KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Efficient Algebraic Effect Handlers

Amr Hany Saleh

Supervisors:
Prof. dr. ir. Tom Schrijvers
Prof. dr. Daniel De Schreye

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

March 2019

Efficient Algebraic Effect Handlers

Amr Hany SALEH

Examination committee:
Prof. dr. ir. Joseph Vandewalle, chair
Prof. dr. ir. Tom Schrijvers, supervisor
Prof. dr. Daniel De Schreye, supervisor
Prof. dr. ir. Gerda Janssens
Prof. dr. ir. Bart Jacobs
Prof. dr. Matija Pretnar
(University of Ljubljana)
Prof. dr. Christophe Scholliers
(University of Gent)

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

March 2019

Preface

Acknowledgments First of all, I want to thank my supervisor, Tom Schrijvers. Your academic advising throughout the last four years has taught me a bunch, particularly being more attentive to detail. I would also like to thank my examination committee members for their insightful remarks. I am so thankful to my colleagues Georgios Karachalias, Alexander Vandenbroucke, Gert-Jan Bottu and Cesar Santos as well, and particularly to Klara Marntiorosian for proof-reading this text. Special thanks also to Emke Van Steekiste for using her astounding translation skills to translate the abstract.

This work would not have come to light without the funding of the Flemish research foundation (FWO).

Many thanks go to my colleagues in the office: George, Steven, Alex, Klara, Ruben and Gert-Jan. I have enjoyed our talks not only about work but also different aspects of life. I will always appreciate these times.

I would like to thank my friends who I got to know throughout this period: Reka Mezei, Roxana Oltean, Stelios Tsampas, Ilias Tsingenopoulos, Elly Louage, Nikolaos Lykouras and Vita Ivanek. You know how stressful some of these periods were and I am thankful to have had you encouraging me through these periods.

Timea Bagosi, your support during the early stages of my PhD has helped me get through those times.

Of course, I am thankful to my friends that I have met throughout my journeys around Europe, who travelled all the way to attend my public defense, keeping that awesome EMCL spirit we had during our studies. I am so happy to have you in Leuven, Andreas Fellner, Ilina Stoilkovska, Hanna Bakker, Itzel vázquez and my travel buddy, Tobias Kaminski. I am also thankful for the encouragement of my very close friends, Andrey Rivkin and Verena Pratissoli.

My Professors and role models, Slim Abdennadher and Haythem Ismail, you have paved the road for me for the success I am having now. Thank you for your teaching

and mentoring that has played a huge role in shaping my perspective in academia and in life. Also, my friends back home, Ali, Noha, Omar, Asmaa and Mostafa, you have always got my back and I am eternally grateful for that.

Finally, I am thankful to be a part of my precious family. My father Hany Beder, my mother Sanaa Elsayed, no words can describe what you have done for me. I hope I made you proud.

To Noura and Haidy

Abstract

In programming languages, a side effect occurs whenever a computation has another effect beside returning its result. There are many examples of side effects such as printing or modifying mutable states. Traditionally, these effects are implicit and built into a language. However, lately, programming using explicit effects is gaining in popularity, since it allows for the free composition of effects. Additionally, it offers a natural separation between the syntax of effects and their semantics, the former given by effect declarations and the latter provided by the so-called *"effect handlers"*. This composition adds an extra layer of modularity to programming with effects and handlers, as it allows the same program to behave differently depending on which handlers are used to deal with its effects. Unfortunately, such flexibility comes at a cost. Even though efficient runtime representations for effects and effect handlers have already been studied, most handler-based programs are still much slower than effect-free hand-written code. In this thesis, we address the performance issue of handler-based programs while preserving the modularity they provide. We investigate this issue in two paradigms: Functional and Logic Programming.

For the Functional Programming part, we focus on Eff, which is an ML-based language with explicit effects. We show that the performance gap can be drastically narrowed (in some cases even closed) utilising a type-and-effect directed optimising compilation. The main aim of these optimisations is to eliminate code related to effects and handlers. Our approach consists of two stages. Firstly, we combine source-to-source transformations with function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for non-effectful computations. Our approach eliminates much of the overhead of handlers and yields competitive performance with hand-written OCaml code.

However, implementing these optimisations for Eff is error-prone because its core language is implicitly-typed, making code transformations very fragile. We also present in this thesis an explicitly-typed polymorphic core calculus for algebraic effect handlers with a subtyping-based type-and-effect system to remedy this fragility.

The proposed calculus tracks the use of subtyping through cast expressions with explicit coercions as subtyping proofs, quickly exposing typing bugs during program transformations.

For the Logic Programming part, we build on delimited control primitives for Prolog to introduce algebraic effects and handlers into the language. Delimited control is used to manipulate the program control-flow dynamically, and to implement a wide range of control-flow and data-flow effects. Unfortunately, delimited control is a rather primitive language feature that is not easy to use. Hence, we use effect handlers as a structured method that makes use of delimited control. We illustrate the expressive power of the feature and provide an implementation using elaboration into the delimited control primitives.

Finally, delimited control adds a non-negligible performance overhead when used extensively. To address this issue, we present an optimised compilation approach that combines partial evaluation with dedicated rewrite rules. The rewrite rules are driven by a lightweight effect inference system that analyses which effects may be called by a program. We illustrate the effectiveness of our approach on a range of benchmarks which shows that the overhead of delimited control is reduced to hand-written Prolog efficiency after the optimisation process.

Beknopte samenvatting

In programmeertalen kunnen neveneffecten voorkomen wanneer een berekening naast het teruggeven van het resultaat nog een bijkomend effect heeft. Voorbeelden van deze neveneffecten zijn het tonen van tekst, of het wijzigen van veranderlijke *toestanden*. Deze effecten zijn gewoonlijk impliciet, en ingebouwd in de taal. Recentelijk wint het gebruik van expliciete effecten echter aan populariteit, omdat hierbij de effecten vrij samengesteld kunnen worden. Het zorgt eveneens voor een duidelijke scheiding tussen de syntaxis van effecten, en de semantiek ervan, waarbij de eerste wordt gegeven door effectdeclaraties, en de tweede door zogenoemde *effect handlers*. Door deze compositie wordt een extra laag van modulariteit ingebouwd in het programmeren van effecten en handlers, omdat het mogelijk is om hetzelfde programma verschillende gedragen te laten vertonen afhankelijk van de handlers die de effecten interpreteren. Jammer genoeg hangt aan deze flexibiliteit ook een prijskaartje. Hoewel er al onderzoek is gedaan naar efficiënte runtime representatie voor effecten en effect handlers, zijn de meeste programma's op basis van handlers nog steeds veel langzamer dan effectvrije handgeschreven code. In deze thesis richten we ons op de kwestie van performantie van handler-gebaseerde programma's, waarbij de modulariteit die ze bieden behouden wordt. We onderzoeken deze kwestie binnen twee paradigma's: Functioneel Programmeren en Logisch Programmeren.

Voor het deel over Functioneel Programmeren focussen we op Eff, een taal op basis van *ML* met expliciete effecten. We laten zien dat de kloof in performantie goed (in sommige gevallen volledig) gedicht kan worden met behulp van type-en-effectgerichte optimaliserende compilatie. Het hoofddoel van deze optimalisaties is om code gerelateerd aan effecten en handlers te elimineren. Onze aanpak bestaat uit twee fasen. Eerst combineren we bron-naar-bron transformaties met specialisatie van functies om de toepasing van handlers drastisch te verminderen. Vervolgens tonen we hoe de brontaal omgezet kan worden tot een handler-loze doeltaal, waarbij geen overhead geleden wordt bij effectloze computaties. Onze aanpak elimineert een groot deel van de overhead veroorzaakt door handlers, en levert een competitieve performantie met handgeschreven OCaml code.

Het implementeren van deze optimalisaties voor Eff is echter foutgevoelig, omdat *types* in de kerntaal van Eff impliciet zijn, wat het transformeren van code een fragiel proces maakt. In deze thesis presenteren wij tevens een polymorfische kerncalculus met expliciete types voor algebraïsche effect handlers met een type-en-effectsysteem dat gebaseerd is op *subtyping*, om deze zwakte aan te pakken. In de voorgestelde calculus is subtyping expliciet: waar een expressie een ander type heeft dan verwacht, worden er *coercions*—dit zijn expliciete bewijzen van subtyping—aan toegevoegd. Zo worden typing bugs snel zichtbaar tijdens programmatransformaties.

Voor het deel over Logisch Programmeren bouwen we verder op *delimited control* primitieven voor *Prolog* om algebraïsche effecten en handlers in de taal te introduceren. Delimited control wordt gebruikt om de *control-flow* dynamisch te manipuleren, en om een breed gamma aan control-flow en *data-flow* effecten te implementeren. Helaas is delimited control een vrij primitief taalconcept, en is het gebruik ervan niet makkelijk. Bijgevolg gebruiken wij effect handlers als een gestructureerde methode die gebruik maakt van delimited control. We illustreren de expressieve kracht van dit taalconcept en geven een implementatie door vertaling naar delimited control primitives.

Tot slot behandelen we de niet verwaarloosbare performantie-overhead die delimited control veroorzaakt wanneer het extensief wordt gebruikt. Hiervoor presenteren we een geoptimaliseerde aanpak voor compilatie, die gedeeltelijke evaluatie met toegewijde herschrijfregels combineert. De herschrijfregels worden gestuurd door een lichtgewicht effectinferentiesysteem dat analyseert welke effectoperaties opgeroepen kunnen worden door een programma. We illustreren de effectiviteit van onze aanpak aan de hand van een aantal maatstaven, waarbij we laten zien dat de overhead veroorzaakt door delimited control gereduceerd wordt tot de efficiëntie van Prolog na het optimalisatieproces.

Translated from English abstract by Imke van Steenkiste and revised by Tom Schrijvers

Contents

List of Figures

Chapter 1

Introduction

Exceptions are a well-known and convenient programming language mechanism for dealing with unexpected situations and unrecoverable problems. The mechanism typically comes in two parts: exceptions are introduced by raising them and they are optionally eliminated by handling them.

By raising an exception the programmer denotes that the program cannot continue its normal execution. The remainder of the code, called the *continuation*, will not be executed. Examples, where this is appropriate, are when the value of a function argument is found to be invalid (e.g., it is outside its expected domain) or when a resource (e.g., a file) is missing.

Because there is no clear path forward, by default an exception is raised, and the further execution of the whole program is aborted. While there is no clear path forward at the program site where the exception is raised, the calling context that leads to the exception may have a strategy to recover from the exception thanks to its broader view of the overall program's purpose.

For instance, if an exception is raised because a file is missing, the task that needs the file may not know how to recover, but the GUI that invoked the task may decide to recover by asking the user for an available file to proceed with. To recover from an exception, the calling context intercepts and *handles* it by specifying an alternative function to invoke when the normal function raises an exception. Hence, exceptions and exception handlers manipulate a program's control-flow. The program's normal control flow is aborted when an exception is raised and execution instead jumps to the exception handler.

In this thesis, we study *algebraic effects and their handlers* [73, 66], which are a

powerful generalisation of exceptions and exception handlers. Like an exception, an algebraic effect can be raised, though we typically say that it is *called* rather than raised. In the same way that raised exceptions can be handled, algebraic effects that were called can be handled too, by an effect handler.

The big difference with normal exception handlers is that the continuation of an algebraic effect is not discarded, but instead captured and made available to the handler. This new capability gives the handler the option to resume that continuation, possibly even multiple times, as part of its strategy for dealing with the algebraic effect. Because of this extended functionality, effect handlers are sometimes informally called "exception handlers on steroids".

The typical applications of algebraic effects and handlers are quite different from those of exceptions. Instead of signalling abnormal situations, they are usually part of the normal program execution. Notably, they are used for programmer-defined control-flow and dataflow effects. Indeed, we can find many such examples in the literature. Bauer and Pretnar [6] give a range of examples implementing non-deterministic choice with backtracking, exceptions, state passing, probabilistic choice and transaction lookups and updates. Lindley [54] uses it to define parser combinators. Schrijvers et al. [84] show how to implement different search heuristics for Prolog [14]. Dolan et al. [24] use effect handlers for implementing schedulers of concurrent systems.

One of the main benefits algebraic effects and handlers provide to the programmer is an additional layer of modularity: the use of effects is now decoupled from their interpretation. The same code that uses effects can serve different purposes by handling those effects in different ways. For instance, the same nondeterminism effects are interpreted differently by different search heuristics in the work of Schrijvers et al., while the same process effects are interpreted differently by different schedulers in the work of Dolan et al.

However, the additional modularity provided by algebraic effects comes at the cost of performance, an aspect many implementations in the literature pay little or no attention to. Nevertheless, algebraic effects are inherently more expensive than regular exceptions because capturing and resuming continuations is more involved than merely discarding the continuation. This brings us to the primary research question of this thesis:

> How can we achieve better performance for algebraic effects and handlers while maintaining their modularity?

To answer this question, we investigate the suitability of optimised compilation for automatically generating efficient code that rivals non-modular hand-written implementations.

1.1 Thesis Overview and Scientific Output

This thesis consists of two parts, which revolve around the two different programming paradigms, Functional Programming and Logic Programming, for which we have investigated the optimised compilation of algebraic effects.

1.1.1 Effect Handlers in Functional Programming

The first part concerns functional programming and focuses in particular on the Eff language [72]. Eff is a functional language based on OCaml [52] with native support for algebraic effects and handlers.

This part consists of three chapters. First, Chapter 2 provides the necessary background on effect handlers in general and on Eff in particular for the next two chapters. The next two chapters present our contributions.

Eff Optimisation Techniques Chapter 3 discusses our approach to the optimised compilation of Eff. This approach consists of a combination of several techniques, including term rewriting rules, function specialisation and specialised code generation for non-effectful computations. In addition to formalizing these techniques, we have also implemented the optimisations in the Eff compiler and present the benchmark results we have obtained.

This chapter is based on my contribution in the technical report:

"Pretnar, M., Saleh, A. H., Faes, A., and Schrijvers, T. Efficient compilation of algebraic effects and handlers. Technical Report CW 708, KU Leuven Department of Computer Science, 2017."

An Explicitly Typed Core Calculus for Eff Chapter 4 presents a new core calculus for Eff. While implementing the optimisations for Eff in Chapter 3, we discovered that Eff's pre-existing core-language is unsuitable for implementing sophisticated type-directed optimisations since the lack of explicit types made the process highly error prone. This chapter presents a new calculus designed from the ground up with that application in mind. In order to facilitate the tracking and updating of type-and-effect information, our calculus is explicitly typed in the style of System F. A particular novelty is that the proposed calculus also tracks the use of subtyping explicitly using cast expressions with coercions as subtyping proofs. This chapter is based on the following conference paper:

"Saleh, A. H., Karachalias, G., Pretnar, M., and Schrijvers, T. Explicit effect subtyping. In European Symposium on Programming (2018), Springer, pp. 327–35"

1.1.2 Effect Handlers in Logic Programming

The second part of this thesis ports the notion of algebraic effects and handlers to Prolog, a Logic Programming language.

Basic Compilation into Prolog The first part of Chapter 5 presents our language design for algebraic effects and handlers in Prolog, together with a simple source-to-source compilation scheme from the effect handler syntax into Prolog. We also provide a Prolog meta-interpreter to interpret the effect handler syntax into Prolog with delimited control. However, the resulting compiled code is rather inefficient compared to native Prolog.

