle} (\alpha, \mathsf{String}))\,!\,\langle \mathtt{fail};\mu\rangle \\
&h_{\mathsf{token}} = \mathbf{handler} \\
&\quad \{\mathbf{return}\ x \qquad \mapsto \boldsymbol{\lambda} s\,.\,(x, s) \\
&\quad , \mathbf{op}\ \mathtt{token}\ x\ k \mapsto \boldsymbol{\lambda} s\,.\,\mathbf{case}\ s\ \mathbf{of}\ [\,] \qquad \to \mathsf{failure}\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (x' : xs) \to \mathbf{if}\ x = x'\ \mathbf{then}\ k\ x\ xs \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathsf{failure}\ () \\
&\quad , \mathbf{fwd}\ f\ p\ k \qquad \mapsto \boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ s, \boldsymbol{\lambda}(t, s')\,.\,k\ t\ s')\}
\end{aligned}
$$

The forwarding clause of h_{token} is the same as that of h_{inc} in Section 4.7.4. We uses the initial state s for the scoped computation p, and the updated state s' for the continuation k. The updated state s' is passed in from the scoped computation p and reflects the consumption of tokens in p. An alternative forwarding semantics is $\mathbf{fwd}\ f\ p\ k \mapsto \boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ s, \boldsymbol{\lambda}(t, s')\,.\,k\ t\ s)$, where the continuation k also uses the initial state s. In this way, we actually discard all the changes to the state in the scoped computation, which could potentially be useful to implement parser lookahead. It is up to the programmer to decide which semantics they want.

We give an example parser for a small expression language, in the typical parser combinator style, built on top of the token-consumer and nondeterminism. For convenience, it uses the syntactic sugar $x \diamond y \quad \equiv \mathbf{op}\ \mathtt{choose}\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)$.

```
digit     : ∀ μ . () → Char ! ⟨token ; choose ; μ⟩
digit _   = op token '0' ◇ op token '1' ◇ ... ◇ op token '9'
many1     : ∀ α μ . (() →μ α) →μ List α
many1 p = do a ← p () ; do as ← many1 p ◇ return [] ; return (a : as)
expr'     : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
expr' _   = (do i ← term' () ; do op token '+' ; do j ← expr' () ; return
            (i + j)) ◇ (do i ← term' () ; return i)
term'     : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
term' _   = (do i ← factor () ; do op token '*' ; do j ← term () ;
            return (i * j)) ◇ (do i ← factor () ; return i)
factor    : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
factor _  = (do ds ← many1 digit ; return (read ds))
            ◇ (do op token '(' ; do i ← expr' () ; do op token ')' ; return i)
```

The expr′ and term′ parsers are naive and can be improved by two steps of refactoring: (1) factoring out the common prefix in the two branches, and (2) pruning the second branch when the first branch successfully consumes a + or ∗, respectively. The optimised versions of expr, term, and factor are shown as follows using the cut operation defined in Section 4.9.3.

```
expr    : ∀ μ . () → Int ! ⟨token ; choose ; cut ; μ⟩
expr _ = do i ← term () ;
             sc call () (_ . (do op token '+' ; do op cut () ;
                              do j ← expr () ; return (i + j)) ◇ i)
term    : ∀ μ . () → Int ! ⟨token ; choose ; cut ; μ⟩
term _ = do i ← factor () ;
             sc call () (_ . (do op token '*' ; do op cut () ;
                              do j ← term () ; return (i * j)) ◇ i)
```

Here is how we invoke the optimised parser on an example input.

```
»> with h_cut handle (with h_token handle expr ()) "(2+5)*8"
opened [(56, "")]
```

Note that only the fully parsed result is returned because in expr and term we prune other branches when the first branch succeeds. If we parse the same input using the unoptimised parser expr′, we also get partially parsed results.

```
»> with h_cut handle (with h_token handle expr' ()) "(2+5)*8"
opened [(56, ""), (7, "*8")]
```

4.10 Related Work

In this section, we discuss related work on algebraic effects, scoped effects, and effect systems.

4.10.1 Algebraic Effects & Handlers

Many research languages for algebraic effects and handlers have been proposed, including Eff [5, 63], Frank [47], Effekt [12], or have been extended to include them, such as Links [30] and Koka [42]. OCaml [72] is the first industrial language supporting algebraic effects and handlers. Although it is possible to use handlers or write algebraic operations with function parameters to simulate the behaviours of scoped effects in these languages, none of them can achieve the expressiveness of real scoped effects & handlers as we have compared in Section 4.9.1 and Section 4.9.2.

There are also many packages for writing effect handlers in general purpose languages like Haskell and OCaml [38, 39, 66, 48, 37]. Yet, as far as we know, λ_{sc} is the first *calculus* that supports scoped effects & handlers.

4.10.2 Effect Systems

Most languages with support for algebraic effects are equipped with an effect system to keep track of the effects that are used in the programs. There is already much work on different approaches to effect systems for algebraic effects.

Eff [5, 63] uses an effect system based on subtyping relations. Each type of computation is decorated with an effect type Δ to represent the set of operations that might be invoked. The subtyping relations are used to extend the effect type Δ with other effects, which makes it possible to compose programs in a modular way. The implementation of such a subtyping system is orthogonal to the implementation of scoped effects, which is why we have opted for Koka-style row polymorphisms, which has yielded a simpler type system. Furthermore, supporting both polymorphism and non-trivial subtyping would complicate the type inference a lot as shown in [62] and [78] which use type inference with constraints and qualified types, respectively. Moreover, since λ_{sc} has type operators, we need to additionally distinguish between the positive and negative positions in type operators and propagate subtyping relations through these positions properly. This would further complicate the formalisation of type system and type inference.

Row polymorphism is another mainstream approach to effect systems. Links [30] uses the Rémy-style row polymorphism [64], where the row types are able to represent the absence of labels and each label is restricted to appear at most once. Koka [42] uses row polymorphism based on scoped labels [40], which allows duplicated labels and as a result is easier to implement. We can use row polymorphism to write handlers that handle particular effects and forward other effects represented by a row variable. In λ_{sc}, we opted for an effect system similar to Koka's, mainly because of its brevity. We believe that the Links-style effect system should also work well with scoped effects.

4.10.3 Scoped Effects & Handlers

Wu et al. [93] first introduced the idea of scoped effects & handlers to solve the problem of separating syntax from semantics in programming with effects that delimit the scope. They proposed a syntax based on higher-order functors, which impose fewer restrictions on the shape of the signatures of scoped operations than λ_{sc}. Their work also considers the problem of modular composition of handlers, and presents a solution–called "weaving"–based on threading a handler state through unknown operations. This approach is rather ad-hoc; it is not a generalization of the forwarding approach for algebraic effects. This approach has been adopted by several Haskell packages [66, 48, 37].

Piróg et al. [56] and Yang et al. [98] have developed denotational semantic domains of scoped effects, backed by category theoretical models. The key idea is to generalize the denotational approach of algebraic effects & handlers that is based on free monads and their unique homomorphisms. Indeed, the underlying category can be seen as a parameter. Then, by shifting from the base category of types and functions to a different (indexed or functor) category, scoped operations and their handlers turn out to be "just" an instance of the generalized notion of algebraic operations and handlers with the same structure and properties. We focus on a calculus for scoped effects instead of the denotational semantics of scoped effects. We make a simplification with respect to Yang et al. [98] where we avoid duplication of the base algebra and endoalgebra (for the outer and inner scoped respectively), and thus duplication of the scoped effect clauses in our handlers. With respect to Piróg et al. [56], we specialize the generic endofunctor Γ with signatures $A_\ell \twoheadrightarrow B_\ell$ of endofunctors of the form $A \times (B \to -)$. Our λ_{sc} calculus uses a similar idea to the 'explicit substitution' monad of Piróg et al. [56], a generalization of Ghani and Uustalu's [26] monad of explicit substitutions where each operation is associated with two computations representing the computation in scope and out of the scope (continuation) respectively. Neither Piróg et al. [56] nor Yang et al.

[98] considered the modular composition of scoped effects or the forwarding of operations in their categorical models.

Yang and Wu [99] develop a framework for (generalized) monoids with operations, of which scope effects are an instance. They study the problem of semantic modularity in this framework, and have some generic results which can be applied to scoped effects. As far as we know, no language design or library implementation of scoped effects has resulted from this yet.

Lindley et al. [46] developed an equational reasoning framework for scoped effects based on parameterised algebraic theories [77]. They do not consider handlers or forwarding.

4.11 Conclusion and Future Work

In this work, we have presented λ_{sc}, a novel calculus in which scoped effects & handlers are built-in. We started from the core calculus of Eff, extended it with a row-based effect system in the style of Koka, and added primitive support for scoped operations and their handlers. We introduced novel forwarding clauses as a means to obtain the modular composition of handlers in the presence of scoped effects. Finally, we have demonstrated the usability of λ_{sc} by implementing a range of examples. We believe that the features to support scoped effect in λ_{sc} are orthogonal to other language features and can be added to any programming language with algebraic effects, polymorphism and type operators.

Scoped effects require *every* handler in λ_{sc} to be polymorphic and equipped with an explicit forwarding clause. This breaks backwards compatibility: calculi that support only algebraic effects, such as Eff, miss an explicit forwarding clause for scoped operations and allow monomorphic handlers. This problem can be easily mitigated by kinds and kind polymorphism. The core idea is that we extend λ_{sc} with two kinds op and sc for effect types, such that $\Gamma \vdash E : \mathsf{op}$ means effect type E only contains algebraic operations, and $\Gamma \vdash E : \mathsf{sc}$ means effect type E may contain some scoped operations. Then, for handlers of type $A!\langle E\rangle \Rightarrow M\ A!\langle F\rangle$ which lack forwarding clauses, we can just add the condition $\Gamma \vdash E : \mathsf{op}$ to their typing rules. We leave the full formalisation and implementation of it to future work.