Optimised Compilation into Prolog The second part of Chapter 5 investigates optimised compilation for algebraic effects and handlers in Prolog. We combine techniques inspired by our work on Eff, including several additional rewriting rules, with partial evaluation to obtain code that is close to its hand-written counterpart.

Most of the content in this chapter is based on the following conference paper:

"Saleh, A. H., and Schrijvers, T. Efficient algebraic effect handlers for Prolog. Theory and Practice of Logic Programming 16, 5-6 (2016), 884–898"

Chapter 2

Background

This chapter provides the technical background on the notion of algebraic effects and handlers that are necessary for this thesis.

Different approaches have been used in the literature to present this notion. It was first introduced by Plotkin and Pretnar [73, 66] in the form of a categorical model. Later Bauer and Pretnar created a language design around this notion, formalised by an abstract syntax, a type system and a denotational semantics that follows the categorical model; they called this language Eff [72]. Later, Bauer and Pretnar [6] also provided an operational semantics for Eff.

For this thesis, we focus on the programming languages aspect of algebraic effects and handlers. Thus we present algebraic effects and handlers in the form of a core calculus for a simplified version of Eff that we call Effy.

In Section 2.1, we provide the necessary background for the notions used in defining a programming languages which are mainly syntax, type-system and semantics. In Section 2.2, we informally introduce Eff by means of a few examples. Section 2.3 gives the formal definitions of Effy. Later, in Section 2.4, we briefly discuss different alternatives and related work.

2.1 Theory of Programming Languages

This section gives a brief overview of the required background related to the theory of programming languages that is used in this work. A programming language is

defined by an abstract syntax (grammar) and execution behaviour for this grammar. Programming languages are either strongly-typed, weakly-typed or un-typed. The semantics given to a language provide meaning to the syntax and a way of execution.

2.1.1 Abstract Syntax

Syntax is a cornerstone for defining any programming language. Abstract syntax is a hierarchy of rules used to define a language's grammar. An abstract syntax definition consists of *sorts* that contain *operators*. The operators are the *terminals* and *non-terminals* of a language [94].

We represent the expressions of a language using *abstract syntax trees*. Abstract syntax trees are data structures that allow for the traversal inside expressions while complying to the language's grammar.

Example The following example shows the abstract syntax of a small programming language and how a syntax tree for one of its expressions can be represented.

The language contains only variables that can have integers or booleans as values and one operation which is subtraction. The abstract syntax looks as follows:

$$\begin{array}{rrll} \text{Expression } e & ::= & x & \text{variable} \\ & | & e_1 - e_2 & \text{subtraction} \end{array}$$

A syntax tree for the expression $x_1 - x_2$ starts with the subtraction operator that has two leaves, one for x_1 and the other for x_2.

2.1.2 Type Systems

A type system is the formal classification of language objects in different categories, each category supporting a limited set of operations [62]. We can call a category in such a theory a type. A type is essentially a set of values. For example, a numeric literal 5 can be assigned type `int` (integer), while the boolean value `true` can be assigned the type `bool`. We also assign a type to an expression, like `int` to 2+3, if that expression evaluates to a value of that type.

Type systems literature Type systems are a central topic in the field of programming languages theory, which studies the formal definition of programming

languages and their metatheory. Well-known examples of type systems are System F [62, Chapter 23] and the Hindley-Milner [17, 36] type systems.

Type systems example The type system specifies the classification by means of a relation, also called a judgment, of the form $\Gamma \vdash v : A$. This judgment denotes that for the given typing environment Γ, value v has type A. The judgment is defined by means of a number of inference rules, often one per syntactic construct (e.g., functions, handlers, etc.), that explain how to assign types to those constructs in terms of the types of their components. For example, an expression $(x - y)$ can be assigned type `float`, if both x and y have type `float`. This inference rule is formally written as:

$$\frac{\Gamma \vdash x : \texttt{float} \qquad \Gamma \vdash y : \texttt{float}}{\Gamma \vdash x - y : \texttt{float}}\ \textsc{Subtract} \tag{2.1}$$

Subtract is the rule name. The above rule can be read as follows: *If the variable x has type* `float` *and y has type* `float`*, then x–y has type* `float`. We get the types of the variables x and y from a typing environment Γ, which records the types of the free variables. For example, if Γ contains information about x and y such that x : `float` and y : `float`, then the Subtract rule would type-check correctly for the expression $x - y$. Otherwise, if the information about x or y is missing from the typing environment, or they have other types than `float`, then the Subtract rule would reject the expression.

Type checking The main purpose of a type system is to rule out programs that contain particular classes of (possibly latent) bugs; these programs are considered *ill-typed* because they cannot be assigned a type in the type system. For example the expression *45 + true* is ill-typed, since we cannot add a numerical value to a boolean value. Therefore, we need a type system that eliminates these ill-typed expressions from the programs. This procedure of checking for ill-typed program and ensuring that a program is well-typed is called *type-checking*. A programming language with a type-checker is said to hold the *type-safety* property.

Subtyping Some type systems have a notion of *subtyping* [62, Chapter 15]. Subtyping is a partial order on types. For example, we might say that the integer type is a subtype of the floating point type, or, conversely, that floating point numbers are a supertype of integers. We will denote relations as follows $A \leqslant A'$ to mean that the type A is a subtype of the type A'.

In the Subtract rule, suppose x has the type `int`. Then the rule rejects the type of the variable as it expects `float`. However, this need not be the case in

a type system with subtyping. Suppose we have the subtyping rule $\texttt{int} \leqslant \texttt{float}$ that declares that the integer type is a subtype of the float type.[1] Moreover, with subtyping we would have the following "subsumption" typing rule in addition to the SUBTRACT rule:

$$\frac{\Gamma \vdash e_1 : A \qquad A \leqslant A'}{\Gamma \vdash e_1 : A'} \ \textsc{TySub} \tag{2.2}$$

Using the TYSUB rule, we can treat a variable as having any of the super types of its type. Therefore the type system would not complain when e_1 of type `int` appears in a subtraction, as `int` is a subtype of `float`, and thanks to the subsumption rule e_1 can thus be seen as being of type `float`.

There are two main notions of subtyping in programming languages:

- Implicit subtyping: The subtyping occurs automatically inside the compiler. The previous example that explains subtyping uses this technique. Many subtyping rules can be applied to an expression until the desired type is achieved.
- Explicit subtyping: The language supports casts to an expression where subtyping proof can appear. In order to change the type of an expression in an explicitly subtyped language, a subtyping witness has to accompany the expression showing the required subtyping relation.

Type inference The notion of type inference or type reconstruction [63, Chapter 10] refers to automatically infer the type of an expression of a language. For example, the following expression written in simply typed Lambda-calculus [62, Chapter 9]:

$$\lambda x.(x + 1)$$

We assume that the type for this expression is a *function type* $\alpha \to \beta$ where α and β are type variables and $\to$ states that the type is a function that takes the input type α and returns the output type β. Since the computation inside the lambda abstraction is an addition, the resulting type of the computation needs to be an integer. Also, the variable x is used in the addition so the type of x is also an integer. Therefore the inferred type of this expression is *int* $\to$ *int*.

Several algorithms have been developed in the literature for type inference. Local type inference was proposed by Pierce and Turner [64]. A different approach for type inference is the greedy approach developed by Cardelli [13]. The inference algorithm developed in Hindley-Milner type system [18] uses unification as the core of the algorithm. Several constraint-based type inference algorithms feature a two-phase

[1] By just adding a .0 to the integer

approach: firstly generating constraints while inferring the type of an expression and secondly resolving these constraints by means of substitutions [1, 92, 69].

2.1.3 Operational Semantics

Semantics is the study of the meaning of languages. Formal semantics for programming languages is the study of formalisation of the practice of computer programming [59]. Operational semantics defines the behaviour of a program. It does so by describing how a valid program is interpreted through sequences of computational steps that are also called reduction steps. A program reduces to a terminal value that gets returned to the user.

Operational semantics of a programming language can be defined in two forms:

Small-step semantics Plotkin [67] introduced Small-step semantics (also called structural semantics). The main idea of structural semantics is to define the behaviour for each sort in the language. The specification takes the form of a set of inference rules that define the valid transitions from one expression to another. It uses structural induction approach over all the sorts of the language. Small-step semantics gives control over the details and order of evaluation which allows for more natural proofs of properties for the language.

Small-step semantics are algorithmic, meaning that we can write a terminating program that reduces an expression to its terminal value using well-defined small-step semantics.

Big-step semantics Big-step semantics was introduced by Kahn [40], and it directly formulates to "this term evaluates to that final value" written $t \Downarrow v$. It provides an interpretation of how a term evaluates to value. The result of evaluating a big-step semantics is usually a tree of inference rules and a terminal value.

The main disadvantage of big-step semantics is that it does not provide a step-by-step evaluation of expressions in the language which makes the implementation of big-step semantics in a compiler a hard task.

Type preservation Type preservation is an essential property for the operational semantics of a language. It states that well-typedness of programs remains invariant under the transition rules of the language. Meaning that the evaluation rules or reduction rules that are given by the operational semantics do not cause a change to the type of the expressions being evaluated.

2.2 Eff by Example

This section explains Eff informally by means of several examples that capture its essence. The examples are variations of those first presented by Bauer and Pretnar [6].

2.2.1 Basic Example

Let us start with a basic example which explores non-determinism in Eff.

Declaring an effect This example starts with the declaration of an *effect*—note that Eff's syntax resembles that of OCaml:

```
effect decide : unit -> bool;;
```

This declaration introduces an effect, called `decide`, that takes a value of type `unit` and returns a value of type `bool`. Both `unit` and `bool` are built-in types: `unit` is a type with a single value denoted `()`, and `bool` has two values denoted `true` and `false`. Note that the declaration does not provide an implementation for `decide` or otherwise assigns a meaning to it.

Using effects Here is an example use of the `decide` effect in a larger expression.

```
let x = (if #decide () then 10 else 20) in
let y = (if #decide () then 0 else 5) in
x - y
```

Note that the effect call is prefixed by #. Like a function call it receives a `unit` value as a parameter, and its context, the conditional expression assumes that it produces a boolean; these two facts are in accordance with the declared type signature for the effect.

Informally, we can understand the whole expression as assigning either the value 10 or 20 to the variable `x`, and the value 0 or 5 to the variable `y`, depending on the successive outcomes of the two calls to `decide`, and then subtracting `y` from `x`. Yet, running the previous code results in an *unhandled effect* error. The reason for this is that the effect `decide` is called while no definition is provided for it.

Defining a handler This is where *(effect) handlers* come in. Handlers give meaning to effects that arise in computations by intercepting their calls, "handling" them and passing control back to the computation.[2] If a second effect is called, the control is passed back to the handler and so on. Finally, the handler decides what to do with the final result of the computation. For instance, the following code fragment "wraps" the above expression in a handler for the `decide` effect.

```
handle
  let x = (if #decide () then 10 else 20) in
  let y = (if #decide () then 0 else 5) in
  x - y
with
| return x -> x
| #decide y k -> k true
```

The first line, also called *clause*, of the handler, `| return x -> x`, states what to do when the computation is finished. In this case, it just returns the final result, which is bound to the variable `x`, as it is. We refer to this clause as the *return clause* or the *value clause*.

The second clause, `| #decide y k -> k true`, explains that the `decide` effect should always return `true`. This is called the *effect clause* or the *operation clause*.The clause receives two inputs: the first is the parameter `y` of type `unit`, which is not used, and the second is the continuation (or evaluation context) `k` that captures the context in which the effect was called in the form of a function. For instance, for the first call to `decide` above the continuation is

```
k  = fun rx ->
       handle
         let x = (if rx then 10 else 20) in
         let y = (if #decide () then 0 else 5) in
         x - y
       with
       | return x -> x
       | #decide y k -> k true
```

The *body* of the clause, `k true`, applies this continuation to the value `true`, thus resulting in the new computation

```
handle
  let x = (if true then 10 else 20) in
```

[2] The interrupt handling mechanism of operating systems works in a similar fashion, but at an entirely different level.

```
    let y = (if #decide () then 0 else 5) in
    x - y
with
| return x -> x
| #decide y k -> k true
```

Note that this computation is still wrapped in the original handler. As a consequence, the second effect call receives continuation

```
k  = fun ry ->
       handle
         let y = (if ry then 0 else 5) in
         10 - y
       with
       | return x -> x
       | #decide y k -> k true
```

and is handled in the same way. Eventually, the computation returns $10 - 0 = 10$.

2.2.2 The N-Queens Problem

Let us now look at a slightly more significant and more meaningful program to motivate the use of effects and handlers. In particular, their main advantage is modularity: we can reuse the same code, but apply different handlers to interpret its effects differently and thus obtain different results.

We demonstrate this advantage on a program that solves the well-known N-Queens Problem[3], and contrast an implementation without effects and handlers (i.e., written in plain OCaml) with an Eff implementation that does use them.

Without Effects and Handlers The following OCaml code implements a naive solver for the N-Queens Problem. While the problem may have multiple solutions for a given problem size N, the implementation only computes and returns the first it encounters.

```
1 let no_attack (x, y) (x', y') =
2   x <> x' && y <> y' && abs (x - x') <> abs (y - y')
3
```

[3]The problem simply asks: How can N number of queens be placed on an $N \times N$ chessboard so that no two of them attack each other?

```
4   let rec not_attacked x' = function
5     | [] -> true
6     | x :: xs -> if no_attack x' x then not_attacked x' xs
7                                    else false
8
9   let available number_of_queens x qs =
10    let rec loop (possible, y) =
11      if y < 1 then
12        possible
13      else if not_attacked (x, y) qs then
14        loop ((y :: possible), (y - 1))
15      else
16        loop (possible, (y - 1))
17    in
18    loop ([], number_of_queens)
19
20  let queens_one number_of_queens =
21    let rec place (x, qs) =
22      if x > number_of_queens then Some qs else
23        let rec choose = function
24          | [] -> None
25          | y :: ys ->
26            begin match place ((x + 1), ((x, y) :: qs)) with
27              | Some qss -> Some qss
28              | None -> choose ys
29            end
30        in
31        choose (available number_of_queens x qs )
32    in
33    place (1, [])
```

Let us go through the functions provided above and give an explanation of what they do. The function `no_attack` takes two locations and checks that these locations cannot attack each other. This is done by checking 1) that their `x` (row) and `y` (column) coordinates are different, and 2) that they cannot attack each other diagonally (using the absolute value function). The function is used in the `not_attacked` function which takes a queen position `x` and a list of positions of other queens. Then, it checks that none of the positions in the list can attack the additional queen at position `x`.

We use the previous function in the `available` function that takes the number of queens, the current row `x` where an additional queen should be placed and the

already placed queens' locations qs. Then it returns the available column positions ys where the additional queen cannot attack any of the previously placed queens.

The main function is queens_one. This function is responsible for correctly placing the queens. It returns a value of type (int list) option which denotes optionally a list of positions for the placed queens. The 'a option datatype features two shapes of values: Some x denotes a value x of type 'a, while None denotes the absence of such a value.

The main function proceeds by placing the queens one by one, selecting for each queen one of the available positions where it does not attack the previously placed queens. In case that there are no places to place the queen at hand, it backtracks and considers an alternative position for one of the previously placed queens. This is done using the two mutually recursive functions place and choose. The first is responsible for the correct placement of the queens, while the latter backtracks over the available column locations for each queen. The interesting line of code in the previous example is line 25. It assigns the current queen with the position (x,y) and adds it to the list of placed queens qs. Then it places the remaining queens, from x+1 onwards, with respect to the extended list of placed queens (x, y)::qs. If there is a full placement, this is returned in line 26. Otherwise, in line 27, an alternative position is considered for queen x.

While the above code works well, it only computes the first solution to the problem. Suppose our requirement changes and we want all solutions. Then we have to modify the code as follows to compute the list of all solutions.

```
let queens_all number_of_queens =
  let rec place (x, qs) =
    if x > number_of_queens then [qs] else
      let rec choose = function
        | [] -> []
        | y :: ys ->
            place ((x + 1), ((x, y) :: qs)) @ choose ys
      in
      choose (available (number_of_queens, x, qs))
  in
  place (1, [])
```

Observe that this code is relatively similar to the single-solution version. Nevertheless, we had to modify the two main functions place and choose to return all solutions. Instead of selecting only one column y for a queen in a row x, we go through all the available columns selections and then we concatenate all the results in a list using the @ function. So we are traversing the whole search tree instead of only selecting one branch.

With Effects and Handlers Let us use EFF to solve the N-queens problem instead of OCAML. This solution is centered around two effects:

```
effect decide : unit -> bool ;;
effect fail : unit -> empty ;;
```

The first effect, `decide`, is familiar from the basic example. We use it here to choose between two alternatives, denoted by `true` and `false`. The second effect, `fail`, indicates that there is no valid alternative. Its return type is `empty`, which is a built-in type without values. Hence, `fail` cannot possibly return; its only option is to abort the current computation. It is usually wrapped in a pattern match as follows.

```
abort = function () -> (match #fail () with)
```

The function `abort` has type `unit -> 'a`, i.e., it pretends to return a value of any type. Of course, the function never returns for there is no value of the empty type that `#fail ()` can yield. (For the same reason the pattern match is empty: it is pointless to provide a pattern to match on if no value is forthcoming.)

The code below uses these two effects in the `choose` function of the solver.

```
let queens number_of_queens =
  let rec choose = function
    | [] -> abort ()
    | x::xs -> if #decide () then x else choose xs
  in
  let rec place (x, qs) =
    if x > number_of_queens then qs else
      let y = choose (available number_of_queens x qs) in
      place ((x + 1), ((x, y) :: qs))
  in
  place (1, [])
```

In order to recover the single-solution behaviour of the program, we use the following handler:

```
let option_handler = handler
  | return y -> (Some y)
  | #decide () k -> ( match k true with
```

```
                        | Some x -> Some x
                        | None -> k false )
    | #fail _ _ -> None
```

The return clause wraps the result in the data-type constructor `Some` to denote the presence of a result.The `#decide` clause calls the continuation `k` with `true`, and if it returns a result (`Some x`), the handler returns that result. If it does not return a result (`None`), then the handler explores the other branch by calling the continuation with `false`. The `#fail` clause discards the continuation and returns `None`.

If we want all solutions instead, we do not have to modify the solver code. We can just wrap it in a different handler:

```
let choose_all = handler
  | return x -> [x]
  | #fail _ k -> []
  | #decide _ k -> k true @ k false
```

The `choose_all` handler collects all solutions in a list. Its first clause turns a single result into a singleton list. Similarly, the second clause turns a `fail` effect into an empty list. Finally, when a choice is made with `decide`, the handler explores both alternatives by applying the continuation twice, once to `true` and once to `false`, and appending the two resulting lists.

Running `queens 4` with the two different handlers yields the following results.

```
# with option_handler handle queens 4;;
(int * int) list option =
Some [(4, 3); (3, 1); (2, 4); (1, 2)]

# with queens_all handle queens 4;;
(int * int) list list =
[[(4, 3); (3, 1); (2, 4); (1, 2)];
 [(4, 2); (3, 4); (2, 1); (1, 3)]]
```

In summary, we have observed that effects and handlers allow us to quickly and easily change the behaviour of a program and thus enable its reuse for different purposes. While we have only given a single example, solving combinatorial problems, effect handlers can be used in many more settings. For instance, Bauer and Pretnar [6] show applications to printing, probabilistic programming, implicit state passing, and cooperative multi-threading.

2.3 Formal Definition of Eff

The previous section has provided an informal description of the EFF language by means of examples. In this section, we give a more formal treatment of the language in terms of a non-polymorphic fragment of EFF, called EFFY. This presents a more principled basis for understanding the theory behind EFF. We will build on this core calculus in later chapters.

2.3.1 Syntax

There are two groups of syntactic sorts in EFFY: types and terms.

Types

The top part of Figure 2.1 shows the types that an EFFY term can take. There are two sorts of types: simple types A and dirty types $\underline{C}$.

Simple types contain primitive types such as `int`, `bool` and `unit` that present integers, booleans and unit values respectively. In addition to primitive types, there are function types $A \to \underline{C}$, and handler types $\underline{C} \Rrightarrow \underline{D}$. We explain below in detail how to assign these types to their respective terms.

Dirty types are simple types accompanied by a set Δ that we call dirt. The dirt is a set of effects that may occur when executing a computation.

Terms

The terms of EFFY are divided into two sorts: values and computations. The lower part of Figure 2.1 shows the terms of EFFY.

First, we take a look at the values; they are terms that by themselves are inert and do not perform any computation. They can be variables x, units $()$, lambda abstractions $\texttt{fun}\ x \mapsto c$ or handlers h. A handler h consists of a return clause $(\texttt{return}\ x \mapsto c_r)$ and zero or more effect clauses. We abbreviate $\texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}$ as $[\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}$, and write $\mathcal{O}$ to denote the set $\{\texttt{Op}_1, \ldots, \texttt{Op}_n\}$.

Computations are the terms that compute and may produce effects. A value can be lifted into a computation using the `return` v term. Calling an effect creates another primitive computation. The `do` construct sequences two computations and passes the result from the first to the second. The $\texttt{handle}\ c\ \texttt{with}\ e$ construct applies a handler to a computation, handling some of its effects and producing

Types

$$
\begin{array}{rl}
\text{Simple types } A, B & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid A \to \underline{C} \mid \underline{C} \Rrightarrow \underline{D} \\
\text{Dirt } \Delta & ::= \{\texttt{Op}_1, \texttt{Op}_2, \dots\} \\
\text{Dirty types } \underline{C}, \underline{D} & ::= A \,!\, \Delta
\end{array}
$$

Terms

$$
\begin{array}{rlll}
\text{value } v & ::= & x & \text{variable} \\
& \mid & \texttt{k} & \text{constant} \\
& \mid & \texttt{fun}\ x \mapsto c & \text{function} \\
& \mid & \{ & \text{handler} \\
& & \texttt{return}\ x \mapsto c_r, & \text{return case} \\
& & \texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \dots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n} & \text{ops cases} \\
& & \} & \\
\text{computation } c & ::= & v_1\, v_2 & \text{application} \\
& \mid & \texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_2 & \text{recursive let} \\
& \mid & \texttt{return}\ v & \text{returned value} \\
& \mid & \texttt{Op}\, v & \text{operation call} \\
& \mid & \texttt{do}\ x \leftarrow c_1; c_2 & \text{sequencing} \\
& \mid & \texttt{handle}\ c\ \texttt{with}\ v & \text{handling} \\
& \mid & \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \text{If-Then-Else}
\end{array}
$$

Figure 2.1: Types and terms of Effy

a new computation. Function application $v_1\ v_2$ also constructs a computation. Finally, $\texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ chooses between two computations based on the condition v.

2.3.2 Type System

Subtyping The type system of Effy features subtyping. Consider that a computation returning values of type A and potentially calling operations from the set Δ is assigned the type $A \,!\, \Delta$. This set Δ is an over-approximation of the operations that are called, and may safely be increased. This induces a natural subtyping judgement $A \,!\, \Delta \leqslant A \,!\, \Delta'$ on dirty types. As dirty types can occur inside pure types (namely in arrow and handler types), we also get a derived subtyping judgement on pure types. Effy's Subtyping rules are given in Figure 2.2.

Pure Types Subtyping

$$\boxed{A \leqslant A'}$$

$$\frac{}{\texttt{bool} \leqslant \texttt{bool}}\ \textsc{Sub-bool} \qquad \frac{}{\texttt{int} \leqslant \texttt{int}}\ \textsc{Sub-int} \qquad \frac{A' \leqslant A \qquad \underline{C} \leqslant \underline{C}'}{A \to \underline{C} \leqslant A' \to \underline{C}'}\ \textsc{Sub-}\to$$

$$\frac{\underline{C}' \leqslant \underline{C} \qquad \underline{D} \leqslant \underline{D}'}{\underline{C} \Rrightarrow \underline{D} \leqslant \underline{C}' \Rrightarrow \underline{D}'}\ \textsc{Sub-}\Rrightarrow$$

Dirty Types Subtyping

$$\boxed{\underline{C} \leqslant \underline{C}'}$$

$$\frac{A \leqslant A' \qquad \Delta \subseteq \Delta'}{A\ !\ \Delta \leqslant A'\ !\ \Delta'}\ \textsc{Sub-!}$$

Figure 2.2: Subtyping for pure and dirty types of EFFY

The judgement $A \leqslant A'$ states that the pure type A is a subtype of A'. While the judgement $\underline{C} \leqslant \underline{C}'$ states that the dirty type $\underline{C}$ is a subtype of $\underline{C}'$, rules SUB-`bool` and SUB-`int` represent reflexivity for integer and boolean types. Notice that for SUB-$\to$ and SUB-$\Rrightarrow$, the subtyping judgement is contra-variant in the argument types of functions and handlers. Rule SUB-! shows subtyping for computation types. If the pure parts of two computation types are subtypes, and the dirts are in a subset relation, then the two dirty types are in a subtyping relation.

Value Typing After we presented the subtyping rules for EFFY, now, let us look at the typing rules. First, we consider the typing rules for EFFY values. The typing judgement for values takes the form $\Gamma \vdash v : A$ which states that: given a typing environment Γ, value v has value type A. The top-level environment Σ contains the typing of effects and constants. Every effect is defined by its name and has an input and an output simple types, e.g., `(effect decide : unit -> bool)`.

The top of Figure 2.3 provides the inference rules that define value typing. Rule TYSUBVAL is the typing subsumption rule for values. It connects the subtyping rules for values to the type system. A value v of type A can have the type A' if the subtyping relation $A \leqslant A'$ holds. Rule TYVAR gets the type of

a free variable x from the typing environment Γ. The rule TyAbs handles lambda abstractions. The lambda variable is added to the typing environment, and then the body computation is type-checked to have a dirty type. The overall type is an arrow type $A \to \underline{C}$, where A is the expected input type of the function, and $\underline{C}$ is the resulting computation type.

The handler case is slightly more complicated. Rule TyHand gives typing for handlers. It requires that the right-hand sides of the return clause and all operation clauses have the same computation type ($B \,!\, \Delta$), and that all operations mentioned are part of the top-level environment Σ. The result type takes the form $A \,!\, \Delta \cup \mathcal{O} \Rrightarrow B \,!\, \Delta$, capturing the intended handler semantics: given a computation of type $A \,!\, \Delta \cup \mathcal{O}$, such that $\mathcal{O}$ is the set of effects that the handler handles and Δ contains the remaining effects that appear in the input computation and are not handled by the handler. The result type of the output computation is B. The handler handles the operations $\mathcal{O}$ and propagates the unhandled operations Δ to the output type.

Computation Typing Next, we describe the rules for computations. The typing judgement for computations takes the form $\Gamma \vdash c : \underline{C}$: Given a typing environment Γ, the computation c has the dirty type $\underline{C}$. Computations have a dirty type due to the potential of triggering an effect. This judgement is defined in the bottom half of Figure 2.3. Rule TySubComp is the subsumption rule for dirty types. TyLetRec type checks recursive functions. Rule TyReturn handles `return` v computations. The keyword `return` effectively lifts a value v of type A into a computation of type $A \,!\, \emptyset$. Rule TyOp checks operation calls. First, we ensure that v has the appropriate type, as specified by the signature of `Op`. The side condition `Op` $\in \Delta$ ensures that the called operation `Op` is captured in the result type. Rule TyDo handles sequencing. Given that c_1 has type $A \,!\, \Delta$, the pure part of the result of type A is bound to term variable x, which is brought in scope for checking c_2. All computations in a `do`-construct should have the same effect set Δ. Rule TyHandleWith eliminates handler types, just as rule TyApp eliminates arrow types. Rule TyITE makes sure that the condition is of type `bool` and the two branches are of the same type.

Typing Derivation Example After presenting the syntax and typing rules for Effy, we illustrate how they work on a small example. Let us type-check the following Effy computation:

```
effect flip: bool -> bool ;;

handle
  do x <- (#flip true) in (#flip x)
```

Expressions

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

$$\boxed{\Gamma \vdash v : A}$$

$$\textsc{TySubVal}\ \frac{\Gamma \vdash v : A \qquad A \leqslant A'}{\Gamma \vdash v : A'} \qquad \textsc{TyVar}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{TyConst}\ \frac{(\mathtt{k} : A) \in \Sigma}{\Gamma \vdash \mathtt{k} : A}$$

$$\textsc{TyAbs}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}}$$

$$\textsc{TyHand}\ \frac{\Gamma, x : A \vdash c_r : B\ !\ \Delta \qquad \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad \Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B\ !\ \Delta \vdash c_{\mathtt{Op}} : B\ !\ \Delta\right]_{\mathtt{Op} \in \mathcal{O}}}{\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A\ !\ \Delta \cup \mathcal{O} \Rrightarrow B\ !\ \Delta}$$

Computations

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

$$\textsc{TySubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{C}'}{\Gamma \vdash c : \underline{C}'} \qquad \textsc{TyApp}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{TyLetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2 : \underline{D}}$$

$$\textsc{TyReturn}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{return}\ v : A\ !\ \emptyset} \qquad \textsc{TyOp}\ \frac{(\mathtt{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathtt{Op}\, v : B\ !\ \{\mathtt{Op}\}}$$

$$\textsc{TyDo}\ \frac{\Gamma \vdash c_1 : A\ !\ \Delta \qquad \Gamma, x : A \vdash c_2 : B\ !\ \Delta}{\Gamma \vdash \mathtt{do}\ x \leftarrow c_1; c_2 : B\ !\ \Delta} \qquad \textsc{TyHandWith}\ \frac{\Gamma \vdash v : \underline{C} \Rrightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D}}$$

$$\textsc{TyITE}\ \frac{\Gamma \vdash v : \mathtt{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \underline{C}}$$

Figure 2.3: EFFY Type System

```
with
  | #flip i k -> if i
                  then k false
                  else k true
  | return y -> return y
```

In order to type-check this computation, we start from an empty typing environment. Also, the set Σ contains the effect type signature `flip : bool -> bool` and the signatures of the constants `true` : `bool` and `false` : `bool`.