Making scoped effects and handlers into practical languages is a pretty new research area full of potential. There is a lot of future work to do. Directions for future work include: extending λ_{sc} with shallow handlers [35, 32], named handlers [8, 95, 12], and other forms of higher-order effects [7] including latent effects [83] and parallel effects [96]; exploring the notion of named scoped effects

where scoped effects themselves generate fresh names for the operations in their scope in a similar style to named handlers; defining a CPS translation for scoped effects and handlers [33, 42]; developing control-flow linearity [78] for scoped effects to soundly extend λ_{sc} with linear types; exploring guiding principles and equational theories for writing and reasoning about forwarding clauses.

4.12 Scientific output

This chapter is based on the following article [10]:

> Bosman, R., van den Berg, B., Tang, W., and Schrijvers, T. A Calculus for Scoped Effects & Handlers. In Logical Methods in Computer Science, November 22, 2024, Volume 20, Issue 4

The design of the calculus was shared between the first three authors. Furthermore, I was responsible for the metatheory and final narrative.

Conclusion

Chapter 5

Conclusion

In this thesis we have utilized calculi to develop a formal understanding of type inference and elaboration, and scoped effects. In this chapter we summarize the contributions and discuss future work.

5.1 Summary of contributions

In this section we summarize the two parts of this thesis.

Part I: Type Inference & Elaboration

In Part I we have formalized an elaboration algorithm for the Hindley-Damas-Milner (HDM) system. We arrived at this elaboration algorithm in two steps.

Firstly, in Chapter 2 we observed that not every type inference algorithm is suitable for extension with elaboration. Specifically, the way a type inference algorithm deals with underconstrained types is important: in the case of type checking these can be ignored, while in the case of elaboration underconstrained types must be fully instantiated. We have made the following contributions:

- We establish the problem of fully grounding and offer a solution (Section 2.2);
- We present the declarative specification (Section 2.3) and algorithm $\mathcal{R}$ (Section 2.4), a fully grounding inference algorithm for the HDM system;

- We mechanical prove algorithm $\mathcal{R}$ correct (Sections 2.5 and 2.6).

In Chapter 3 we proceeded with the logical next step: the extension of algorithm $\mathcal{R}$ with elaboration. In this chapter we made the following contributions:

- We formalized a system that would serve as the elaboration target: System F, but with HDM types (Section 3.3);
- We extended both the declarative specification (Section 3.4) and the algorithm (Section 3.5) with elaboration, and revisited instantiation from algorithmic to declarative contexts (Section 3.6);
- We established the need for an equivalence relation on elaborated terms, and defined it in terms of a logical relation (Section 3.7);
- We revisited the correctness, extending the soundness and completeness statements with the nessicarly machinery for the elaborated terms (Section 3.8), and discussed the implementation (Section 3.9).

Part II: Scoped Effects

In Part II we presented λ_{sc}, a calculus for scoped effects & handlers. In this chapter we made the following contributions:

- We introduced algebraic effects & handlers, showcasing how they achieve their modularity and how effects can interact, and discussed machinery such as algebraic and forwarding. We observed that not all effects are algebraic, and distinguished a class of effects that have or create a delimited scope: scoped effects (Section 4.2).
- We discussed the design challenges of a calculus with scoped effects & handlers, and discussed how the scoped result type mandates polymorphic handling and explicit forwarding clauses for forwarding unknown operations (Section 4.3).
- We presented λ_{sc}, the first calculus featuring scoped effects & handlers, discussing its syntax (Section 4.4), operational semantics (Section 4.5), and type-and-effect system (Section 4.6). Finally, we generalized its forwarding clause (Section 4.7).
- We discussed λ_{sc}'s metatheory (Section 4.8) and gave a series of examples that demonstrate the expressivity of our calculus (Section 4.9).

5.2 Future work

Finally, we discuss the future work of the parts in isolation as well as the combination of inference/elaboration and (scoped) effects.

Mechanizing the elaboration proof The most obvious avenue of future work of our work on fully grounding type inference and elaboration in Part I is to mechanize the proof laid oud in Chapter 3. All definitions have been formalized and significant effort was put into the mechanization of the proofs, but much work remains to be done here. Furthermore, it would be interesting to incorporate (mechanically formalized) fully grounding inference/elaboration into elaboration based features such as typeclasses. For example, Bottu et al. [11] prove (on paper) the coherence of type class resolution. However, they do so in a bidirectional setting. We believe our algorithm forms can be extended to reproduce this result in the HDM system. As this is more representative of the real-world implementation of languages like Haskell, such result would further our confidence in the correctness of typeclasses.

Backwards compatibility of λ_{sc} In our section on the future work of our calculus for scoped effects & handlers (Section 4.11) we observed that our calculus is not backwards compatible, because we require *every* handler in λ_{sc} to be polymorphic and equipped with an explicit forwarding clause. This means that we cannot run programs made in calculi such as Eff that only have algebraic effects & handlers, because these handlers do not feature a forwarding clause. We already sketch a solution to this, which Jakob Heirwegh has investigated for their master's thesis [29]. What we sketch in Section 4.11 matches the approach we applied: we added a kind for scoped effects and for algebraic effects, two handler definitions, and two handler applications. Using this it is possible to be fully backwards compatible with calculi such as Eff.

Scoped effect subtyping with fully grounding inference Karachalias et al. [36] describe a type inference algorithm for their calculus featuring algebraic effects & handlers. Specifically they focus on developing subtyping for effects. This work is doubly interesting, because it combines the two parts of this thesis: inference/elaboration and effects. It would be interesting to combine scoped effects with explicit subtyping. In theory these should be orthogonal features, since the addition of scoped effects does not overhaul the reasoning about the types of effects in any significant way. More interestingly however is the inference of handler types. Since polymorphic handlers add type constructors

and scoped effects require polymorphic handlers, it would require dealing with higher-kinded polymorphism, a problem we already encountered and described in Section 4.8.1.

Since their unification, like algorithm $\mathcal{W}$, is free-variable-based, it is expected that their algorithm is not fully grounding (and therefore suffers from similar bugs to the ones laid oud in Chapter 2). Analysis of the typing rules confirmed that their algorithm indeed is not fully grounding.

Other non-algebraic effects In Part II we studied a particular non-algebraic class of effects: scoped effects. Yet, not all not-algebraic effects are nessicarely scoped. Van den Berg et al. [83] describe another class of effects called *latent effects*, which are effects that may include deferred computation such as lazy evaluation or staging. Another class of effects, as described by Xie et al. [96], are *parallel effects*, which describe effects with computations that can be run in parallel, such as for-loops. Van den Berg and Schrijvers formulated a framework for higher order effects [7], of which algebraic, scoped, latent and parallel effects are instances.

Mechanization improvements The mechanisations of Part I were done in Coq. We did not include anyone in the project that had experience with Coq mechanizations of this scale. Since this section is not the place for reflecting on that decision. However, we can at least describe the challenges of going into a project of this scale without much experience from previous mechanisations.

Firstly, even trough the locally nameless representation promises to alleviate some of the boilerplate associated with the nameless representation, we still spent a significant proportion dealing with substitution. While the Ott/LNgen stack automatically generates substitution functions for datatypes and laws about them, it does not generate substitution lemmas for judgments based on these definitions. For example, one might wish to assert that, if $\Psi \vdash_{\mathtt{ty}} \tau_1$ and $\Psi \vdash_{\mathtt{ty}} \tau_2$ then $\Psi \vdash_{\mathtt{ty}} [\tau_1/\alpha]\tau_2$. Even though these have become *properties* and not law, proving them for every judgment is cumbersome. Should some of this be automated it significantly speeds up the process.

Secondly, and perhaps more importantly, it is unclear whether the choice we made for Ott/LNgen and the locally nameless representation was the right one, because no objective resources exist that compare the two approaches. While documentation on alternatives exists (for example, the fully nameless representation with autosubst [67] or its reimplementation autosubst 2 [76]), it suffers from the "academic presentation" where separate documents showcase why their respective tool is useful, without objectively comparing the two.

Another avenue progress can be made is in the availability of books and tutorials that focus on "real-world" mechanisations. Whilst there exist excellent resources for learning how to prove theorems, most notably the Software Foundations series [55], these are unsatisfactory in two areas. Firstly, whilst they cover the *proving* of theorems, they do not give the same amount of attention to the *formulation* of definitions and theorems. This is a crucial, and perhaps most important part of the formalization process, as often if the definitions are right, the proof almost "falls out". Secondly, the existing resource suffer from the same problems that for example books about Haskell suffer from: often problems are picked which have a clean, (mostly) pure solution. This is understandable, as such problems are a much better vehicle to help understand the *language*. Yet, they do not prepare the reader for real-world scenarios, which often do not have such a clean solution.

Appendix A

Semantic Derivations

This Appendix contains semantic derivations of different handler applications that are used in the examples throughout Chapter 4.