We go through the derivation in a bottom-up fashion. In order to type-check the `handle/with` computation, we use the rule TyHandleWith as follows:

$$\textsc{TyHandWith}\ \frac{\mathcal{T}_a \qquad \mathcal{T}_b}{\emptyset \vdash \texttt{handle do}\ x \leftarrow (\#\mathit{flip}\ \texttt{true}); (\#\mathit{flip}\ x)\ \texttt{with}\ h : \texttt{bool}\ !\ \emptyset}$$

where h abbreviates the handler above, and the two derivations $\mathcal{T}_a$ and $\mathcal{T}_b$ are for the handler and the `do` computation respectively. The derivation $\mathcal{T}_a$ for the handler looks as follows:

$$(\mathcal{T}_a)\textsc{TyHand}\ \frac{\mathcal{T}_1 \qquad \mathcal{T}_2}{\emptyset \vdash h : \texttt{bool}\ !\ \{\mathit{flip}\} \Rrightarrow \texttt{bool}\ !\ \emptyset}$$

Derivation tree $\mathcal{T}_1$ type-checks the return clause of the handler using TyReturn as follows:

$$(\mathcal{T}_1)\ \textsc{TyReturn}\ \frac{\textsc{TyVar}\ \dfrac{y \in \{y : \texttt{bool}\}}{y : \texttt{bool} \vdash y : \texttt{bool}}}{y : \texttt{bool} \vdash \texttt{return}\ y : \texttt{bool}\ !\ \emptyset}$$

For the operation clause, we use TyITE since it is an `if/then/else` statement:

$$(\mathcal{T}_2)\ \textsc{TyITE}\ \frac{\mathcal{T}_3 \qquad \mathcal{T}_4 \qquad \mathcal{T}_5}{\Gamma_1 \vdash \texttt{if}\ i\ \texttt{then}\ k\ \texttt{true}\ \texttt{else}\ k\ \texttt{false} : \texttt{bool}\ !\ \emptyset}$$

The environment Γ_1 is equal to $\{i : \texttt{bool},\ k : \texttt{bool} \to \texttt{bool}\ !\ \emptyset\}$. There are three subderivations, for the three components of the `if/then/else`. The first derivation tree $\mathcal{T}_3$ type-checks the condition:

$$(\mathcal{T}_3)\ \textsc{TyVar}\ \frac{i : \texttt{bool} \in \Gamma_1}{\Gamma_1 \vdash i : \texttt{bool}}$$

The tree $\mathcal{T}_4$ type-checks the first branch of the conditional:

$$(\mathcal{T}_4)\ \textsc{TyApp}\ \frac{\textsc{TyVar}\ \dfrac{k \in \Gamma_1}{\Gamma_1 \vdash k : \texttt{bool} \to \texttt{bool} \,!\, \emptyset} \qquad \textsc{TyConst}\ \dfrac{\texttt{false} : \texttt{bool} \in \Sigma}{\Gamma_1 \vdash \texttt{false} : \texttt{bool}}}{\Gamma_1 \vdash k\ \texttt{false} : \texttt{bool} \,!\, \emptyset}$$

The use of Rule TyVar type-checks the continuation function k which is present in Γ_1. The continuation function is applied to `false` which is type-checked using TyConst. Then both of the derivations are combined using the TyApp rule.

The tree $\mathcal{T}_5$, which is similar to $\mathcal{T}_4$, type-checks the other branch of conditional:

$$(\mathcal{T}_5)\ \textsc{TyApp}\ \frac{\textsc{TyVar}\ \dfrac{k \in \Gamma_1}{\Gamma_1 \vdash k : \texttt{bool} \to \texttt{bool} \,!\, \emptyset} \qquad \textsc{TyConst}\ \dfrac{\texttt{true} : \texttt{bool} \in \Sigma}{\Gamma_1 \vdash \texttt{true} : \texttt{bool}}}{\Gamma_1 \vdash k\ \texttt{true} : \texttt{bool} \,!\, \emptyset}$$

The tree $\mathcal{T}_b$ type-checks the `do` computation by using the TyDo rule:

$$(\mathcal{T}_b)\ \textsc{TyDo}\ \frac{\mathcal{T}_6 \qquad \mathcal{T}_7}{\emptyset \vdash \texttt{do}\ x \leftarrow (\#\mathit{flip}\ \texttt{true}); (\#\mathit{flip}\ x) : \texttt{bool} \,!\, \{\mathit{flip}\}}$$

This rule type-checks the two computations in the `do` computation. The trees $\mathcal{T}_6$ and $\mathcal{T}_7$ show the derivations respectively. We use the rule TyOp to type checking the first computation since it is an effect call.

$$(\mathcal{T}_6)\ \textsc{TyOp}\ \frac{\mathit{flip} : \texttt{bool} \to \texttt{bool} \in \Sigma \qquad \textsc{TyConst}\ \dfrac{\texttt{true} : \texttt{bool} \in \Sigma}{\emptyset \vdash \texttt{true} : \texttt{bool}}}{\emptyset \vdash (\#\mathit{flip}\ \texttt{true}) : \texttt{bool} \,!\, \{\mathit{flip}\}}$$

The second computation is also an effect call. We use TyOp again to type check it.

$$(\mathcal{T}_7)\ \textsc{TyOp}\ \frac{\mathit{flip} : \texttt{bool} \to \texttt{bool} \in \Sigma \qquad x : \texttt{bool} \vdash x : \texttt{bool}}{x : \texttt{bool} \vdash (\#\mathit{flip}\ x) : \texttt{bool} \,!\, \{\mathit{flip}\}}$$

2.3.3 Operational Semantics

Figure 2.4 defines the big-step operational semantics of Effy. The judgement $c \Downarrow r$ states that computation c reduces to result r. A result is either a returned value, `return` v, or an unhandled operation, $\texttt{Op}\, v\, (y.c)$, where v is the operation's parameter and $y.c$ is its continuation.

$$\text{result } r \ ::= \ \texttt{return}\ v \ \mid \ \texttt{Op}\, v\, (y.\, c)$$

Evaluation

$$\boxed{c \Downarrow r}$$

$$\textsc{Eval-App}\ \frac{c[v/x] \Downarrow r}{(\texttt{fun}\ x \mapsto c)\, v \Downarrow r} \qquad \textsc{Eval-LetRec}\ \frac{c_2[(\texttt{fun}\ x \mapsto \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2/f] \Downarrow r}{\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 \Downarrow r}$$

$$\textsc{Eval-Ret}\ \frac{}{\texttt{return}\ v \Downarrow \texttt{return}\ v} \qquad \textsc{Eval-Op}\ \frac{}{\texttt{Op}\ v\ (y.c) \Downarrow \texttt{Op}\ v\ (y.c)}$$

$$\textsc{Eval-Do-Ret}\ \frac{c_1 \Downarrow \texttt{return}\ v \qquad c_2[v/x] \Downarrow r}{\texttt{do}\ x \leftarrow c_1; c_2 \Downarrow r} \qquad \textsc{Eval-Do-Op}\ \frac{c_1 \Downarrow \texttt{Op}\, v\, (y.\, c_1')}{\texttt{do}\ x \leftarrow c_1; c_2 \Downarrow \texttt{Op}\, v\, (y.\, \texttt{do}\ x \leftarrow c_1'; c_2)}$$

$$\textsc{Eval-With-Ret}\ \frac{c \Downarrow \texttt{return}\ v \qquad c_r[v/x] \Downarrow r}{\texttt{handle}\ c\ \texttt{with}\ h \Downarrow r}$$

$$\textsc{Eval-With-Handled-Op}\ \frac{c \Downarrow \texttt{Op}\, v\, (y.\, c') \qquad c_{\texttt{Op}}[v/x, (\texttt{fun}\ y \mapsto \texttt{handle}\ c'\ \texttt{with}\ h)/k] \Downarrow r}{\texttt{handle}\ c\ \texttt{with}\ h \Downarrow r}$$

$$\textsc{Eval-With-Unhandled-Op}\ \frac{c \Downarrow \texttt{Op}'\, v\, (y.\, c') \qquad \texttt{Op}' \notin \mathcal{O}}{\texttt{handle}\ c\ \texttt{with}\ h \Downarrow \texttt{Op}'\, v\, (y.\, \texttt{handle}\ c'\ \texttt{with}\ h)} \qquad \textsc{Eval-If-True}\ \frac{c_1 \Downarrow r}{\texttt{if true then}\ c_1\ \texttt{else}\ c_2 \Downarrow r}$$

$$\textsc{Eval-If-False}\ \frac{c_2 \Downarrow r}{\texttt{if false then}\ c_1\ \texttt{else}\ c_2 \Downarrow r}$$

Figure 2.4: Operational semantics of Effy

Rule Eval-App substitutes the lambda binder x with the value v in the lambda computation. Next, the value result is generated by the `return` v computation (Eval-Ret), while the unhandled operation just reduces to itself in the computation (Eval-Op). If the intermediate result of a sequential `do` is a value (Eval-Do-Ret), it is substituted into the second computation. If it is an unhandled operation (Eval-Do-Op), the second computation is appended to its continuation.

In the last three rules, $h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$. When a handled computation evaluates to a value (Eval-With-Ret), this value is substituted into the handler's return case. Finally, an unhandled operation is passed to the appropriate operation case, if there is one (Eval-With-Handled-Op), or propagated further, if there is not (Eval-With-Unhandled-Op). In either case, the continuation $y.c'$ is handled by the same handler. This is because Eff and Effy feature a *deep handlers* semantics, which means that the handler is re-wrapped around the continuation as we showed in the basic example in Section 2.2. Rules Eval-If-True and Eval-If-False reduce to the then-branch if the condition is `true` and to the else-branch if the condition is `false`.

2.4 Related Work

In this section, we present an overview of the work related to algebraic effects and handlers. We start with the different related calculi and optimisation techniques. Then, we show different systems from the literature, and we discuss the difference between their work and ours.

2.4.1 Related Calculi

Related type-and-effect systems The calculus presented in this chapter resembles closely that of Pretnar [71] for inferring algebraic effects for Eff. The type system is inspired by his work. Pretnar goes further into discussing a type inference algorithm with constraint solving algorithm that we modify and build on in Chapter 4.

In the work of Bauer and Pretnar [5], the presentation of an effect system that gives information of what effects might be triggered by a computation opened the gate for better optimisations in order to eliminate the overhead that effects and handlers generate. The optimisations we show later in Chapter 3 depend heavily on the effect system we introduce to be able to get more efficient run-times.

Several type-and-effects systems with various inference algorithms have been discussed in the literature. The work of Dolan and Mycroft uses effect sub-typing

accompanied by a unification-based algorithm for inference [25]. Different systems that are based on the notion of row-polymorphism [30] such as the work of Leijen on *Koka* [49, 51]. Lindley et al. used implicit effect polymorphism in Frank [55]. Also, the work of Faes explored the extension of *algebraic subtyping* [23] into effect systems [28].

Different Calculi for Algebraic Effects Kammar and Pretnar [43] presented a calculus for algebraic effects and handlers with Hindley-Milner-style polymorphism. Our work, contrarily, depends heavily on System F-style polymorphism [62, Chapter 23]. The Hindley-Milner style allows universal quantifiers only at the top-level of types. In System F however, they are allowed anywhere in a type. This will be used in Chapter 4.

In our work, we deal with deep handlers, which means that the handler is re-wrapped around the continuation in order to handle any other effect that might arise in the continuation. Kammar et al. [41] introduced *shallow* handlers; these only handle the first occurrence of an effect in a computation and are not wrapped around the continuation. This means that a new handler is needed to handle the effects in the continuation. Hillerström and Lindley [34] also introduced a calculus and an implementation for shallow handlers that uses CPS translation.

Leijen [51] introduces an effect system for algebraic effects and handlers based on types with scoped labels to track effects. Hillerström and Lindley [33] also presented a calculus for algebraic effects based on row-typing and provided operational semantics for it.

The recent work of Biernacki et al. [7] introduces Helium, which is language that supports abstraction over algebraic effects. Helium's calculus is equipped with a row-based polymorphic type-and-effect system that supports existential quantification over both types of an effect.

Calculi with Explicit Subtyping So far, in this thesis, we talk about Effy's type-system with implicit subtyping. We will see later that implicit subtyping introduces bugs when we try to optimise Effy's code using source to source transformations. This is because we can not precisely pinpoint the type of a given term when we run the optimisations.

Other than implicit subtyping, the notion of explicit subtyping has also been discussed in the literature. We expand the calculus with notions of subtyping coercions that accompany the terms to tell us the exact type of a given term without depending on implicit subtyping. In the literature, the work of Mitchell [56] introduced the idea

of inserting coercions during type inference for ML-based languages, as a means for explicit casting between different numeric types.

Breazu-Tanne et al. [11] also present a translation of polymorphic languages into System F [62, Chapter 23], extended with coercions. System F_C [88] uses explicit type-equality coercions to encode complex language features (e.g. GADTs [61] or type families [83]).

2.4.2 Effect Handlers Implementations

Many systems implement effect handlers, employing a range of different implementation techniques. Almost every calculus we have discussed above features an implementation as a proof of concept. Some toy languages were developed in order to discover a property of algebraic effects or to show that one translation of effects and handlers is more efficient than another.

As we will see later, Effy translates handlers to a free monad representation [3, 89] that can be executed by OCaml. While this translation works in principle, it is not very efficient.

Multicore OCaml [27] adapts the stack-switching design by Bruggeman et al. [12] to provide an efficient native implementation. Multicore OCaml provides support for algebraic effects in terms of multicore *fibers* to efficiently represent delimited continuations at runtime. These come both in a cheaper one-shot and more expensive multi-shot form. Several works [47, 35] have shown that this provides an effective compilation target for algebraic effects. The performance of Multicore OCaml is notable when continuations are called only once. However, when a continuation is called more than once, it needs to be cloned first; this degrades the performance considerably.

Kammar et al. [41] presented an implementation of effect handler libraries in Haskell, OCaml, SML and Racket. The implementation technique uses an abstract free-monad interface, implemented with continuations in Haskell and delimited continuations in the other languages.

Kiselyov and others [46, 91, 45] investigate a number of different implementations of the free monad that exhibit good runtime performance and/or algorithmic time complexity. They consider a library in the lazily evaluated language Haskell. They also support shallow handlers and explicit manipulation of the free monad structure in the source language.

The web programming languages Koka [51] and Link [35] use different strategies to compile effects and handlers. Koka performs a selective continuation passing style (CPS) translation to lift effectful code into a free monad representation. Links

implements handlers by using a CEK machine on the server side and uses of a higher-order CPS translation on the client-side [35].

Kammar and Plotkin [42] develop a theory of optimisations valid for effectful programs that satisfy a specific algebraic theory. For example, if we assume symmetry of non-deterministic choice, we may safely exchange the order in which non-deterministic computations are executed. This may result in speedups.

Recently, effects and effect handlers are getting into mainstream languages. Brachthäuser et al. [9] developed an implementation of effect handlers in *Java*. Their work provides a library for effect handlers in terms of delimited continuations. In Chapter 5, we provide an implementation for effects and handlers based on delimited continuations also, though in Prolog. Brachthäuser and Schuster implemented an algebraic effects and handlers library for Scala [8]. Leijen implemented a library for effect handlers in *C* [50].

Chapter 3

Optimised Compilation for Eff

This chapter presents our work on optimising the Eff compiler. We introduce our optimisation techniques and the evaluation on a set of benchmarks.

3.1 Motivation

Currently, there is a variety of implementations of algebraic effects and handlers available to use as we have seen in the previous chapter. Due to this diversity of implementations, runtime performance is becoming more of a concern. So far, most implementations come in the form of libraries [46, 47, 41, 10] and interpreters [35, 6]. As a consequence, much of the effort to improve performance has been directed toward improving the runtime representation of computations with handlers and associated operations [35, 27, 45]. However, we see that in practice effect handlers still incur a significant performance overhead compared to hand-written code and native side-effects.

With an end-to-end overview of a compiler for the Koka language, Leijen [51] has demonstrated that compilation is a valid alternative avenue for implementing algebraic effects and handlers. This showed that *optimising* compilation, in particular, is interesting because it can further narrow the performance gap with hand-written code and native effects.