A.1 Nondeterminism

$$
\begin{array}{ll}
 & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ c_{\mathsf{ND1}} \\
\equiv & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2) \\
\rightsquigarrow & \{\text{- E-HandOp -}\} \\
 & \quad \textbf{do}\ xs \leftarrow (\boldsymbol{\lambda} y\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{true} \\
 & \quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} y\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{false} \\
 & \qquad xs +\!\!+ ys \\
\rightsquigarrow & \{\text{- E-AppAbs -}\} \\
 & \quad \textbf{do}\ xs \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ \mathsf{true}\ \textbf{then return}\ 1\ \textbf{else return}\ 2 \\
 & \quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} y\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{false} \\
 & \qquad xs +\!\!+ ys \\
\rightsquigarrow & \{\text{- reducing } \textbf{if} \text{ -}\} \\
 & \quad \textbf{do}\ xs \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle return}\ 1 \\
 & \quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} y\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{false} \\
 & \qquad xs +\!\!+ ys \\
\rightsquigarrow & \{\text{- E-HandRet -}\} \\
 & \quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} y\,.\,\textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2)\ \mathsf{false} \\
 & \qquad [1] +\!\!+ ys \\
\rightsquigarrow & \{\text{- similar to above (the first branch of } \textbf{if}\text{) -}\} \\
 & \qquad [1] +\!\!+ [2]
\end{array}
$$

$\rightsquigarrow$ {- reducing ++ -}
return $[1, 2]$

A.2 Increment

with h_{ND} **handle** $run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}}$ $\equiv$ **with** h_{ND} **handle** $(\boldsymbol{\lambda} c\ p\,.\,\mathbf{do}\ p' \leftarrow$
with h_{inc} **handle** $p\,;\,p'\ c)\ 0\ c_{\mathsf{inc}}$
$\rightsquigarrow$ {- E-AppAbs -}
with h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle** $c_{\mathsf{inc}}\,;\,p'\ 0$
$\equiv$ {- definition of c_{inc} -}
with h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle op** `choose` $()$ $(b\,.\,$**if** b **then**
op `inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2))\,;\,p'\ 0$
$\rightsquigarrow$ {- E-FwdOp -}
with h_{ND} **handle do** $p' \leftarrow$ **op** `choose` $()$ $(b\,.\,$**with** h_{inc} **handle if** b **then**
op `inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2))\,;\,p'\ 0$
$\rightsquigarrow$ {- E-Hand and E-DoOp -}
with h_{ND} **handle op** `choose` $()$ $(b\,.\,$**do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then**
op `inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$
$\rightsquigarrow$ {- E-HandOp -}
do $xs \leftarrow (\boldsymbol{\lambda} b\,.\,$**with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$ $\mathsf{true}\,;$
do $ys \leftarrow (\boldsymbol{\lambda} b\,.\,$**with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$ $\mathsf{false}\,;\,xs +\!\!+ ys$
$\rightsquigarrow$ {- E-AppAbs -}
do $xs \leftarrow$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** true **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0$
do $ys \leftarrow (\boldsymbol{\lambda} b\,.\,$**with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$ $\mathsf{false}\,;\,xs +\!\!+ ys$
$\rightsquigarrow$ {- reducing **if** -}
do $xs \leftarrow$ **with** h_{ND} **handle** $($**do** $p' \leftarrow$ **with** h_{inc} **handle op** `inc` $()$
$(x\,.\,x+5)\,;\,p'\ 0)$
do $ys \leftarrow (\boldsymbol{\lambda} b\,.\,$**with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$ $\mathsf{false}\,;\,xs +\!\!+ ys$
$\rightsquigarrow$ {- E-HandOp -}
do $xs \leftarrow$ **with** h_{ND} **handle** $($**do** $p' \leftarrow$ **return** $(\boldsymbol{\lambda} s\,.\,$**do** $s' \leftarrow s+1\,;$**do** $k' \leftarrow$
$(\boldsymbol{\lambda} x\,.\,$**with** h_{inc} **handle** $(x+5))\ s'\,;k'\ s')\,;p'\ 0)$
do $ys \leftarrow (\boldsymbol{\lambda} b\,.\,$**with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x\,.\,x+5)$ **else op** `inc` $()$ $(y\,.\,y+2)\,;\,p'\ 0)$ $\mathsf{false}\,;\,xs +\!\!+ ys$
$\rightsquigarrow$ {- E-DoRet -}
do $xs \leftarrow$ **with** h_{ND} **handle** $(\boldsymbol{\lambda} s\,.\,$**do** $s' \leftarrow s+1\,;k' \leftarrow (\boldsymbol{\lambda} x\,.\,$**with** h_{inc}

$$
\begin{array}{l}
\quad\quad \textbf{handle}\ (x+5))s' \,;\, k'\ s') \ 0 \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,;\, xs +\!\!+ ys \\
\rightsquigarrow^* \quad \{\text{- E-AppAbs and reducing + -}\} \\
\quad \textbf{do}\ xs \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ (\textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle}\ (\textbf{return}\ 6))\ 1 \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,;\, xs +\!\!+ ys \\
\rightsquigarrow \quad \{\text{- E-HandRet -}\} \\
\quad \textbf{do}\ xs \leftarrow (\boldsymbol{\lambda} s \,.\, \textbf{return}\ (6, s))\ 1 \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,;\, xs +\!\!+ ys \\
\rightsquigarrow \quad \{\text{- E-AppAbs -}\} \\
\quad \textbf{do}\ xs \leftarrow \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle return}\ (6, 1) \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,;\, xs +\!\!+ ys \\
\rightsquigarrow \quad \{\text{- E-HandOp -}\} \\
\quad \textbf{do}\ xs \leftarrow \textbf{return}\ [(6,1)] \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,;\, xs +\!\!+ ys \\
\rightsquigarrow \quad \{\text{- E-DoRet -}\} \\
\quad \textbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{if}\ b\ \textbf{then with}\ h_{\mathsf{inc}}\ \textbf{handle op} \\
\quad\quad \mathtt{inc}\ ()\ (x \,.\, x+5)\ \textbf{else with}\ h_{\mathsf{inc}}\ \textbf{handle op}\ \mathtt{inc}\ ()\ (y \,.\, y+2) \,;\, p'\ 0)\ \mathsf{false} \,; \\
\quad\quad\quad [(6,1)] +\!\!+ ys \\
\rightsquigarrow^* \quad \{\text{- similar to above (the first branch of } \textbf{if}\text{) -}\} \\
\quad \textbf{do}\ ys \leftarrow \textbf{return}\ [(3,1)]\ [(6,1)] +\!\!+ ys \\
\rightsquigarrow^* \quad \{\text{- E-DoRet -}\} \\
\quad [(6,1)] +\!\!+ [(3,1)] \\
\rightsquigarrow^* \quad \{\text{- reducing } +\!\!+ \text{ -}\} \\
\quad \textbf{return}\ [(6,1),(3,1)]
\end{array}
$$

A.3 Once

$$
\begin{array}{l}
\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ c_{\mathsf{once}} \\
\rightsquigarrow \quad \{\text{- E-HandSc -}\} \\
\quad \textbf{do}\ ts \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \mathtt{choose}\ ()\ (x \,.\, \textbf{return}\ x))\ () \,; \\
\quad \textbf{do}\ b\ \leftarrow ts = [\,] \,; \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts \,; (\boldsymbol{\lambda} p \,.\, \textbf{with} \\
\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \mathtt{choose}\ (b \,.\, \textbf{return}\ b) \,; \textbf{return}\ (p, q)))\ t \\
\rightsquigarrow \quad \{\text{- E-Do and E-AppAbs -}\} \\
\quad \textbf{do}\ ts \leftarrow \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \mathtt{choose}\ ()\ (x \,.\, \textbf{return}\ x) \,; \\
\quad \textbf{do}\ b\ \leftarrow ts = [\,] \,; \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts \,; (\boldsymbol{\lambda} p \,.\, \textbf{with}
\end{array}
$$

h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))\ t$

$\rightsquigarrow$ {- E-Do and E-HandOp -}

do $ts \leftarrow$ **do** $xs \leftarrow (\boldsymbol{\lambda} x\,.\,$**with** h_{once} **handle return** $x)$ $\mathsf{true}\,;$
do $ys \leftarrow (\boldsymbol{\lambda} x\,.\,$**with** h_{once} **handle return** $x)$ $\mathsf{false}\,; xs +\!\!+ ys\,;$
do $b \leftarrow ts = [\,]\,;$**if** b **then return** $[\,]$ **else do** $t \leftarrow \mathsf{head}\ ts\,; (\boldsymbol{\lambda} p\,.\,$**with**
h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))\ t$

$\rightsquigarrow$ {- E-Do and E-AppAbs -}

do $ts \leftarrow$ **do** $xs \leftarrow$ **with** h_{once} **handle return** $\mathsf{true}\,;$
do $ys \leftarrow$ **with** h_{once} **handle return** $\mathsf{false}\,; xs +\!\!+ ys\,;$
do $b \leftarrow ts = [\,]\,;$**if** b **then return** $[\,]$ **else do** $t \leftarrow \mathsf{head}\ ts\,; (\boldsymbol{\lambda} p\,.\,$**with**
h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))\ t$

$\rightsquigarrow$ {- E-Do and E-HandRet -}

do $ts \leftarrow$ **do** $xs \leftarrow$ **return** $[\mathsf{true}]\,;$**do** $ys \leftarrow$ **return** $[\mathsf{false}]\,; xs +\!\!+ ys\,;$
do $b \leftarrow ts = [\,]\,;$**if** b **then return** $[\,]$ **else do** $t \leftarrow \mathsf{head}\ ts\,; (\boldsymbol{\lambda} p\,.\,$**with**
h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))\ t$

$\rightsquigarrow^*$ {- E-DoRet -}

do $ts \leftarrow [\mathsf{true}, \mathsf{false}]\,;$
do $b \leftarrow ts = [\,]\,;$**if** b **then return** $[\,]$ **else do** $t \leftarrow \mathsf{head}\ ts\,; (\boldsymbol{\lambda} p\,.\,$**with**
h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))\ t$

$\rightsquigarrow^*$ {- E-DoRet -}

$(\boldsymbol{\lambda} p\,.\,$**with** h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(p, q)))$
true

$\rightsquigarrow$ {- E-AppAbs -}

with h_{once} **handle** (**do** $q \leftarrow$ **op** `choose` $(b\,.\,$**return** $b)\,;$**return** $(\mathsf{true}, q))$

$\rightsquigarrow^*$ {- similar to A.1 (handling of `choose`) -}

return $[(\mathsf{true}, \mathsf{true}), (\mathsf{true}, \mathsf{false})]$

Appendix B

Type Equivalence Rules

This appendix shows the type equivalence rules of λ_{sc}. Figures B.1 and B.2 contains the rules. Rules Q-AppAbs and Q-Swap deserve special attention. The other rules are straightforward.