To substantiate this belief, we present in this chapter our approach to the optimised compilation of Eff. Currently, Eff is considered inefficient due to the use of a free monad representation in the compilation to OCaml, the disadvantages of which we are going to discuss later in this chapter. Our approach to optimise Eff

is interleaving three different techniques with the aim of eliminating the overhead introduced by handler code as much as possible. The first optimisation technique is term-rewriting rules that aim to partially evaluate the program during compile time. The second technique is purity-aware compilation which aims to compile non-effectful code using native OCaml code instead of free monad representation. Finally, function specialisation aims to aid the first two techniques exposing more opportunities to apply rewriting and purity aware compilation.

Chapter overview We start this chapter by showing the current state of Eff compilation by means of an example in Section 3.2. This describes the performance limitations of the current compilation techniques and also shows the potential optimisations possible to improve the efficiency of the output code. These potential optimisations are our approach to optimising Eff. The first technique we present is source-to-source transformations and selective function specialisation with the aim of eliminating explicit uses of handlers and the resulting overhead (Section 3.3). Next, we give a basic monadic elaboration of Effy into OCaml, where effectful computations of Effy are translated into pure computations of OCaml that generate elements of the free monad (Section 3.4). We further refine this translation into one that exploits the purity of computations to generate pure OCaml expressions where possible, resulting in mostly idiomatic OCaml code (Section 3.5). This stage crucially relies on the information from Eff's type-and-effect system to do its job. In the next sections, we discuss our implementation of the Eff compiler (Section 3.6), and present an experimental evaluation, which clearly demonstrates the effectiveness of this approach on a number of benchmarks. (Section 3.7). At the end of the chapter, we discuss the results of optimising Eff in Section 3.8.

Throughout this chapter, we work with both OCaml and Eff, whose syntax closely follows OCaml, and we use colours to distinguish between `OCaml code` and `Eff code`.

3.2 Compilation of Eff to OCaml

This section motivates the need for optimised compilation of Eff. We start this section by showing a small example in Eff and then go through the process of compiling it into OCaml using the Eff's compiler strategy. We discuss the issues of this compilation regarding the inefficiency of the produced code. Afterwards, we briefly show how the compilation can be more efficient using our optimisation techniques.

3.2.1 Programming with Algebraic Effect Handlers

The following example is a simple loop that repeatedly increments an implicit integer state. We first declare two effects to manipulate the state and then define a recursive function that increments the state a given number of times `n`.

```
effect Put: int -> unit;;
effect Get: unit -> int;;
let rec loop n =
  if n = 0
  then
    ()
  else
    (Put (Get () + 1); loop (n - 1))
```

The following handler gives the semantics of the two effects in the `loop` function:

```
let state_handler = handler
  | #Put s' k -> (fun _ -> k () s')
  | #Get () k -> (fun s -> k s s)
  | return _ -> (fun s -> s)
```

The `Put` case is defined in terms of the parameter `s'` (the new value of the state) and the continuation `k`. We handle `Put` as a function that accepts (though ignores) the initial state, and then resumes the continuation by passing it the expected result `() : unit`.

The case for `Get` is similar, except that we pass the initial state `s` to the continuation `k` twice: first as the result of the lookup, and second as the new (unmodified) state. Finally, the return case ignores the final result of a computation and instead returns the current value of the state. Note that by modifying the handled computation into a function, the handler changes the computation's type.

To tie everything together, we define the function `main`, which applies the handler to `loop` and provides the initial state `0` to the resulting function:

```
let main n =
  (with state_handler handle loop n) 0
```

3.2.2 Basic Compilation to OCaml

We represent an EFF computation by a (value of the) *free monad* in OCAML, which is implemented as the `'a computation` data type. It is used to build an abstract syntax tree based on EFF's syntax.

The data type `'a computation` represents computations that have effects. In other words, it represents terms with dirty types. For instance, in the `loop` example, the `'a computation` data type is defined as follows:

```
type 'a computation =
  | Value of 'a
  | Put of int * (unit -> 'a computation)
  | Get of unit * (int -> 'a computation)
```

The `Value` constructor represents non-effectful results using the data type `'a`. The second and third constructors represent the information of the two effects in our program.

We then build computations using the following constructors: a value embedding `return : 'a -> 'a computation` or basic operations `put : int -> unit computation` and `get : unit -> int computation`. We compose effectful computations using the monadic bind `>>=` operator, which evaluates an effectful computation of type `'a computation` and passes its result to a continuation of type `'a -> 'b computation`, resulting in a `'b computation`. We postpone the discussion of the generic implementation of this data type to Section 3.4.

EFF functions have bodies that can trigger effects. For that reason, an EFF function of type `'a -> 'b` translates into a function of type `'a -> 'b computation` in OCAML. This applies equally to (curried) multi-argument functions. Thus an EFF function `f : 'a -> 'b -> 'c` is translated as: `f : 'a -> ('b -> 'c computation )computation`. The application `f x y` translates to `f x >>= fun g -> g y`.

Using the above definitions, the presented compiler translates the `loop` function into the following code:

```
let rec loop n =
  equal n >>= fun f ->
  f 0     >>= fun b ->
  if b then return () else
    Get ()  >>= fun s  ->
    plus s  >>= fun g  ->
    g 1     >>= fun s' ->
    Put s'  >>= fun _  ->
    minus n >>= fun h  ->
    h 1     >>= fun n' ->
    loop n'
```

where `equal`, `plus` and `minus` are translations of EFF's arithmetic operations into predefined OCAML constants of the appropriate function type. For example, we define addition as follows:

```
let plus = fun x -> return (fun y -> return (x + y))
```

We translate `state_handler` to a record of functions having each of the effect clauses and the value as the return case of the handler, yielding the result shown below

```
let state_handler = handler {
  put_case = (fun s' k ->
              return (fun _ -> k () >>= fun f -> f s'));
  get_case = (fun () k ->
              return (fun s -> k s >>= fun f -> f s));
  return_case = (fun _ ->
                  return (fun s -> return s));
}
```

The record specifying handler cases are of a predefined record type:

```
type ('a, 'b) handler_cases = {
  put_case : int ->
              (unit -> 'b computation) -> 'b computation;
  get_case : unit ->
              (int -> 'b computation) -> 'b computation;
  return_case : 'a -> 'b computation;
}
```

The handler is represented as the function `handler : ('a, 'b)handler_cases -> ('a computation -> 'b computation)` that takes handler cases and returns a handler, which is represented as a function between computations. Finally, the `main` function may be translated as

```
let main n =
  state_handler (loop n) >>= (fun f -> f 0)
```

Drawbacks of basic compilation When running the compiled OCaml program, the continual composition and decomposition of computations produce a substantial overhead. However, this compilation is safe as it assumes that any computation in the program can be effectful. Therefore, if an effect is triggered, it will be caught. This comes at the cost of performance.

3.2.3 Purity Aware Compilation

The idea behind this optimisation is that by identifying pure non-effectful computations and generating regular OCaml code for them, as proposed by Leijen [51], we can avoid some of the overhead created by the free monad. For instance, since the arithmetic operators used in `loop` are pure,we can translate them

directly into OCaml's arithmetic operations and bind their result with `let` rather than with the more expensive `>>=` operator:

```
let rec loop n =
  let f = (=) n in
  let b = f 0 in
  if b then return () else
    Get () >>= fun s  ->
    let g = plus s in
    let s' = g 1 in
    Put s' >>= fun _  ->
    let h = minus n in
    let n' = h 1 in
    loop n'
```

Nevertheless, the backbone of the computation still makes use of `>>=` to sequence the effectful `Get` and `Put` operations. Hence, the overall impact of this optimisation on this example is limited.

In general, purity aware compilation effectiveness is directly proportional with the percentage of the effect-free code used in a program. Applying a strong effect system that gets information about effects in terms can benefit the overall performance by applying the purity aware-compilation technique during compilation.

3.2.4 Optimising Compilation

The previous strategy leaves room for more optimisations. We achieve that with the help of more aggressively optimised compilation, which replaces operation calls in a handled operation with their corresponding operation cases. In addition to partial evaluations, beta reduction techniques and specialisations of functions, we can obtain altogether more efficient and tighter code:

```
let main n = (
  let rec state_handler_loop m =
    if m = 0
    then
      (fun s -> s)
    else
      (fun s -> state_handler_loop (m - 1) (s + 1))
  in
  state_handler_loop n) 0
```

All of the handler, the explicit operations `Get` and `Put`, and their explicit sequencing with `>>=` have been eliminated. The recursive function `loop` has been locally specialised for the particular interpretation of `state_handler`. The resulting code is

very close to the hand-written equivalent, the only difference being that a human programmer would hoist the abstractions out of the two branches of the conditional expression.

3.3 Source-Level Optimisations

This section discusses the heart of our optimising compiler, which consists of source-to-source transformations that aim to improve the runtime performance of the program. We look at term rewriting rules that aim to eliminate most of the overhead introduced by effects and handlers. We also interleave the rewriting rules with function specialisation that aims to access the function's body to apply the rewriting rules for a more efficient output code.

3.3.1 Term Rewriting Rules

Equivalences and Partial Evaluation

To facilitate equational reasoning [65] about programs with algebraic effect handlers, Bauer & Pretnar [5] establish several observational equivalences, which we adapt to Effy and they are shown in Figure 3.1 (we omit structural equivalences that make $\equiv$ a congruence). The meta-variables v, c stand for arbitrary values and computations that make the left- and right-hand sides of the equivalences well-typed with the same type. In the last three rules, h is of the form of $\{\texttt{return}\ x \mapsto c_r, [\mathsf{Op}\ x\ k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in \mathcal{O}}\}$.

Rules 3.1 up to 3.5 are beta-reductions following the rules in Figure 2.4. Rule 3.6 is the associativity rule pushing the sequencing outside. We assume in this rule that free variables are unique, hence there is no variable shadowing. The last four rules are handler partial evaluation rules. These equivalences aim to expose handlers for the next part that mainly tries to reduce the exposed handlers to normal OCaml computations.

Simplification and Handler Reduction

We use the information of the type and effect system and the syntactic structure of the terms to perform a number of optimisations. We mainly target optimisations that (help) remove handlers. We denote these optimisations in terms of semantics-preserving rewrite rules of the form $c \rightsquigarrow c'$ and we try to expose opportunities to eliminate the handlers using the term rewriting rules that can also be seen as partial

$$(\texttt{fun}\ x \mapsto c)\, v \equiv c[v/x] \tag{3.1}$$

$$\texttt{fun}\ x \mapsto v\, x \equiv v \tag{3.2}$$

$$\texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_2 \equiv c_2[(\texttt{fun}\ x \mapsto \texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_1)/f] \tag{3.3}$$

$$\texttt{do}\ x \leftarrow \texttt{return}\ v; c \equiv c[v/x] \tag{3.4}$$

$$\texttt{do}\ x \leftarrow c; \texttt{return}\ x \equiv c \tag{3.5}$$

$$\texttt{do}\ x_2 \leftarrow (\texttt{do}\ x_1 \leftarrow c_1; c_2); c_3 \equiv \texttt{do}\ x_1 \leftarrow c_1; (\texttt{do}\ x_2 \leftarrow c_2; c_3) \tag{3.6}$$

$$\texttt{handle}\ c_1\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_2\} \equiv \texttt{do}\ x \leftarrow c_1; c_2 \tag{3.7}$$

$$\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \equiv c_r[v/x] \tag{3.8}$$

$$\texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{Op}\, v; c)\ \texttt{with}\ h \equiv c_{\texttt{Op}}[v/x, (\texttt{fun}\ y \mapsto \texttt{handle}\ c\ \texttt{with}\ h)/k] \tag{3.9}$$

$$\texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{Op}'\, v; c)\ \texttt{with}\ h \equiv \texttt{do}\ y \leftarrow \texttt{Op}'\, v; \texttt{handle}\ c\ \texttt{with}\ h \quad (\texttt{Op}' \notin \mathcal{O}) \tag{3.10}$$

Figure 3.1: Basic Equivalences

evaluation rules which are listed in Figure 3.2. The rewriting rules are divided into two groups:

Simplification These three rules simplify the structure of the program in the hope of exposing opportunities to eliminate handlers. The first two, App-Fun and Do-Ret, are static (during compile time) counterparts of the semantic rules Eval-App and Eval-Do-Ret of Figure 2.4. It is a straightforward partial evaluation of the computations. The last rule, Do-Op, exploits the associativity of do to bring an operation to the front, where it can potentially be reduced by a handler.

Handler Reduction These rules reduce terms of the form (handle c with h) for different shapes of computation c and are the heart of our optimisation. Rule With-LetRec moves the handler into the main sub-computation. Rules With-Ret and With-Handled-Op are the static counterparts of the semantic rules Eval-With-Ret and Eval-With-Handled-Op, respectively.

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

Simplification

$$\textsc{App-Fun}\ \frac{}{(\texttt{fun}\ x \mapsto c)\ v \rightsquigarrow c[v/x]} \qquad \textsc{Do-Ret}\ \frac{}{\texttt{do}\ x \leftarrow \texttt{return}\ v; c \rightsquigarrow c[v/x]}$$

$$\textsc{Do-Op}\ \frac{}{\texttt{do}\ x \leftarrow (\texttt{do}\ y \leftarrow \texttt{Op}\, v; c_1); c_2 \quad \rightsquigarrow \quad \texttt{do}\ y \leftarrow \texttt{Op}\, v; (\texttt{do}\ x \leftarrow c_1; c_2)}$$

Handler Reduction

$$\textsc{With-LetRec}\ \frac{}{\begin{array}{l}\texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ v \rightsquigarrow \\ \quad \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ v)\end{array}}$$

$$\textsc{With-Ret}\ \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \rightsquigarrow c_r[v/x]}$$

$$\textsc{With-Handled-Op}\ \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ (\texttt{Op}\, v)\ \texttt{with}\ h \rightsquigarrow c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto c_r)/k]}$$

$$\textsc{With-Pure}\ \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} \qquad \Gamma \vdash c : A\ !\ \Delta \qquad \Delta \cap \mathcal{O} = \emptyset}{\texttt{handle}\ c\ \texttt{with}\ h \rightsquigarrow \texttt{do}\ x \leftarrow c; c_r}$$

$$\textsc{With-Do}\ \frac{\begin{array}{c}h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} \\ h' = \{\texttt{return}\ y \mapsto (\texttt{handle}\ c_2\ \texttt{with}\ h), [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}\end{array}}{\texttt{handle}\ (\texttt{do}\ y \leftarrow c_1; c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c_1\ \texttt{with}\ h'}$$

Figure 3.2: Term Rewriting Rules

Next, rule WITH-PURE applies when the computation c is pure relative to the handler h. This is the case when the intersection of the operations that may be called by c with the operations handled by h is empty. In this case, only the handler's `return` case is relevant. Hence, we sequence it at the end of c using the sequencing operator.

Finally, the most unusual rule is WITH-DO which reduces the handling of a sequence of two computations to a form where the two computations are handled separately. The validity of this transformation becomes more obvious when we split it into two steps:

1. We replace the sequential `do` with a handler:

$$\texttt{do}\ y \leftarrow c_1; c_2 \rightsquigarrow \texttt{handle}\ c_1\ \texttt{with}\ \{\texttt{return}\ y \mapsto c_2\}$$

   The intuition is that both forms express that the value returned by c_1 gets bound to y in c_2.

2. We change the association of the handlers:

$$\begin{gathered}\texttt{handle}\ (\texttt{handle}\ c_1 \\ \texttt{with}\ \{\texttt{return}\ y \mapsto c_2\}) \\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} \\ \rightsquigarrow \\ \texttt{handle}\ c_1 \\ \texttt{with}\ \{\texttt{return}\ y \mapsto (\texttt{handle}\ c_2 \\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}\end{gathered}$$

   The intuition is that any operations generated by c_1 are forwarded anyway from the inner handler to the outer handler. However, any return case is first handled by the inner handler, and the computation that results is further processed by the outer handler. The rewritten form accomplishes the same workflow with a single handler around c_1. The hope is that the handler around c_2 can be specialised independently from specialising the handler around c_1.

Soundness The rewrite rules preserve observational equivalence. That is, if a term is transformed according to the rules of Fig. 3.1, the resulting term preserves the behaviour of the original, despite their syntactical difference. In order to prove this property, we make use of the following induction principle [5, 68] which algebraic effects and handlers comply to: to show that a property ϕ holds for all computations $\Gamma \vdash c : A \,!\, \Delta$, it suffices to show that 1) $\phi(\texttt{return}\ v)$ holds for all values v, and 2) that $\phi(\texttt{do}\ x \leftarrow \texttt{Op}\, v; c')$ holds for all operations $\texttt{Op} \in \Delta$, values v and computations c', given that $\phi(c')$ holds.

The equivalences and induction principle will turn out to be useful in proving the soundness of the rewriting-rules:

Theorem 1.

$$\forall c, c', \Gamma, A, \Delta : \quad \Gamma \vdash c : A \,!\, \Delta \;\wedge\; c \rightsquigarrow c' \quad \Rightarrow \quad c \equiv c'$$

The proof of this theorem is given in Appendix A.1.

3.3.2 Function Specialisation

The rewrite rules in Figure 3.2 deal with most computations of the form $(\texttt{handle}\ c\ \texttt{with}\ h)$ where h is a handler expression, either dropping the handler altogether or pushing it down in the sub-computations. However, one important case is not dealt with: the case where c is of the form $f\,v$ with f the name of a user-defined recursive function. However, if f is a function parameter of a higher-order function, we can not perform the optimisation.

Consider this small example of the above situation:

```
let rec go n = go (Next n) in
handle (go 0) with
| return x -> x
| #Next n k -> if n > 100 then n else k (n * n + 1)
```

The non-terminating recursive function `go` seems to diverge. Yet, with the provided handler, its argument steadily increases and evaluation eventually terminates when the argument exceeds `100`.

In order to optimise this situation, we create a specialised copy of the function that has the handler pushed into its body. In other words, for any recursive definition $\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$, we perform the following general rewrite inside c:

$$\texttt{handle}\ f\ v\ \texttt{with}\ h \quad \rightsquigarrow \quad \texttt{let rec}\ f'\ x = \texttt{handle}\ c_f\ \texttt{with}\ h\ \texttt{in}\ f'\ v$$

The expectation is that, by exposing the handler to the body of the function (c_f), further optimisations succeed in eliminating the explicit handler. A critical step involved in the post-processing is to *"tie the knot"*: after several rewrite steps in c_f, the handler is applied to the (original) recursive call, so we have a term of the form $\texttt{handle}\ f\ v'\ \texttt{with}\ h$, which we can replace by $f'\,v'$. This eliminates the handler entirely and turns the original example into

```
let rec go n = ... in
let rec go' n = if n > 100 then n else go' (n * n + 1) in
go' 0
```

Generalization to non-tail recursion The above basic specialisation strategy only works when the recursive call has a tail position. Yet, that is often not the case. Take for instance the following example.

```
let rec range n =
  match n with
  | 0 -> []
  | _ -> Fetch () :: range (n - 1)
in
handle (range 5) with
| return x  -> x
| #Fetch _ k -> k 42
```

The function `range` creates a list of given length, filling it with elements obtained by the `Fetch` operation. To keep the example small we use a handler that always yields the value `42`.

With the basic specialisation strategy, further optimisation does not succeed in tying the knot. Instead, we obtain this partially optimised form:

```
let rec range n = ... in
let rec range' n =
  match n with
  | 0 -> []
  | _ -> handle (range (n - 1)) with
         | return x  -> 42 :: x
         | #Fetch _ k -> k 42
in
range' 5
```

In the tail position, the rewrite rule With-Do has kicked in to pull the call's continuation into the `return` case of the handler. This has turned the call into a tail call, but the `return` case of the new handler around this call differs from the original handler's return case. This prevents us from tying the knot.

We could create a second specialised function definition for this new handler, but the same problem would arise at its recursive call and so on, yielding an infinite sequence of specialised functions. Instead, we use generalisation to break out of this diverging process. Instead of specialising the function for one specific handler in this diverging sequence, we specialise it for what they all have in common (the operation cases) and parametrise it in what is different (the return case).

This yields the following general rewrite rule: for any recursive definition $\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$, we perform the following general rewrite inside c:

$$\texttt{handle}\ f\ v\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}$$

$$\rightsquigarrow$$

$$\texttt{let rec}\ f'\ (x, k) =$$

$$\texttt{handle}\ c_f\ \texttt{with}\ \{\texttt{return}\ x \mapsto k\ x, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}\ \texttt{in}\ f'\ (v, \texttt{fun}\ x \mapsto c_r)$$

and replace each handled recursive call $\texttt{handle}\ f\ v'\ \texttt{with}\ \{\texttt{return}\ x \mapsto c'_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}$ with $f'\ (v', \texttt{fun}\ x \mapsto c'_r)$.

This strategy enables us to tie the knot in the `range` example and obtain the following form:

```
let rec range' (n, k) =
  match n with
  | 0 -> k []
  | _ -> range' (n - 1, fun x -> k (42 :: x))
in
range' (5, fun x -> x)
```

Note that in effect this approach selectively CPS-transforms recursive functions to specialise them for a particular handler.

Termination If left unchecked, function specialisation can diverge. This is illustrated by the following small example program:

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in handle (go m) with
   | #Decide _ k ->
       handle k true with
       | #Fail _ _ -> k false
```

After specialisation for the top-level handler, we obtain

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in let rec go1 n1 =
     if n1 = 0 then Fail
       else handle go1 (n1-1) with
```

```
            | #Fail -> go1 (n1-2)
    in go1 m
```

Note that the specialised function `go1` still contains the second handler, which is now applied to a recursive call. Hence, we can continue by specialising this handled call to obtain

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in let rec go1 n1 =
     if n1 = 0 then Fail
       else let rec go2 n2 =
              if n2 = 0 then go1 (n1-2)
                else handle (handle go1 (n2-1) with
                               | #Fail -> go1 (n2-2)) with
                     | #Fail -> go1 (n1-2)
            in go2 (n1-1)
   in go1 m
```

Now the resulting code contains two nested handlers around a recursive call. However, the inner of those two handlers is distinct from any of the previous handlers because it refers to the new variable `n2`. Hence, we can specialise again and again without end. This non-termination is undesirable, and so we currently enforce termination by not re-specialising any already specialised function.

3.4 Basic Translation of Effy to OCaml

3.4.1 Translating Types

As a concrete target for translating Effy, we pick a small subset of OCaml that includes standard constructs such as booleans, integers, functions and local definitions (both non-recursive and recursive). Its types are given in Figure 3.3, and in addition to the standard ones, they include a predefined type T `computation`, which represents computations returning values of type T, and a type (T_1, T_2) `handler_cases`, which lists all cases of a handler that takes computations of type T_1 `computation` into T_2 `computation`.

We translate types of Effy into OCaml by means of the compilation function $[\![\cdot]\!]$ listed in Figure 3.4. Primitive types and function types are straightforwardly mapped onto their OCaml counterparts. The handler type is translated to a function type that turns one type of computation into another.

$$
\begin{array}{rcl}
\text{type } T & ::= & \texttt{bool} \\
& | & \texttt{int} \\
& | & T_1 \to T_2 \\
& | & T \texttt{ computation} \\
& | & (T_1, T_2) \texttt{ handler\_cases}
\end{array}
$$

Figure 3.3: Types of (a subset of) OCAML

$$
\begin{array}{c}
[\![\texttt{bool}]\!] = \texttt{bool} \\
[\![\texttt{int}]\!] = \texttt{int} \\
[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!] \\
[\![\underline{C} \Rrightarrow \underline{D}]\!] = [\![\underline{C}]\!] \to [\![\underline{D}]\!] \\
[\![A \,!\, \Delta]\!] = [\![A]\!] \texttt{ computation}
\end{array}
$$

Figure 3.4: Compilation of EFFY types to OCAML

Computation types are mapped to the predefined `computation` type, defined by default as a datatype representation of a free monad (where $\texttt{Op}_i : A_i \to B_i$ ranges over the signature of all EFFY operations):

```
type 'a computation =
| Return: 'a -> 'a computation
| Op1: [[A1]] * ([[B1]] -> 'a computation) -> 'a computation
| ...
| Opn: [[An]] * ([[Bn]] -> 'a computation) -> 'a computation
```

Here `Return x` represents a value `x` as a (pure) computation, and `Op x k` denotes an impure computation that calls operation `Op` with argument `x` and continuation `k`. We can interchange the implementation of `'a computation` to obtain different runtime representations, though in this work, we stick to the free monad representation, as it is both simple and efficient enough for our purposes.

Note that the translation erases the dirt Δ from computation types $A \,!\, \Delta$, for lack of a convenient way to represent it in OCAML. The algebraic effect handlers implementation of Multicore OCAML [27] has made a similar choice not to reflect the set of possible operations in the type.

Finally, the type of all handler cases is defined to be the record type:

```
type ('a, 'b) handler_cases = {
  return: 'a -> 'b;
  op1: [[A1]] -> ([[B1]] -> 'b) -> 'b;
```

```
  ...
  op_n: [[A_n]] -> ([[B_n]] -> 'b) -> 'b
}
```

Here the operation cases are represented by a function that takes the argument and the continuation of the operation and then performs the operation's corresponding behaviour. (Note that the domain `'b` is not necessarily of the form `_ computation`. We exploit this fact in Section 3.5 to allow handlers that handle all computations into a pure value.)

3.4.2 Translating Terms

OCaml terms, given in Figure 3.5, include the standard ones: variables, constants (corresponding to ones in Effy), function abstractions & applications, both non-recursive & recursive local definitions, and conditionals. Next, we include records that list handler cases and a number of predefined functions for value embedding, operations, handler definitions and sequencing (or in other words, monadic binds). Both Effy values and computations are translated to OCaml expressions as described in Figure 3.6. With several notable exceptions, most forms have a direct counterpart in OCaml. The fourth rule in that figure translates handler clauses where the translated operation clauses are of the form $\mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n$. We expand E_i as follows:

$$E_i = \begin{cases} \mathtt{fun}\ x\ k \mapsto [\![c_{\mathtt{Op}_i}]\!] & \mathtt{Op}_i \in \mathcal{O} \\ \mathtt{fun}\ x\ k \mapsto \mathtt{op}_i\ x >\!\!>\!\!= k & \text{otherwise} \end{cases}$$

A handler value is translated to an application of the `handler` function to a record value that gathers the return and operation cases. For the default free monad representation, `handler` is defined as follows:

```
let rec handler (h : ('a, 'b) handler_cases) : ('a
    computation -> 'b) =
  function
  | Return x -> h.return x
  | Op1 (x, k) -> h.op1 x (fun y -> handler h (k y))
  ...
  | Op_n (x, k) -> h.op_n x (fun y -> handler h (k y))
```

In case a handler does not provide a handling case for an operation $\mathtt{Op}_i$, we fill it in with a default case that propagates it outwards, in which case `'b` needs to be of the form `_ computation`. Note that this is always the case with the basic translation presented in this section.

$$
\begin{array}{rcl}
\text{expression } E & ::= & x \\
& | & \mathtt{k} \\
& | & \mathtt{fun}\ x \mapsto E \\
& | & E_1\ E_2 \\
& | & \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 \\
& | & \mathtt{let\ rec}\ f\ x = E_1\ \mathtt{in}\ E_2 \\
& | & \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3 \\
& | & \{\mathtt{return} = E; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} \\
& | & \mathtt{return}\ E_1 \\
& | & \mathtt{op}_1\ |\ \cdots\ |\ \mathtt{op}_n \\
& | & \mathtt{handler} \\
& | & >\!\!>\!\!=
\end{array}
$$

Figure 3.5: Terms of (a subset of) OCaml

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\mathtt{k}]\!] & = & \mathtt{k} \\
[\![\mathtt{fun}\ x \mapsto c]\!] & = & \mathtt{fun}\ x \mapsto [\![c]\!] \\
[\![\{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ x\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\}]\!] & = & \begin{array}{l} \mathtt{handler} \\ \{\mathtt{return} = \\ \mathtt{fun}\ x \mapsto [\![c_r]\!]; \\ \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} \end{array} \\
[\![v_1\ v_2]\!] & = & [\![v_1]\!]\ [\![v_2]\!] \\
[\![\mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2]\!] & = & \mathtt{let\ rec}\ f\ x = [\![c_1]\!]\ \mathtt{in}\ [\![c_2]\!] \\
[\![\mathtt{return}\ v]\!] & = & \mathtt{return}\ [\![v]\!] \\
[\![\mathtt{Op}_i\ v]\!] & = & \mathtt{op}_i\ [\![v]\!] \\
[\![\mathtt{do}\ x \leftarrow c_1; c_2]\!] & = & [\![c_1]\!] >\!\!>\!\!= (\mathtt{fun}\ x \mapsto [\![c_2]\!]) \\
[\![\mathtt{handle}\ c\ \mathtt{with}\ v]\!] & = & [\![v]\!]\ [\![c]\!] \\
[\![\mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!] & = & \mathtt{if}\ [\![v]\!]\ \mathtt{then}\ [\![c_1]\!]\ \mathtt{else}\ [\![c_2]\!]
\end{array}
$$

Figure 3.6: Compilation of Effy terms to OCaml

Building a computation from a value, from an operation or by binding two computations together all happens in terms of the corresponding operations on the underlying free monad representation.