$\boxed{\sigma_1 \equiv \sigma_2}$ Type equivalence

$$\frac{}{\sigma \equiv \sigma}\ \text{Q-Refl} \qquad \frac{\sigma_1 \equiv \sigma_2}{\sigma_2 \equiv \sigma_1}\ \text{Q-Symm}$$

$$\frac{\sigma_1 \equiv \sigma_2 \qquad \sigma_2 \equiv \sigma_3}{\sigma_1 \equiv \sigma_3}\ \text{Q-Trans} \qquad \frac{A_1 \equiv A_2 \qquad B_1 \equiv B_2}{(A_1, B_1) \equiv (A_2, B_2)}\ \text{Q-Pair}$$

$$\frac{A \equiv B \qquad \underline{C} \equiv \underline{D}}{A \to \underline{C} \equiv B \to \underline{D}}\ \text{Q-Fun} \qquad \frac{\underline{C_1} \equiv \underline{D_1} \qquad \underline{C_2} \equiv \underline{D_2}}{\underline{C_1} \Rightarrow \underline{C_2} \equiv \underline{D_1} \Rightarrow \underline{D_2}}\ \text{Q-Hand}$$

$$\frac{\sigma_1 \equiv \sigma_2}{\forall\, \alpha\,.\,\sigma_1 \equiv \forall\, \alpha\,.\,\sigma_2}\ \text{Q-AllTy} \qquad \frac{\sigma_1 \equiv \sigma_2}{\forall\, \mu\,.\,\sigma_1 \equiv \forall\, \mu\,.\,\sigma_2}\ \text{Q-AllRow}$$

$$\frac{A \equiv B}{\lambda\, \alpha\,.\, A \equiv \lambda\, \alpha\,.\, B}\ \text{Q-Abs} \qquad \frac{M_1 \equiv M_2 \qquad A \equiv B}{M_1\ A \equiv M_2\ B}\ \text{Q-App}$$

$$\frac{}{(\lambda\, \alpha\,.\, A)\ B \equiv A\,[B\,/\,\alpha]}\ \text{Q-AppAbs} \qquad \frac{A \equiv B \qquad E \equiv_{\langle\rangle} F}{A\,!\,\langle E\rangle \equiv B\,!\,\langle F\rangle}\ \text{Q-Comp}$$

Figure B.1: Type equivalence of λ_{sc}.

$\boxed{E \equiv_{\langle\rangle} F}$ Row equivalence

$$\frac{}{E \equiv_{\langle\rangle} E}\ \text{R-Refl} \qquad \frac{E \equiv_{\langle\rangle} F}{F \equiv_{\langle\rangle} E}\ \text{R-Symm}$$

$$\frac{E_1 \equiv_{\langle\rangle} E_2 \qquad E_2 \equiv_{\langle\rangle} E_3}{E_1 \equiv_{\langle\rangle} E_3}\ \text{R-Trans} \qquad \frac{E \equiv_{\langle\rangle} F}{\ell ; E \equiv_{\langle\rangle} \ell ; F}\ \text{R-Head}$$

$$\frac{\ell_1 \neq \ell_2}{\ell_1 ; \ell_2 ; E \equiv_{\langle\rangle} \ell_2 ; \ell_1 ; E}\ \text{R-Swap}$$

Figure B.2: Row equivalence of λ_{sc}.

Appendix C

Well-scopedness Rules

This appendix shows the well-scopedness rules of λ_{sc}. Figure C.1 contains the rules.

$\boxed{\Gamma \vdash \sigma}$ $\boxed{\Gamma \vdash M}$ $\boxed{\Gamma \vdash E}$ $\boxed{\Gamma \vdash \underline{C}}$ Type well-scopedness

$$\frac{}{\Gamma \vdash ()}\ \text{W-Unit} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A, B)}\ \text{W-Pair} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{W-Var}$$

$$\frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall\, \alpha\,.\, A}\ \text{W-All} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash E}{\Gamma \vdash A\,!\,\langle E \rangle}\ \text{W-Comp} \qquad \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \lambda\, \alpha\,.\, A}\ \text{W-Abs}$$

$$\frac{\Gamma \vdash M \qquad \Gamma \vdash A}{\Gamma \vdash M\ A}\ \text{W-App} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash \underline{C}}{\Gamma \vdash A \to \underline{C}}\ \text{W-Fun}$$

$$\frac{\Gamma \vdash \underline{C} \qquad \Gamma \vdash \underline{D}}{\Gamma \vdash \underline{C} \Rightarrow \underline{D}}\ \text{W-Hand} \qquad \frac{\mu \in \Gamma}{\Gamma \vdash \mu}\ \text{W-RowVar}$$

$$\frac{}{\Gamma \vdash \cdot}\ \text{W-EmptyRow} \qquad \frac{\Gamma \vdash E}{\Gamma \vdash \ell\,;\, E}\ \text{W-Extension}$$

Figure C.1: Well-scopedness rules of λ_{sc}.

Appendix D

Syntax-directed version of λ_{sc}

This section describes the syntax-direction version of λ_{sc}.

The syntax-directed rules can be found in Figure D.1 for value typing, Figure D.2 for computation and Figure D.3 for handler typing.

$\boxed{\Gamma \vdash_{\mathsf{SD}} v : A}$ Value Typing

$$\frac{(x : \sigma) \in \Gamma \qquad \sigma \leqslant A \qquad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} x : A}\ \text{SD-Var} \qquad \frac{}{\Gamma \vdash_{\mathsf{SD}} () : ()}\ \text{SD-Unit}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v_1 : A \qquad \Gamma \vdash_{\mathsf{SD}} v_2 : B}{\Gamma \vdash_{\mathsf{SD}} (v_1, v_2) : (A, B)}\ \text{SD-Pair} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} x \,.\, c : A \to \underline{C}}\ \text{SD-Abs}$$

SD-Handler

$$\frac{\begin{array}{c} F \equiv_{\langle\rangle} \mathit{labels}(\mathit{oprs}) ; E \qquad \alpha \notin \Gamma \qquad \Gamma, \alpha \vdash_M \mathbf{return}\ x \mapsto c_r : M\ \alpha!\langle E \rangle \\ \Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle E \rangle \qquad \Gamma, \alpha \vdash_M \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha!\langle E \rangle \qquad \Gamma \vdash_{\mathsf{SD}} A \end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{handler}_M\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{fwd}\ f\ p\ k \mapsto c_f\} : A!\langle F \rangle \Rightarrow M\ A!\langle E \rangle}$$

Figure D.1: Syntax-directed value typing.

The syntax-directed system is obtained by incorporating the non-syntax-directed rules into the syntax-directed-ones where needed. In particular, we inline the non-syntax-directed rules for equivalence (T-EqV and T-EqC) into the syntax-directed rules that mention the same type or row twice in their assumptions (e.g., SD-App, SD-Do). Similarly, we inline the rules T-Inst, T-InstEff, T-Gen

$\boxed{\Gamma \vdash_{\mathsf{SD}} c : \underline{C}}$ Computation Typing

$$\frac{\Gamma \vdash_{\mathsf{SD}} v_1 : A_1 \to \underline{C} \qquad \Gamma \vdash_{\mathsf{SD}} v_2 : A_2 \qquad A_1 \equiv A_2}{\Gamma \vdash_{\mathsf{SD}} v_1\ v_2 : \underline{C}}\ \text{SD-App}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} c_1 : A!\langle E_1\rangle \qquad \Gamma, x : A \vdash_{\mathsf{SD}} c_2 : B!\langle E_2\rangle \qquad E_1 \equiv E_2}{\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\,;\, c_2 : B!\langle E_2\rangle}\ \text{SD-Do}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A \\ (\overline{\alpha} \notin \Gamma) \qquad (\overline{\mu} \notin \Gamma) \qquad \Gamma, x : \overline{\forall\, \alpha}\,.\,\overline{\forall\, \mu}\,.\, A \vdash_{\mathsf{SD}} c : \underline{C}\end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}}\ \text{SD-Let}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v : A}{\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : A!\langle E\rangle}\ \text{SD-Ret}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v : \underline{C}_1 \Rightarrow \underline{D}_1 \qquad \Gamma \vdash_{\mathsf{SD}} c : \underline{C}_2 \qquad \underline{C}_1 \equiv \underline{C}_2 \qquad \underline{D}_1 \equiv \underline{D}_2}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ v\ \mathbf{handle}\ c : \underline{D}_2}\ \text{SD-Hand}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \in \Sigma \\ \Gamma \vdash_{\mathsf{SD}} v : A_1 \qquad A_{\mathsf{op}} \equiv A_1 \qquad \Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c : A!\langle E\rangle \qquad \ell^{\mathsf{op}} \in E\end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c) : A!\langle E\rangle}\ \text{SD-Op}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}) \in \Sigma \\ \Gamma \vdash_{\mathsf{SD}} v : A_1 \qquad A_{\mathsf{sc}} \equiv A_1 \qquad \Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : B!\langle E_1\rangle \\ \Gamma, z : B \vdash_{\mathsf{SD}} c_2 : A!\langle E_2\rangle \qquad E_1 \equiv E_2 \qquad \ell^{\mathsf{sc}} \in E_2\end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A!\langle E_2\rangle}\ \text{SD-Sc}$$

Figure D.2: Syntax-directed computation typing.

and T-GenEff for instantiating and generalizing type and row variables. The generalization is incorporated into the rule for let-bindings (T-Let). Instantiation is incorporated into the variable rule (T-Var) using $\sigma \leqslant A$ defined in Figure D.4.

Instantiation is also incorporated into the handler rule: we implicitly instantiate α with an arbitrary type A, which results in a monomorphically typed handler. However, since SD-Handler insists on sufficiency polymorphic handler clauses, we can still handle scoped effects by polymorphic recursion.

Figure D.5 displays declarative and syntax-directed typing derivations for both inline handler application (**with** h **handle** c) as well as let-bound handlers. As can be seen in the first derivation, in the case of inline handler application, the

$\boxed{\Gamma \vdash_M \ \mathbf{return}\ x \mapsto c_r : M\ A\,!\langle E\rangle}$ $\boxed{\Gamma \vdash_M \ oprs : M\ A\,!\langle E\rangle}$
$\boxed{\Gamma \vdash_M \ \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A\,!\langle E\rangle}$

Return-, operation-, and forwarding-clause typing

$$\frac{\Gamma, x : A_1 \vdash_{\mathsf{SD}} c_r : M\ A_2\,!\langle E\rangle \qquad A_1 \ \equiv\ A_2}{\Gamma \vdash_M \ \mathbf{return}\ x \mapsto c_r : M\ A\,!\langle E_2\rangle}\ \text{SD-Return}$$

$$\frac{}{\Gamma \vdash_M \ \cdot : M\ A\,!\langle E\rangle}\ \text{SD-Empty}$$

$$\frac{\begin{array}{c}\Gamma \vdash_M \ oprs : M\ A_1\,!\langle E_1\rangle \qquad (\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \ \in\ \Sigma \\ \Gamma, x : A_{\mathsf{op}}, k : B_{\mathsf{op}} \to M\ A_1\,!\langle E_1\rangle \vdash_{\mathsf{SD}} c : M\ A_2\,!\langle E_2\rangle \\ M\ A_1\,!\langle E_1\rangle \ \equiv\ M\ A_2\,!\langle E_2\rangle\end{array}}{\Gamma \vdash_M \ \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, oprs : M\ A_2\,!\langle E_2\rangle}\ \text{SD-OprOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash_M \ oprs : M\ A_1\,!\langle E_1\rangle \qquad (\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}) \ \in\ \Sigma \qquad \beta \notin \Gamma \\ \Gamma, \beta, x : A_{\mathsf{sc}}, p : B_{\mathsf{sc}} \to M\ \beta\,!\langle E_1\rangle, k : \beta \to M\ A_1\,!\langle E_1\rangle \vdash_{\mathsf{SD}} c : M\ A_2\,!\langle E_2\rangle \\ M\ A_1\,!\langle E_1\rangle \ \equiv\ M\ A_2\,!\langle E_2\rangle\end{array}}{\Gamma \vdash_M \ \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, oprs : M\ A_2\,!\langle E_2\rangle}\ \text{SD-OprSc}$$

$$\frac{\begin{array}{c}\alpha, \beta, \gamma, \delta \notin \Gamma \qquad A_p = \alpha \to M\ \beta\,!\langle E_1\rangle \\ A'_p = \alpha \to \gamma\,!\langle E_1\rangle \qquad A_k = \beta \to M\ A_1\,!\langle E_1\rangle \qquad A'_k = \gamma \to \delta\,!\langle E_1\rangle \\ \Gamma, \alpha, \beta, p : A_p, k : A_k, f : \forall\ \gamma\ \delta\,.\,(A'_p, A'_k) \to \delta\,!\langle E_1\rangle \vdash_{\mathsf{SD}} c_f : M\ A_2\,!\langle E_2\rangle \\ M\ A_1\,!\langle E_1\rangle \ \equiv\ M\ A_2\,!\langle E_2\rangle \qquad \Gamma \vdash_{\mathsf{SD}} A_2\end{array}}{\Gamma \vdash_M \ \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A_2\,!\langle E_2\rangle}\ \text{SD-Fwd}$$

Figure D.3: Syntax-directed handler typing.

$\boxed{\sigma \leqslant A}$ σ-instantiation

$$\frac{}{A \leqslant A}\ \sigma\text{-Inst-Base} \qquad \frac{[B\ /\ \alpha]\sigma \leqslant A}{\forall\ \alpha\,.\,\sigma \leqslant A}\ \sigma\text{-Inst-}\alpha \qquad \frac{[E\ /\ \mu]\ \sigma \leqslant A}{\forall\ \mu\,.\,\sigma \leqslant A}\ \sigma\text{-Inst-}\mu$$

Figure D.4: σ-instantiation.

declarative system derives a polymorphically typed handler, which is instantiated. The syntax-directed system essentially combines these steps, as can be seen in the second derivation. In the case of a let-bound handler, the declarative system keeps the polymorphic handler type as-is (third derivation). The syntax-directed system however instantiates and then immediately generalizes handlers, as can be seen in the fourth derivation.

The other rules of the declarative system are syntax-directed and remain unchanged.

$$\dfrac{\dfrac{\dfrac{\Gamma, \alpha \vdash_{\mathsf{SD}} \mathit{oprs} : M\ \alpha\,!\langle E\rangle}{\Gamma \vdash_{\mathsf{SD}} h : \forall\, \alpha\,.\, \alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle}\ 3 \quad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} h : A\,!\langle F\rangle \Rightarrow M\ A\,!\langle E\rangle}\ 2 \quad \Gamma \vdash_{\mathsf{SD}} c : A\,!\langle F\rangle}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle}\ c : M\ A\,!\langle E\rangle}\ 1$$

$$\dfrac{\dfrac{\Gamma, \alpha \vdash_{M} \mathit{oprs} : M\ \alpha\,!\langle E\rangle \qquad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} h_M : A\,!\langle F\rangle \Rightarrow M\ A\,!\langle E\rangle}\ 5 \qquad \Gamma \vdash_{\mathsf{SD}} c : A\,!\langle F\rangle}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h_M\ \mathbf{handle}\ c : M\ A\,!\langle E\rangle}\ 4$$

$$\dfrac{\dfrac{\Gamma, \alpha \vdash_{\mathsf{SD}} \mathit{oprs} : M\ \alpha\,!\langle E\rangle}{\Gamma \vdash_{\mathsf{SD}} h : \forall\, \alpha\,.\, \alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle}\ 3 \quad \Gamma, x : \forall\, \alpha\,.\, \alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = h\ \mathbf{in}\ c : \underline{C}}\ 6$$

$$\dfrac{\dfrac{\Gamma, \alpha, \beta \vdash_{M} \mathit{oprs} : M\ \beta\,!\langle E\rangle \quad \Gamma, \alpha \vdash_{\mathsf{SD}} \alpha}{\Gamma \vdash_{\mathsf{SD}} h_M : \alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle}\ 5 \quad \Gamma, x : \forall\, \alpha\,.\, \alpha\,!\langle F\rangle \Rightarrow M\ \alpha\,!\langle E\rangle \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = h_M\ \mathbf{in}\ c : \underline{C}}\ 7$$

1 = T-Hand
2 = T-Inst
3 = T-Handler
4 = SD-Hand
5 = SD-Handler
6 = T-Let
7 = SD-Let

Figure D.5: Handler generalisation and instantiation

Appendix E

Metatheory

E.1 Lemmas

Lemma E.1.1 (Canonical forms)**.**

- *If* $\cdot \vdash_{\mathsf{SD}} v : A \to \underline{C}$ *then* v *is of shape* $\boldsymbol{\lambda} x \,.\, c$*.*
- *If* $\cdot \vdash_{\mathsf{SD}} v : \underline{C} \Rightarrow \underline{D}$ *then* v *is of shape* h*.*

Lemma E.1.2 (Generalisation-equivalence)**.** *If* $\sigma_1 \leqslant A_1$ *and* $\sigma_1 \equiv \sigma_2$*, then there exists a* A_2 *such that* $A_1 \equiv A_2$ *and* $\sigma_2 \leqslant A_2$*.*

Lemma E.1.3 (Generalisation-instantiation)**.** *If* $\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A$ *and* $\overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A \leqslant B$*, then* $\Gamma \vdash_{\mathsf{SD}} v : B$*.*

Lemma E.1.4 (Preservation of types under term substitution)**.** *Given* $\Gamma_1, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A_1$ *and* $A_1 \equiv A_2$ *we have that:*

- *If* $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}_1$*, then there exists a* $\underline{C}_2$ *such that* $\underline{C}_1 \equiv \underline{C}_2$ *and* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, c : \underline{C}_2$*.*
- *If* $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} v : B_1$*, then there exists a* B_2 *such that* $B_1 \equiv B_2$ *and* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, v : B_2$*.*

Proof. By mutual induction on the typing derivations. The only interesting case, SD-Var, requires us to show that, given $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} y : B_1$, there exists a B_2 such that $B_1 \equiv B_2$ and $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, y : B_2$. If $x \neq y$, it is trivial. If $x = y$, then $\overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2 \leqslant B_1$, which means by Lemma E.1.2 there

exists a B_2 such that $B_1 \equiv B_2$ and $\overline{\forall\, \alpha}\, \overline{\forall\, \mu}\,.\, A_1 \leqslant B_2$, which means the result follows from Lemma E.1.3. □

Lemma E.1.5 (Preservation of types under type substitution). *If* $\Gamma_1, \alpha, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $\Gamma_1 \vdash_{\mathsf{SD}} B$, *then* $\Gamma_1, [B \,/\, \alpha]\, \Gamma_2 \vdash_{\mathsf{SD}} c : [B \,/\, \alpha]\, \underline{C}$.

Lemma E.1.6 (Unused binding insertion). *If* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $x \notin c$ *then* $\Gamma_1, x : A, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$.

Lemma E.1.7 (Handlers are polymorphic). *If* $\Gamma \vdash_{\mathsf{SD}} h : A\,!\,\langle F\rangle \Rightarrow M\ A\,!\,\langle E\rangle$ *and* $\Gamma \vdash_{\mathsf{SD}} B$, *then* $\Gamma \vdash_{\mathsf{SD}} h : B\,!\,\langle F\rangle \Rightarrow M\ B\,!\,\langle E\rangle$.

Lemma E.1.8 (Op membership). *If* $\Gamma \vdash_{\mathsf{SD}} oprs : \underline{C}$ *and* $\mathbf{op}\ \ell^{op}\ x\ k \mapsto c \in oprs$, *then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1, \mathbf{op}\ \ell^{op}\ k \vdash_{\mathsf{SD}} c, oprs_2$ *and* $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{op}\ x\ k \mapsto c, oprs_2 : \underline{C}$.

Lemma E.1.9 (Sc membership). *If* $\Gamma \vdash_{\mathsf{SD}} oprs : \underline{C}$ *and* $(\mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c) \in h$, *then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1, \mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c, oprs_2$ *and* $\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c, oprs_2 : \underline{C}$.

Lemma E.1.10 (Syntax-directed to Non-syntax-directed value typing). *If* $\Gamma \vdash v : \overline{\forall\, \alpha}\,.\, \overline{\forall\, \mu}\,.\, A$ *then* $\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A$.

Lemma E.1.11 (Non-syntax-directed iff Syntax-directed). $\Gamma \vdash c : \underline{C} \iff \Gamma \vdash_{\mathsf{SD}} c : \underline{C}$.

E.2 Subject reduction

Theorem 4.8.1 (Subject Reduction). *If* $\Gamma \vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$, *then there exists a* $\underline{C}'$ *such that* $\underline{C} \equiv \underline{C}'$ *and* $\Gamma \vdash c' : \underline{C}'$.

Proof. By Lemma E.1.11 (above) and Theorem E.2.1 (below). □

Theorem E.2.1 (Syntax-directed Subject Reduction). *If* $\Gamma \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $c \rightsquigarrow c'$, *then there exists a* $\underline{C}'$ *such that* $\underline{C} \equiv \underline{C}'$ *and* $\Gamma \vdash_{\mathsf{SD}} c' : \underline{C}'$.

Proof. Assume, without loss of generality, that $\underline{C} = B\,!\,\langle F\rangle$ for some B, F. Proceed by induction on the derivation $c \rightsquigarrow c'$.

- E-AppAbs: Inversion on $\Gamma \vdash_{\mathsf{SD}} (\boldsymbol{\lambda} x\,.\, c)\ v : B\,!\,\langle F\rangle$ (SD-App) gives $\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} x\,.\, c : A_1 \to B\,!\,\langle F\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} v : A_2$ (2), and $A_1 \equiv A_2$ (3). Inversion on fact 1 (SD-Abs) gives $\Gamma, x : A_1 \vdash_{\mathsf{SD}} c : B\,!\,\langle F\rangle$ (4), which means the goal follows from facts 2 and 4 and Lemma E.1.4.

- E-Let: Inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : B!\langle F\rangle$ (SD-Let) gives $\Gamma \vdash_{\mathsf{SD}} v : A$ (1), $\sigma = gen\ (A, \Gamma)$ (2), and $\Gamma, x : \sigma \vdash_{\mathsf{SD}} c : B!\langle F\rangle$ (3), which means the goal follows from facts 1 and 3 and Lemma E.1.4.

- E-Do: Follows from the IH.

- E-DoRet: Inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{return}\ v\ \mathbf{in}\ c : B!\langle F_2\rangle$ (SD-Do) gives $\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : A!\langle F_1\rangle$ (1) and $\Gamma, x : A \vdash_{\mathsf{SD}} c : B!\langle F_2\rangle$ (2). Inversion on (1) (SD-Ret) gives $\Gamma \vdash_{\mathsf{SD}} v : A$ (3). The case follows from facts 2 and 4 and Lemma E.1.4.

- E-DoOp: Similar to E-DoSc. By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1)\ \mathbf{in}\ c_2 : B!\langle F_2\rangle$ (SD-Do) we have that $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : A!\langle F_1\rangle$ (1), $\Gamma, x : A \vdash_{\mathsf{SD}} c_2 : B!\langle F_2\rangle$ (2), and $F_1 \equiv F_2$ (3). From inversion on fact 1 (SD-Op) it follows that $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (4), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (5), $A_{\mathsf{op}} \equiv A_1$ (6), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A!\langle F_1\rangle$ (7), and $\ell^{\mathsf{op}} \in F_1$ (8). Lemma E.1.6 on (2) gives us $\Gamma, y : B_{\mathsf{op}}, x : A \vdash_{\mathsf{SD}} c_2 : B!\langle F_2\rangle$ (9). Facts 3, 7 and 9 and rule SD-Do give us $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : B!\langle F_2\rangle$ (10). Our goal then follows from facts 4, 5, 6, 8, and 10 and rule SD-Op.

- E-DoSc: Similar to E-DoOp. By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\ \mathbf{in}\ c_3 : B!\langle F_3\rangle$ (SD-Do) we have that $\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A!\langle F_2\rangle$ (1), $\Gamma, x : A \vdash_{\mathsf{SD}} c_3 : B!\langle F_3\rangle$ (2), and $F_2 \equiv F_3$ (2.1). From inversion on fact 1 (SD-Sc) it follows that $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (3), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (4), $A_{\mathsf{sc}} \equiv A_1$ (5), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : B'!\langle F_1\rangle$ (6), $\Gamma, z : B' \vdash_{\mathsf{SD}} c_2 : A!\langle F_2\rangle$ (7), $F_1 \equiv F_2$ (8), and $\ell^{\mathsf{sc}} \in F_2$ (9). Lemma E.1.6 on (2) gives us $\Gamma, z : B', x : A \vdash_{\mathsf{SD}} c_3 : B!\langle F_3\rangle$ (10), which means facts 2.1, 7 and 10 and rule SD-Do give us $\Gamma, z : B' \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_2\ \mathbf{in}\ c_3 : B!\langle F_3\rangle$ (11). Our goal then follows from facts 3, 4, 5, 6, 8 9, and 11 and rule SD-Sc.

- E-Hand: Follows from the IH.

- E-HandRet: By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ return}\ v : B!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E\rangle$, and $\Gamma, \alpha \vdash_M \mathbf{return}\ x \mapsto c_r : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 4 (SD-Return) gives $\Gamma, \alpha, x : A_1 \vdash_{\mathsf{SD}} c_r : M\ \alpha!\langle F_2\rangle$ (5) and $A_1 \equiv A_2$ (6). Inversion on fact 2 (SD-Ret) gives $\Gamma \vdash_{\mathsf{SD}} v : A2'$ (7). From facts 4-8 and Lemma E.1.4, we get that $\Gamma, \alpha \vdash_{\mathsf{SD}} [v/x]\ c_r : M\ \alpha!\langle F_2\rangle$ (8). We obtain our goal from fact 8 and Lemma E.1.5.

- E-HandOp: By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}}$

op $\ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-HANDLER) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E\rangle$, and $\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 2 (SD-OP) gives us $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (6), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (7), $A_{\mathsf{op}} \equiv A_1$ (8), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A2'!\langle E'\rangle$ (9), and $\ell^{\mathsf{op}} \in E'$ (10). By Lemma E.1.8 we get that $\Gamma, \alpha \vdash_M$ **op** $\ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs}_2 : M\ \alpha!\langle F_2\rangle$ (11). Inversion on fact 11 (SD-OPROP) gives that $\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_1\rangle$ (12), $(\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \in \Sigma$ (13), $\Gamma, \alpha, x : A_{\mathsf{op}}, k : B_{\mathsf{op}} \to M\ \alpha!\langle F_1\rangle \vdash_{\mathsf{SD}} c : M\ \alpha!\langle F_2\rangle$ (14), and $F_1 \equiv F_2$ (15). Facts 1, 4 and 9 in combination with constructors SD-ABS and ST-HAND gives us that $\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} y\,.$ **with** h **handle** $c_1 : B_{\mathsf{op}} \to M\ A_2!\langle F_2\rangle$ (16). The goal follows from facts 7, 8, 14 and 16 and lemmas Lemmas E.1.4 and E.1.5.

- E-FWDOP By inversion on $\Gamma \vdash_{\mathsf{SD}}$ **with** h **handle op** $\ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-HAND) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}}$ **op** $\ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-HANDLER) gives $B = M\ A_2$, and $\underline{C}_1 = A_2!\langle E\rangle$. Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 2 (SD-OP) gives us $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (6), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (7), $A_{\mathsf{op}} \equiv A_1$ (8), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A2'!\langle E'\rangle$ (9), and $\ell^{\mathsf{op}} \in E'$ (10). The goal follows from facts 1, 4, 5, 6, 7, 8, 10, constructors SD-HAND and SD-OP, and Lemma E.1.6.

- E-HANDSC: By inversion on $\Gamma \vdash_{\mathsf{SD}}$ **with** h **handle op** $\ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-HAND) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-HANDLER) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E_1\rangle$, and $\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E_2\rangle$, $A_2 \equiv A2'$ (5), and $E_1 \equiv E_2$ (6). Inversion on fact 2 (SD-SC) gives us $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (7), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (8), $A_{\mathsf{sc}} \equiv A_1$ (9), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : A_3!\langle E_3\rangle$ (10), $\Gamma, z : A_3 \vdash_{\mathsf{SD}} c_2 : A2'!\langle E_2\rangle$ (11), $E_3 \equiv E_2$ (12), and $\ell^{\mathsf{sc}} \in E_2$ (13). Lemma E.1.8 we get that $\Gamma, \alpha \vdash_M$ **op** $\ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs}_2 : M\ \alpha!\langle F_2\rangle$ (13.1). Inversion on fact 13.1 (SD-OPRSC gives $\ell^{\mathsf{sc}} \in \Sigma$ (14), β *fresh* (15), $\Gamma, \alpha, \beta, x : A_{\mathsf{sc}}, p : B_{\mathsf{sc}} \to M\ \beta!\langle F_3\rangle, k : \beta \to M\ \alpha!\langle F_3\rangle \vdash_{\mathsf{SD}} c : M\ \alpha!\langle F_2\rangle$ (16), and $F_2 \equiv F_3$ (17). Facts 1, 5, 6, 10 and 12, constructors SD-ABS and SD-HAND and Lemmas E.1.6 and E.1.7 give us that $\Gamma, \beta \vdash_{\mathsf{SD}}$ $\boldsymbol{\lambda} y\,.$ **with** h **handle** $c_1 : p : B_{\mathsf{sc}} \to M\ \beta!\langle F_2\rangle$ (18). Facts 1, 5, 6 and 11 and constructors SD-ABS and SD-HAND and Lemma E.1.6 give us that $\Gamma, \beta \vdash_{\mathsf{SD}} \boldsymbol{\lambda} z\,.$ **with** h **handle** $c_2 : \beta \to M\ A_2!\langle F_2\rangle$ (19). The goal now follows from facts 8, 9, 16, 17 and 18 and Lemma E.1.4.

- E-FWDSC By inversion on $\Gamma \vdash_{\mathsf{SD}}$ **with** h **handle op** $\ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-HAND) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}}$

$\ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E_1\rangle$, and $\Gamma, \alpha \vdash_M \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E_2\rangle$, $A_2 \equiv A2'$ (5), and $E_1 \equiv E_2$ (6). Inversion on fact 2 (SD-Sc) gives us $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (7), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (8), $A_{\mathsf{sc}} \equiv A_1$ (9), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : A_3!\langle E_3\rangle$ (10), $\Gamma, z : A_3 \vdash_{\mathsf{SD}} c_2 : A2'!\langle E_2\rangle$ (11), $E_3 \equiv E_2$ (12), and $\ell^{\mathsf{sc}} \in E_2$ (13). Inversion on fact 4 (SD-Fwd) gives $A_p = \alpha' \to M\ \beta!\langle F_1\rangle$, $A'_p = \alpha' \to \gamma!\langle F_1\rangle$, $A_k = \beta \to M\ A_4!\langle F_1\rangle$, $A'_k = \gamma \to \delta!\langle F_1\rangle$, $\Gamma, \alpha, \alpha', \beta, p : A_p, k : A_k, f : \forall\ \gamma\ \delta\,.\,(A'_p, A'_k) \to \delta!\langle F_1\rangle \vdash_{\mathsf{SD}} c_f : M\ \alpha!\langle F_2\rangle$ (14) and $M\ A_1!\langle F_1\rangle \equiv M\ \alpha!\langle F_2\rangle$ (15). Facts 1, 6, 10 and 12, constructors SD-Abs and SD-Hand and Lemmas E.1.6 and E.1.7 give us that $\Gamma, \alpha, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle}\ c_1 : M\ A_3!\langle F_2\rangle$ (16) Facts 1, 6, 10 and 12, constructors SD-Abs and SD-Hand and Lemma E.1.6 give us that $\Gamma, \alpha \vdash_{\mathsf{SD}} \boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2 : A_3 \to M\ A_4!\langle F_2\rangle$ (17) Facts 7, 8, 9, 13, the fact that $\ell^{\mathsf{sc}} \notin \mathit{labels}\,(\mathit{oprs})$, constructors SD-Abs, SD-App and SD-Var and Lemma E.1.6 give us that $\Gamma, \alpha, (p', k') : \forall\ \gamma\ \delta\,.\,(B_{\mathsf{sc}} \to \gamma!\langle F_1\rangle, \gamma \to \delta!\langle F_1\rangle) \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z) : \delta!\langle F_1\rangle$ (18). Our goal then follows from facts 14, 15, 16, 17, 18 and Lemma E.1.4.

□

E.3 Progress

Theorem 4.8.2 (Progress)**.** *If* $\cdot \vdash c : A!\langle E\rangle$*, then either:*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is in a* normal form*, which means it is in one of the following forms: (1)* $c = \mathbf{return}\ v$*, (2)* $c = \mathbf{op}\ \ell^{\mathit{op}}\ v\ (y\,.\,c')$ *where* $\ell^{\mathit{op}} \in E$*, or (3)* $c = \mathbf{sc}\ \ell^{\mathit{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ *where* $\ell^{\mathit{sc}} \in E$*.*

Proof. By Lemma E.1.11 (above) and Theorem E.3.1 (below). □

Theorem E.3.1 (Syntax-directed Progress)**.** *If* $\cdot \vdash_{\mathsf{SD}} c : A!\langle E\rangle$*, then either:*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is in a* normal form*, which means it is in one of the following forms: (1)* $c = \mathbf{return}\ v$*, (2)* $c = \mathbf{op}\ \ell^{\mathit{op}}\ v\ (y\,.\,c')$ *where* $\ell^{\mathit{op}} \in E$*, or (3)* $c = \mathbf{sc}\ \ell^{\mathit{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ *where* $\ell^{\mathit{sc}} \in E$*.*

Proof. By induction on the typing derivation $\cdot \vdash_{\mathsf{SD}} c : \underline{C}$.

- SD-App: Here, $\cdot \vdash_{\mathsf{SD}} v_1\ v_2$. Since v_1 has type $A \to B!\langle F\rangle$, by Lemma E.1.1 it must be of shape $\boldsymbol{\lambda} x\,.\,c$, which means we can step by rule E-AppAbs.
- SD-Do: Here, $\cdot \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : \underline{C}$. By the induction hypothesis, c_1 can either step (in which case we can step by E-Do), or it is a computation result. Every possible form has a corresponding reduction: if $c_1 = \mathbf{return}\ v$ we can step by E-DoRet, if $c_1 = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c)$ we can step by E-DoOp, and if $\mathbf{sc}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1')\ (z\,.\,c_2')$ we can step by E-DoSc.
- SD-Let: Here, $\cdot \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}$, which means we can step by E-Let.
- SD-Ret, SD-Op, and SD-Sc: all of these are computation results (forms (1), (2), and (3), resp.). Since they are well-typed, the scoping condition of ℓ^{op} or ℓ^{sc} must be satisfied.
- SD-Hand: Here $\cdot \vdash_{\mathsf{SD}} \mathbf{with}\ v\ \mathbf{handle}\ c : M\ A!\langle F\rangle$. By Lemma E.1.1, v is of shape h. By the induction hypothesis, c can either step (in which case we can step by E-Hand), or it is in a normal form. Proceed by case split on the three forms.
  1. Case $c = \mathbf{return}\ v$. Since $\cdot \vdash_{\mathsf{SD}} h : \underline{C} \Rightarrow \underline{D}$, there must be some $(\mathbf{return}\ x \mapsto c_r) \in h$ which means we can step by rule E-HandRet.
  2. Case $c = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c')$. Depending on $(\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c) \in h$ we can step by E-HandOp or E-FwdOp.
  3. Case $c = \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$. If $(\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c) \in h$, we can step by E-HandSc. If not, since $\cdot \vdash_{\mathsf{SD}} h : \underline{C} \Rightarrow \underline{D}$, there must be some $(\mathbf{fwd}\ f\ p\ k \mapsto c_f) \in h$ which means we can step by rule E-FwdSc.

□

Bibliography

[1] Ocaml – language extensions. https://ocaml.org/manual/5.2/effects.html.

[2] Amal Ahmed. Oregon Programming Languages Summer School — Logical Relations Course. https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html.

[3] Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2010.

[4] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008. doi:10.1145/1328438.1328443.

[5] Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013. doi:10.1007/978-3-642-40206-7_1.

[6] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011. URL: https://www.sciencedirect.com/science/article/pii/S2352220814000194, doi:https://doi.org/10.1016/j.jlamp.2014.02.001.

[7] Birthe van den Berg and Tom Schrijvers. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086, 2024. URL: https://www.sciencedirect.com/science/article/pii/S0167642324000091, doi:https://doi.org/10.1016/j.scico.2024.103086.

[8] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL):48:1–48:29, 2020. doi:10.1145/3371116.

[9] Roger Bosman, Georgios Karachalias, and Tom Schrijvers. No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.8, doi:10.4230/LIPICS.ITP.2023.8.

[10] Roger Bosman, Birthe van den Berg, Wenhao Tang, and Tom Schrijvers. A Calculus for Scoped Effects & Handlers. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/lmcs-20(4:17)2024.

[11] Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. Coherence of type class resolution. *Proc. ACM Program. Lang.*, 3(ICFP):91:1–91:28, 2019. doi:10.1145/3341695.

[12] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428194.

[13] Hendrik Bünder. Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*, pages 129–140. SciTePress, 2019. doi:10.5220/0007556301310142.

[14] Arthur Charguéraud. The Locally Nameless Representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

[15] Dominique Clément, Joëlle Despeyroux, Th. Despeyroux, and Gilles Kahn. A Simple Applicative Language: Mini-ML. In William L. Scherlis,

John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, pages 13–27. ACM, 1986. doi:10.1145/319838.319847.

[16] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 364–377. ACM, 2016. doi:10.1145/2951913.2951945.

[17] Luís Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, UK, 1984. URL: https://hdl.handle.net/1842/13555.

[18] Luís Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

[19] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340, doi:https://doi.org/10.1016/1385-7258(72)90034-0.

[20] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155. ACM, 2011. doi:10.1145/2034773.2034796.

[21] Catherine Dubois and Valérie Ménissier-Morain. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. *J. Autom. Reason.*, 23(3-4):319–346, 1999. doi:10.1023/A:1006285817788.

[22] Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013. doi:10.1145/2500365.2500582.

[23] Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 525–539. ACM, 2017. doi:10.1145/3062341.3062357.

[24] Andrew Gacek. The Abella Interactive Theorem Prover (System Description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008. doi:10.1007/978-3-540-71070-7_13.

[25] Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Math. Struct. Comput. Sci.*, 25(4):867–891, 2015. doi:10.1017/S0960129513000066.

[26] Neil Ghani and Tarmo Uustalu. Explicit Substitutions and Higher-Order Syntax. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '03, page 1–7, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/976571.976580.

[27] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[28] Adam Gundry, Conor McBride, and James McKinna. Type Inference in Context. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 43–54. ACM, 2010. doi:10.1145/1863597.1863608.

[29] Jakob Heirwegh. Lifting Algebraic Effects & Handlers to Scoped Effects & handlers, 2023.

[30] Daniel Hillerström and Sam Lindley. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 15–27, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976022.2976033.

[31] Daniel Hillerström and Sam Lindley. Shallow Effect Handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

[32] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

[33] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020. doi:10.1017/S0956796820000040.

[34] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[35] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

[36] Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. Explicit effect subtyping. *J. Funct. Program.*, 30:e15, 2020. doi:10.1017/S0956796820000131.

[37] Alexis King. eff – screaming fast extensible effects for less. https://github.com/hasura/eff, 2019.

[38] Oleg Kiselyov, Amr Sabry, Cameron Swords, and Ben Foppa. extensible-effects: An Alternative to Monad Transformers. https://hackage.haskell.org/package/extensible-effects, 2019.

[39] Oleg Kiselyov and KC Sivaramakrishnan. Eff Directly in OCaml. *Electronic Proceedings in Theoretical Computer Science*, 285:23–58, Dec 2018. URL: http://dx.doi.org/10.4204/EPTCS.285.2, doi:10.4204/eptcs.285.2.

[40] Daan Leijen. Extensible records with scoped labels. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, volume 6 of *Trends in Functional Programming*, pages 179–194. Intellect, 2005.

[41] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science*, 153, 06 2014. doi:10.4204/EPTCS.153.8.

[42] Daan Leijen. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 486–499, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009872.

[43] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[44] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[45] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad Transformers and Modular Interpreters. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995. doi:10.1145/199448.199528.

[46] Sam Lindley, Cristina Matache, Sean Moss, Sam Staton, Nicolas Wu, and Zhixuan Yang. Scoped Effects as Parameterized Algebraic Theories. In *European Symposium on Programming 2024*, Lecture Notes in Computer Science. Springer, December 2023. URL: https://etaps.org/2024/conferences/esop/.

[47] Sam Lindley, Conor McBride, and Craig McLaughlin. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 500–514, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009897.

[48] Sandy Maguire. polysemy: Higher-order, low-boilerplate free monads. https://hackage.haskell.org/package/polysemy, 2019.

[49] Conor McBride and James McKinna. Functional pearl: i am not a number-i am a free variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pages 1–9. ACM, 2004. doi:10.1145/1017472.1017477.

[50] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[51] Eugenio Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[52] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science. doi:https://doi.org/10.1016/0890-5401(91)90052-4.

[53] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3-4):299–318, 1999. doi:10.1023/A:1006277616879.

[54] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*, chapter 23, pages 247–252. 2008.

[55] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey, Andrew W. Appel, Leonidas Lampropoulos, Lennart Beringer, and Qinxiang Cao. Software Foundations. https://softwarefoundations.cis.upenn.edu/.

[56] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 809–818, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209166.

[57] Maciej Piróg and Sam Staton. Backtracking with cut via a distributive law and left-zero monoids. *J. Funct. Program.*, 27:e17, 2017. doi:10.1017/S0956796817000077.

[58] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-00590-9_7.

[59] Gordon D. Plotkin and John Power. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

[60] Andrey Popp, Rusty Key, Louis Roché, Oleksiy Golovko, Rudi Grinberg, Sacha Ayoun, cannorin, Ulugbek Abdullaev, Thibaut Mattio, and Max Lantas. ocaml-lsp-server 1.15.1-5.0 – opam, Jan 2023. URL: https://opam.ocaml.org/packages/ocaml-lsp-server/ocaml-lsp-server.1.15.1-5.0/.

[61] François Pottier. Hindley-Milner elaboration in applicative style: functional pearl. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 203–212. ACM, 2014. doi:10.1145/2628136.2628145.

[62] Matija Pretnar. Inferring Algebraic Effects. *Log. Methods Comput. Sci.*, 10(3), 2014. doi:10.2168/LMCS-10(3:21)2014.

[63] Matija Pretnar. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). URL: https://www.sciencedirect.com/science/article/pii/S1571066115000705, doi:https://doi.org/10.1016/j.entcs.2015.12.003.

[64] Didier Rémy. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Citeseer, 1994.

[65] John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7_148.

[66] Rob Rix, Patrick Thomson, Nicolas Wu, and Tom Schrijvers. fused-effects: A fast, flexible, fused effect system. https://hackage.haskell.org/package/fused-effects, 2018.

[67] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015. doi:10.1007/978-3-319-22102-1_24.

[68] Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: Stable and coherent implicits. *J. Funct. Program.*, 29:e3, 2019. doi:10.1017/S0956796818000242.

[69] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP*

2019, Berlin, Germany, August 18-23, 2019, pages 98–113. ACM, 2019. doi:10.1145/3331545.3342595.

[70] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. Guarded impredicative polymorphism. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 783–796. ACM, 2018. doi:10.1145/3192366.3192389.

[71] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 1–12. ACM, 2007. doi:10.1145/1291151.1291155.

[72] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 206–221, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454039.

[73] Lau Skorstengaard. An Introduction to Logical Relations. *CoRR*, abs/1907.11133, 2019. URL: http://arxiv.org/abs/1907.11133, arXiv:1907.11133.

[74] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP):86:1–86:29, 2019. doi:10.1145/3341690.

[75] J. Michael Spivey. A Functional Theory of Exceptions. *Sci. Comput. Program.*, 14(1):25–42, 1990. doi:10.1016/0167-6423(90)90056-J.

[76] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

[77] Sam Staton. Instances of Computational Effects: An Algebraic Perspective. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, page 519. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.58.

[78] Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. Soundly Handling Linearity. *Proc. ACM Program. Lang.*, 8(POPL):1600–1628, 2024. doi:10.1145/3632896.

[79] GHC Team. Using GHCi - GHC User's Guide 9.4.4. URL: https://downloads.haskell.org/ghc/9.4.4/docs/users_guide/index.html.

[80] The Coq Development Team. The Coq Proof Assistant, September 2022. doi:10.5281/zenodo.7313584.

[81] The Haskell IDE Team. haskell-language-server documentation. URL: https://haskell-language-server.readthedocs.io/en/latest/.

[82] Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. Fusing industry and academia at GitHub (experience report). *Proc. ACM Program. Lang.*, 6(ICFP):496–511, 2022. doi:10.1145/3547639.

[83] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent Effects for Reusable Language Components. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 182–201. Springer, 2021. doi:10.1007/978-3-030-89051-3_11.

[84] Jeff Vaughan. A proof of correctness for the Hindley-Milner type inference algorithm.

[85] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011. doi:10.1017/S0956796811000098.

[86] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let Should Not Be Generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, page 39–50, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1708016.1708023.

[87] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988. doi:10.1007/3-540-19027-9_23.

[88] Philip Wadler. Comprehending Monads. In Gilles Kahn, editor, *Proceedings of the 1990 ACM Conference on LISP and Functional*

Programming, LFP 1990, Nice, France, 27-29 June 1990, pages 61–78. ACM, 1990. doi:10.1145/91556.91592.

[89] Philip Wadler. Monads for Functional Programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

[90] Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

[91] Stephanie Weirich. Github repository: sweirich/ott, Apr 2022. URL: https://github.com/sweirich/ott/tree/aa65f53ea0587223662aaad9c48cb0770549f018.

[92] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 275–286. ACM, 2013. doi:10.1145/2500365.2500599.

[93] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 1–12, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2633357.2633358.

[94] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi:10.1145/604131.604150.

[95] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. First-class names for effect handlers. *Proc. ACM Program. Lang.*, 6(OOPSLA2):30–59, 2022. doi:10.1145/3563289.

[96] Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. Parallel Algebraic Effect Handlers. *CoRR*, abs/2110.07493, 2021. URL: https://arxiv.org/abs/2110.07493, arXiv:2110.07493.

[97] Jeremy Yallop and Leo White. Lightweight Higher-Kinded Polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing.

[98] Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured Handling of Scoped Effects. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 462–491. Springer, 2022. doi:10.1007/978-3-030-99336-8_17.

[99] Zhixuan Yang and Nicolas Wu. Modular Models of Monoids with Operations. *Proc. ACM Program. Lang.*, 7(ICFP):566–603, 2023. doi:10.1145/3607850.

[100] Francesco Zappa Nardelli. A locally-nameless backend for Ott, Mar 2009. URL: https://fzn.fr/projects/ln_ott/.

[101] Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP):112:1–112:29, 2019. doi:10.1145/3341716.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven
roger.bosman@kuleuven.be
http://www.dtai.cs.kuleuven.be