```
let return (x : 'a) : 'a computation = Return x

let op_1 (x : [[A_1]]) (k : [[B_1]] -> 'a computation) : 'a
    computation = Op_1 (x, k)
...
let op_n (x : [[A_n]]) (k : [[B_n]] -> 'a computation) : 'a
    computation = Op_n (x, k)

let rec (>>=) (c : 'a computation) (f : 'a -> 'b
    computation) : 'b computation =
  match c with
  | Return x -> f x
  | Op_1 (x, k) -> Op_1 (x, (fun y -> (k y) >>= f))
  ...
  | Op_n (x, k) -> Op_n (x, (fun y -> (k y) >>= f))
```

Finally, handling a computation with a handler is simply translated as applying the handler to the computation. The following theorem shows that this translation is type-preserving.

Theorem 2.

$$\Gamma \vdash c : \underline{C} \quad \Longrightarrow \quad [\![\Gamma]\!] \vdash [\![c]\!] : [\![\underline{C}]\!] \qquad (\forall c, t, \Gamma)$$

The proof of this theorem is given in Appendix A.2; it proceeds by induction on the typing derivation.

3.5 Purity-Aware Translation to OCaml

The basic compilation scheme's free monad representation introduces a substantial performance overhead for pure computations. This section presents an extended compilation scheme that avoids this overhead for pure computations.

The main aim of our extended compilation scheme is to differentiate between pure and impure computations. This is nicely summarised in the more nuanced compilation of computation types:

$$[\![A \,!\, \Delta]\!] = \begin{cases} [\![A]\!] & , \textit{if } \Delta = \emptyset \\ [\![A]\!] \texttt{ computation} & , \textit{if } \Delta \neq \emptyset \end{cases}$$

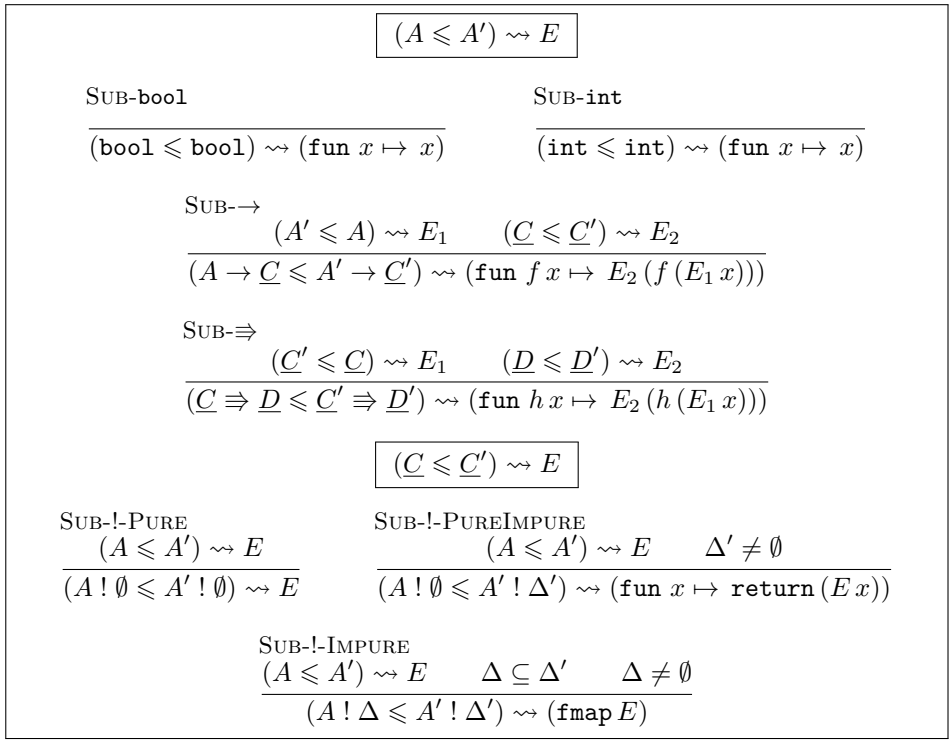

Figure 3.7: Subtyping induced coercions

At the term level, we express the extended compilation by means of type-and-effect-directed elaboration judgements that extend the declarative type system with a target OCaml expression. The key ingredients are the judgements that elaborate subtyping derivation of value types $A \leqslant B$ or of computation types $\underline{C} \leqslant \underline{D}$ into functions that coerce between the two types (Figure 3.7).

The reflexive cases for `bool` and `int` yield an identity coercion, while function and handler types give rise to pre- and post-composition of the coercions. We distinguish three different cases for the coercion between computation types. If both computations are pure, the coercion is just that between the values. A pure computation is coerced to an impure one by composing the value coercion with the `return` embedding. Finally, if the left-hand side computation is impure, so is the other one, and we map the coercion over the free monad with the predefined function

```
let rec fmap (f : 'a -> 'b) : ('a computation -> 'b
    computation) =
  function
  | Return x -> f x
  | Op1 (x, k) -> Op1 (x, (fun y -> fmap f (k y)))
  ...
  | Opn (x, k) -> Opn (x, (fun y -> fmap f (k y)))
```

The elaboration judgement $(\Gamma \vdash v : A) \rightsquigarrow E$ yields a corresponding OCaml expression E for the value v. Figure 3.8 shows the elaboration rules for values. There are three noteworthy cases. Firstly, SubVal applies the subtyping coercion to the elaborated term. Secondly, HandPure captures the case where the handler maps pure expressions to pure expressions, which is only possible when there are no operation cases. In this situation, we elaborate the handler into its elaborated value case. The impure case, HandImpure, works similar to the basic translation. For each operation $\mathtt{Op}_i$, we need to provide a case E_i. If an operation is handled by a handler, we take the corresponding elaboration. If the operation is not handled, but is still listed in the outgoing dirt Δ, we propagate it as before. Finally, the case when the operation is neither handled nor listed in Δ can never occur at runtime, so we may safely raise an (OCaml) exception. Such treatment ensures that a handler with empty Δ and non-empty $\mathcal{O}$ is translated with a pure co-domain.

The elaboration judgement $(\Gamma \vdash c : \underline{C}) \rightsquigarrow E$ yields a corresponding OCaml expression E for the computation c. Figure 3.9 shows the elaboration rules for computations. There are three differences with the basic compilation scheme. Firstly, a pure `return` v computation is translated just like the value v, i.e., without the `return` wrapper. Secondly, we distinguish between pure and impure do computations. The latter are translated in terms of the auxiliary $>>=$ operator like before, but the former can be simplified to a more efficient OCaml `let` expression. Finally, computation subtyping is handled in the same way as expression subtyping, by applying the coercion function.

Just like the basic compilation scheme, this more advanced elaboration is also type-preserving.

Theorem 3.

$$\Gamma \vdash c : \underline{C} \rightsquigarrow E \quad \Longrightarrow \quad [\![\Gamma]\!] \vdash E : [\![\underline{C}]\!] \qquad (\forall \Gamma, c, \underline{C}, E)$$

The proof of this theorem can be found in the work of Pretnar et al. [74].

Values

$$\boxed{\Gamma \vdash v : A \rightsquigarrow E}$$

SUBVAL
$$\frac{(\Gamma \vdash v : A) \rightsquigarrow E_1 \qquad (A \leqslant A') \rightsquigarrow E_2}{(\Gamma \vdash v : A') \rightsquigarrow (E_2\, E_1)}$$

VAR
$$\frac{(x : A) \in \Gamma}{(\Gamma \vdash x : A) \rightsquigarrow x}$$

CONST
$$\frac{(\mathtt{k} : A) \in \Sigma}{(\Gamma \vdash \mathtt{k} : A) \rightsquigarrow \mathtt{k}}$$

FUN
$$\frac{(\Gamma, x : A \vdash c : \underline{C}) \rightsquigarrow E}{(\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}) \rightsquigarrow (\mathtt{fun}\ x \mapsto E)}$$

HANDPURE
$$\frac{(\Gamma, x : A \vdash c_r : B \,!\, \emptyset) \rightsquigarrow E_r}{(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r\} : A \,!\, \emptyset \Rrightarrow B \,!\, \emptyset) \rightsquigarrow (\mathtt{fun}\ x \mapsto E_r)}$$

HANDIMPURE
$$\frac{\begin{array}{c}(\Gamma, x : A \vdash c_r : B \,!\, \Delta) \rightsquigarrow E_r \\ \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \quad (\Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta \vdash c_{\mathtt{Op}} : B \,!\, \Delta) \rightsquigarrow E_{\mathtt{Op}}\right]_{\mathtt{Op} \in \mathcal{O}} \\ E_i = \begin{cases} \mathtt{fun}\ x\, k \mapsto [\![c_{\mathtt{Op}_i}]\!] & \mathtt{Op}_i \in \mathcal{O} \\ \mathtt{fun}\ x\, k \mapsto \mathtt{op}_i x >\!\!>\!\!= k & \mathtt{Op}_i \in \Delta - \mathcal{O} \\ \mathtt{fun}\ x\, k \mapsto \mathtt{assert\ false} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{c}(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A \,!\, \Delta \cup \mathcal{O} \Rrightarrow B \,!\, \Delta) \\ \rightsquigarrow \mathtt{handler}\ \{\mathtt{return} = \mathtt{fun}\ x \mapsto E_r; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\}\end{array}}$$

Figure 3.8: Type-&-effect-directed compilation for Values

3.6 Implementation in Eff

To test the presented ideas in practice, we have implemented an optimising compiler for EFF, a prototype functional programming language with algebraic effects and handlers. This section describes the practical aspects of transforming EFF source code into an efficient OCAML code, and points out the main differences between the idealised representation and the actual implementation.

Computations

$$\boxed{\Gamma \vdash c : \underline{C} \rightsquigarrow E}$$

$$\textsc{SubComp}\ \frac{(\Gamma \vdash c : \underline{C}) \rightsquigarrow E_1 \qquad (\underline{C} \leqslant \underline{C}') \rightsquigarrow E_2}{(\Gamma \vdash c : \underline{C}') \rightsquigarrow (E_2\, E_1)}$$

$$\textsc{App}\ \frac{(\Gamma \vdash v_1 : A \to \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma \vdash v_2 : A) \rightsquigarrow E_2}{(\Gamma \vdash v_1\, v_2 : \underline{C}) \rightsquigarrow (E_1\, E_2)}$$

$$\textsc{LetRec}\ \frac{(\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}) \rightsquigarrow E_2}{(\Gamma \vdash \mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2 : \underline{D}) \rightsquigarrow (\mathtt{let\ rec}\ f\, x = E_1\ \mathtt{in}\ E_2)}$$

$$\textsc{Ret}\ \frac{(\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \mathtt{return}\ v : A\ !\ \emptyset) \rightsquigarrow E} \qquad \textsc{Op}\ \frac{(\mathtt{Op} : A \to B) \in \Sigma \qquad (\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \mathtt{Op}\, v : B\ !\ \{\mathtt{Op}\}) \rightsquigarrow (\mathtt{op}\, E)}$$

$$\textsc{DoPure}\ \frac{(\Gamma \vdash c_1 : A\ !\ \emptyset) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B\ !\ \emptyset) \rightsquigarrow E_2}{(\Gamma \vdash \mathtt{do}\ x \leftarrow c_1; c_2 : B\ !\ \emptyset) \rightsquigarrow (\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2)}$$

$$\textsc{DoImpure}\ \frac{(\Gamma \vdash c_1 : A\ !\ \Delta) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B\ !\ \Delta) \rightsquigarrow E_2 \qquad \Delta \neq \emptyset}{(\Gamma \vdash \mathtt{do}\ x \leftarrow c_1; c_2 : B\ !\ \Delta) \rightsquigarrow (E_1 \mathtt{>>=fun}\ x \mapsto E_2)}$$

$$\textsc{With}\ \frac{(\Gamma \vdash v : \underline{C} \Rrightarrow \underline{D}) \rightsquigarrow E_1 \qquad (\Gamma \vdash c : \underline{C}) \rightsquigarrow E_2}{(\Gamma \vdash \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D}) \rightsquigarrow (E_1\, E_2)}$$

$$\textsc{ITE}\ \frac{(\Gamma \vdash v : \mathtt{bool}) \rightsquigarrow E_1 \qquad (\Gamma \vdash c_1 : \underline{C}) \rightsquigarrow E_2 \qquad (\Gamma \vdash c_2 : \underline{C}) \rightsquigarrow E_3}{(\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \underline{C}) \rightsquigarrow \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3}$$

Figure 3.9: Type-&-effect-directed compilation for Computations

3.6.1 Converting Source to Core Syntax

The actual source syntax of Eff is based on OCaml's and features only a single syntactic sort of terms, which lumps together values and computations. This source syntax is desugared into the core syntax, which is very close to Effy, in a straightforward manner [6]. For example, the Eff program `if f x then 0 else g 5 x` gets elaborated into

$$\texttt{do}\ b \leftarrow f\ x; \texttt{if}\ b\ \texttt{then}\ (\texttt{return}\ 0)\ \texttt{else}\ (\texttt{do}\ h \leftarrow g\ 5; h\ x)$$

Our implementation supports standard features such as datatype declarations and control structures (like the conditional above), which we have omitted from Effy to avoid the clutter.

3.6.2 Translating Higher-Order Functions

One of the crucial properties of OCaml, which Eff respects, is that higher-order functions can accept both pure and impure functions as arguments. However, as described in Section 3.5, these two kinds of functions are translated differently, so higher-order functions need to be translated in a way that can deal with both pure and impure code. One possible approach is to compile multiple versions of each higher-order function, and select the appropriate one depending on the purity of its arguments. However, in this work, we opted for a more straightforward approach and labelled all higher-order functions as accepting impure arguments. Meaning, the translation does not make use of purity-aware compilation. If such a function is then applied to a pure argument, we use the coercions described in Section 3.5.

3.6.3 Embedding pure computations into values

Recall that in Effy, the two subterms of an application are values, whereas the application itself is a computation. This implies that a nested application `f x y` must be translated into `f x >>= fun g -> g y`. With the purity-aware translation, we can do a bit better when `f x` is pure (the common case for curried functions), and translate it as `let g = f x in g y`. However, this still incurs a significant overhead in comparison to `f x y`. To remedy that, we extend the core syntax of Eff with a coercion from computations into values, which behaves as a retraction of the value embedding.

In the basic translation to OCaml, we translate this coercion using a function `run : 'a computation -> 'a`, defined as

```
let run (Return x) = x
```

and the above application is translated as `(run (f x))y`. Note that this function is partial and causes a runtime error if applied to an operation call. To avoid that, we make sure we apply coercions only to computations guaranteed to be pure by the effect system. The second thing to note is that this translation is no better than `f x >>= fun g -> g y`, as both variants need to extract the value, returned by `f x`.

In the purity-aware translation, we make the coercion invisible, just like the value embedding, and we translate the nested application simply as `f x y`, resulting in zero overhead.

3.6.4 Extensible Set of Operations

In Section 3.4 we assumed a fixed set of operations. However, users may want to declare their own operations, and EFF enables them to do so through declarations such as:

```
effect Decide : unit -> bool
```

To support this extensibility in our translation, we make use of OCAML's extensible (GADT)[1] variant type feature to define an initially empty type of operations, indexed by their argument and result type:

```
type ('arg, 'res) operation = ..
```

Then, an operation declaration like the one above can be translated to an extension of the `operation` type:

```
type (_, _) operation += Decide : (unit, bool) operation
```

Next, the free monad representation is adapted to:

```
type 'a computation =
| Return: 'a -> 'a computation
| Call: ('arg, 'res) operation * 'arg * ('res -> 'a
    computation) -> 'a computation
```

where instead of multiple constructors, we have a single one that takes the called operation, its argument and its continuation.

Handler cases are described with two fields: a return case is a function that takes a value and returns a free monad representation, and a function that takes an operation to its appropriate case:

[1] Generalized algebraic data type

```
type ('a, 'b) handler_cases = {
  return: 'a -> 'b computation;
  operations: 'arg 'res. ('arg, 'res) operation ->
    'arg -> ('res -> 'b computation) -> 'b computation
}
```

A handler $\{\texttt{return}\ x \mapsto c_r, \mathtt{Op}_1\ x\ k \mapsto c_{\mathtt{Op}_1}, \ldots, \mathtt{Op}_n\ x\ k \mapsto c_{\mathtt{Op}_n}\}$ is translated as:

```
handler {
  return = (fun x -> ⟦c_r⟧);
  operations = (function
    | Op_1 -> (fun x k -> ⟦c_Op_1⟧)
    ...
    | Op_n -> (fun x k -> ⟦c_Op_n⟧)
    | op' -> (fun x k -> Call (op', x, k))
    )
  }
```

where the last case of the `operations` function reinvokes any operation `op'` that is not captured by the handler so it propagates to a higher level handler (if there is any).

The function `handler` and computation sequencing are redefined analogously:

```
let rec handler (h : ('a, 'b) handler_cases) : ('a
    computation -> 'b computation) =
  function
  | Return x -> h.return x
  | Call (op, x, k) -> (h.operations op) x (fun y ->
      handler h (k y))

let rec (>>=) (c : 'a computation) (f : 'a -> 'b
    computation) : 'b computation =
  match c with
  | Return x -> f x
  | Call (op, x, k) -> Call (op, x, (fun y -> (k y) >>= f)
      )
```

3.7 Evaluation

We evaluate the effectiveness of our optimising compiler for Eff on two different types of benchmarks. First, we compare our different compilation schemes with hand-written OCaml code to show the impact of every optimisation technique we developed. Then, we measure our compiler's best performance which combines all

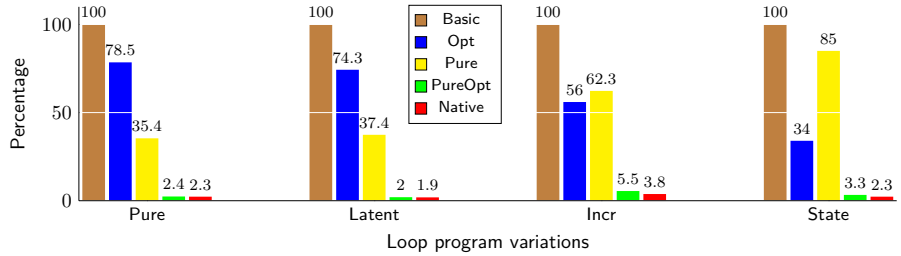

Figure 3.10: Relative run-times of Loops example

the optimisations against other OCaml-based implementations of algebraic effects and handlers. All benchmarks were run on a MacBook Pro with a 2.5 GHz Intel Core I7 processor and 16 GB 1600 MHz DDR3 RAM running Mac OS 10.12.3.

3.7.1 Eff versus OCaml

Our first evaluation, in Fig. 3.10, considers four different variations on the looping program we introduced in Section 3.2.1: 1) `Pure` is a version without side-effects, 2) `Latent` contains an operation that is never called during the execution of the benchmark, 3) `Incr` calls a single increment operation that increments an implicit state, 4) `State` is the version of Section 3.2.1 that increments the implicit state with the `Get` and `Put` operations. We compile these programs in four different ways: 1) basic compilation mode without any optimisation (Basic), 2) purity-aware compilation (Pure), 3) source-to-source optimisations (Opt), 4) the combination of the previous two (PureOpt). Finally, we compare these different versions against hand-written (Native) OCaml code: 1) a pure loop, 2) a latent OCaml exception, 3) a reference cell increment, and 4) a reference cell read followed by a write. The programs were compiled with version 4.02.2 of the OCaml compiler.

In total we have 20 compiled programs, we run each program for 10,000 iterations. In Figure 3.10 shows the resulting times relative to the basic version which scores the slowest performance. The results show a substantial gap between the basic compilation scheme and the hand-written OCaml, in the range of 25×–50× in favour of the hand-written OCaml code. The source-to-source transformations and purity-aware code generation each have individually varying success in reducing the gap to a smaller, but still significant level. It is only when the two optimisations are combined that we obtain a performance that is competitive with the hand-written versions (1×-1.5×). In particular, the combined optimisations succeed in eliminating all traces of the handlers and the free monad from the generated OCaml code.

3.7.2 Eff versus Other Systems

Our second evaluation features two different versions of the well-known N-Queens problem. Both versions use the same underlying program to explore the combinatorial space with the operations `Decide : unit -> bool` and `Fail : unit -> void` for non-determinism. The two versions only differ in which handler they use to interpret the operations.

The first handler computes *all solutions* and returns them in a list and the second computes only the *first solution*:

```
let all_solutions = handler
  | return x    -> [x]
  | #Decide _ k -> k true @ k false
  | #Fail _ _   -> []

let first_solution = handler
  | return x     -> Some x
  | #Decide _ k -> match k true with
                   | Some x -> Some x
                   | None -> k false
  | #Fail _ _    -> None
```

We compare the fully optimised Eff version against the same programs in three OCaml-based systems and use the hand-written OCaml program as a baseline. The three systems are Multicore OCaml [26, 27] (MultiCore), which provides native lightweight threads for running continuations, but requires expensive copying of the continuation for the non-linear use of the continuation in the above two handlers. The second and third system are the effect handler implementation of Kammar et al. [41] OCaml implementation that uses monad transformers [53], to compile effects and handlers (HandlersInAction). We also compare against the *Eff Directly in* OCaml implementation [47] (EffInOCaml). The last two implementations are based on OCaml's DelimCC libray for delimited control [44]. Because this library does not support native compilation, we compile all benchmarks to bytecode and run that instead. We compare our work also with the OCaml backend of Links [35, 15]. However, the performance of Links is on average twenty times slower than the hand-written OCaml code and thus we exclude the results hence they would dwarf all other runtimes.

Figure 3.11 shows the `all_solutions` runtimes of the different systems for different problem sizes, each time relative to the hand-written OCaml code. The results clearly show that Eff is consistently the fastest and competitive with hand-written OCaml code. We see an even more uniform picture in Figure 3.12, where Eff is consistently 25-30% faster than the closest competitor Multicore OCaml.

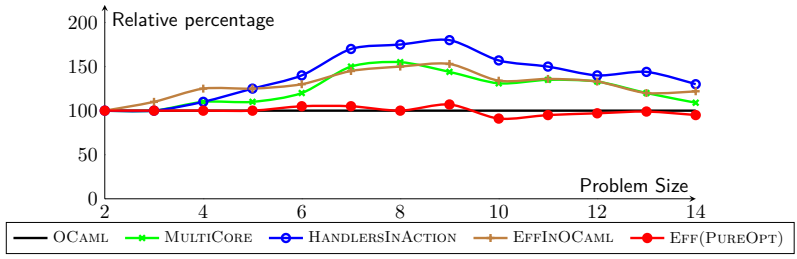

Figure 3.11: Results of running N-Queens for all solutions on multiple systems

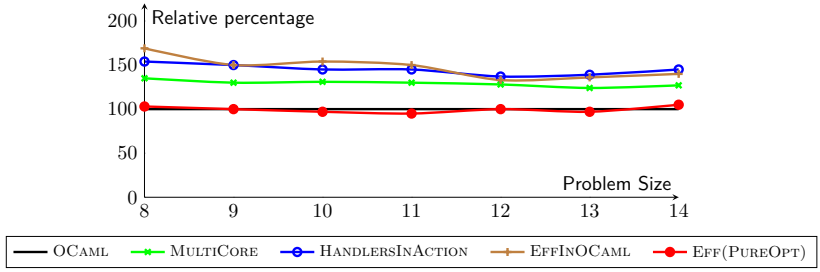

Figure 3.12: Results of running N-Queens for one solution on multiple systems

3.8 Discussion

This chapter has presented a two-pronged approach for the optimisation of the compilation of algebraic effects and handlers. First, we perform some source-to-source transformations, including the specialisation of recursive function definitions for appropriate handlers. Then we generate target code in a purity-aware fashion. Our innovative combination shows that the synergy between these two approaches is effective at eliminating handlers from some benchmarks and obtaining performance that is competitive with hand-written code and better than that of existing non-optimising implementations of algebraic effects and handlers.

However, due to the implicit type-system that the Eff calculus is based on, we faced some difficulties with the optimisation of more extensive programs. Eff computes the necessary type information through an inference algorithm [71]. For practical reasons, in order to build more significant programs, the effect system of Eff is polymorphic. Furthermore, to account for sub-typing, the inference algorithm is constraint-based, and a type scheme consists of a quantified type together with a set of constraints between its parameters. For example, the type of the identity

function `fun` $x \mapsto$ `return` x is:

$$\forall \alpha_1, \alpha_2, \delta_1.\ \alpha_1 \to \alpha_2\ !\ \delta_1 \mid \alpha_1 \leqslant \alpha_2 \tag{3.11}$$

where α_i and δ_i are type and dirt variables respectively.

The existing algorithm of Eff computes the most general type scheme of a given term in a bottom-up fashion, where constraints of sub-terms are joined together with additional constraints determined by the shape of the term.

In order for our optimisations to function correctly, they need full access to the types of all subterms. Moreover, the optimisations manipulate the types of the subterms. Also, function specialisation uses the inference system information to infer the type of the function it is going to specialise and then gives the new function a type based on the old function's type. This process can give rise to implementation bugs because of the implicit sub-typing of the sub-terms in an Eff term. In some programs, we cannot figure out the exact type of a sub-term since it can have many types implicitly. Moreover, since the optimisations can change such types, bugs can be introduced.

In order to remedy this issue, in our implementation, we construct terms through the use of "smart" constructors, which take already typed subterms as arguments and contain the necessary logic to return the appropriately annotated term. The inference algorithm then traverses the untyped term and applies the corresponding smart constructors. However, this was not enough to solve the issue with the bugs introduced by optimisations.

The optimisation showed much success with a small range of programs, and this acts as a proof of concept that the optimisations can be effective on run-times. However, the need for a more concrete inference system is present so that the application of the optimisations would have all the needed exact information about the terms' types. In the next chapter, we will discuss a new Eff calculus that opens the gate for such applications.

Chapter notes and contributions This chapter is based on the report *"Efficient compilation of algebraic effects and handlers"* by Pretnar et al. [74]. My contribution to this work is the following:

- Formalisation of the term rewriting rules.
- Formalisation of the function specialisation optimisation.
- Implementation of the formalised optimisations in Eff.
- Benchmarking against hand-written OCaml code and other systems.

- Partial work on the proofs provided in Appendix A.

Chapter 4

Explicit Subtyping for Algebraic Effects

4.1 Introduction

As we saw in the previous chapter, optimising the compiler of Eff showed promising results, in some cases reaching even the performance of the hand-tuned code. However, the optimisations introduced were very fragile. We argued that the main reason behind this fragility is the complexity of subtyping in combination with the implicit typing of Eff's core language. Therefore, the main aim of the chapter is to propose an explicitly-typed calculus based on subtyping in the aim to eliminate the fragility of implicit subtyping in optimisations.

We divide this chapter as follows: First, we give a brief overview of the proposed calculus in Section 4.2. Section 4.3 presents ImpEff, a polymorphic implicitly-typed calculus for algebraic effects and handlers with a subtyping-based type-and-effect system. ImpEff is essentially a (de-sugared) source language as it appears in the compiler front-end of a language like Eff.

Next, Section 4.4 presents ExEff, the core calculus, which combines explicit System F-style polymorphism with explicit coercions for subtyping in the style of Reazu-Tannen [11]. This calculus comes with a type-and-effect system, a small-step operational semantics and a proof of type-safety.

Section 4.5 specifies the typing-directed elaboration of ImpEff into ExEff and presents a type inference algorithm for ImpEff that produces the elaborated ExEff term as a by-product. It also establishes that the elaboration preserves

typing, and that the algorithm is sound with respect to the specification and yields principal types.

Section 4.6 defines SkelEff, which is a variant of ExEff without effect information or coercions. SkelEff is also representative of Multicore OCaml's support for algebraic effects and handlers [27], which is a possible compilation target of Eff. By showing that the erasure from ExEff to SkelEff preserves semantics, we establish that ExEff's coercions are computationally irrelevant and that, despite the existence of multiple proofs for the same subtyping, the coherence property does not break. To enable erasure, ExEff annotates its types with *(type) skeletons*, which capture the erased counterpart and are, to our knowledge, a novel contribution.

We have also developed an implementation of a compiler from Eff to OCaml with ExEff as its core language, and it can be found on https://github.com/matijapretnar/eff/tree/explicit-effect-subtyping. The proofs of the theorems provided in this chapter were developed by Pretnar, and they can be found on https://github.com/matijapretnar/proofs/tree/master/explicit-effect-subtyping

4.2 Overview

This section presents an informal overview of the ExEff calculus and the main issues with elaborating to and erasing from it.

4.2.1 Elaborating Subtyping

Assume effects $\texttt{Tick} : \texttt{Unit} \to \texttt{Unit}$ and $\texttt{Tock} : \texttt{Unit} \to \texttt{Unit}$ that take a unit value as a parameter and yield a unit value as a result.

Also, consider the computation $\texttt{do}\ x \leftarrow \texttt{Tick}\ (); f\ x$ and assume that f has the function type $\texttt{Unit} \to \texttt{Unit}\ !\ \{\texttt{Tock}\}$, taking unit values to unit values and perhaps calling Tock operations in the process. The whole computation then has the type $\texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$ as it returns the unit value and may call Tick and Tock.

The above typing implicitly appeals to subtyping in several places. For instance, $\texttt{Tick}\ ()$ has type $\texttt{Unit}\ !\ \{\texttt{Tick}\}$ and $f\ x$ type $\texttt{Unit}\ !\ \{\texttt{Tock}\}$. Yet, because they are sequenced with do, the type system expects that they have the same set of effects. The discrepancies are implicitly reconciled by the subtyping which admits both $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$ and $\{\texttt{Tock}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$.

We elaborate the ImpEff term into the explicitly-typed core language ExEff to make those appeals to subtyping explicit by means of casts with coercions:

$$\texttt{do}\ x \leftarrow ((\texttt{Tick}\ ()) \rhd \gamma_1); (f\ x) \rhd \gamma_2$$

A coercion γ is a witness for a subtyping $A\ !\ \Delta \leqslant A'\ !\ \Delta'$ and can be used to cast a term c of type $A\ !\ \Delta$ to a term $c \rhd \gamma$ of type $A'\ !\ \Delta'$. In the above term, γ_1 and γ_2 respectively witness $\texttt{Unit}\ !\ \{\texttt{Tick}\} \leqslant \texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$ and $\texttt{Unit}\ !\ \{\texttt{Tock}\} \leqslant \texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$.

4.2.2 Polymorphic Subtyping for Types and Effects

The above basic example only features monomorphic types and effects. Yet, our calculus also supports polymorphism, which makes it considerably more expressive. For instance the type of f in $\texttt{let}\ f = (\texttt{fun}\ g \mapsto g\ ())\ \texttt{in}\ \ldots$ is generalised to:

$$\forall \alpha, \alpha'. \forall \delta, \delta'. \alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\texttt{Unit} \to \alpha\ !\ \delta) \to \alpha'\ !\ \delta'$$

This polymorphic type scheme follows the qualified types convention [39] where the type $(\texttt{Unit} \to \alpha\ !\ \delta) \to \alpha'\ !\ \delta'$ is subjected to several qualifiers, in this case $\alpha \leqslant \alpha'$ and $\delta \leqslant \delta'$. The universal quantifiers bind the type variables α and α', and the effect set variables δ and δ'.

The elaboration of f into ExEff introduces explicit binders for both the quantifiers and the qualifiers, as well as the explicit casts where subtyping is used.

$$\Lambda\alpha.\Lambda\alpha'.\Lambda\delta.\Lambda\delta'.\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta').\texttt{fun}\ (g : \texttt{Unit} \to \alpha\,!\,\delta) \mapsto (g\ ()) \rhd (\omega\,!\,\omega')$$

Here the binders for qualifiers introduce coercion variables ω between pure types and ω' between operation sets, which are then combined into a computation coercion $\omega\ !\ \omega'$ and used for casting the function application $g\ ()$ to the expected type.

Suppose that h has type $\texttt{Unit} \to \texttt{Unit}\,!\,\{\texttt{Tick}\}$ and $f\ h$ type $\texttt{Unit}\,!\,\{\texttt{Tick}, \texttt{Tock}\}$. In the ExEff calculus the corresponding instantiation of f is made explicit through type and coercion applications

$$f\ \texttt{Unit}\ \texttt{Unit}\ \{\texttt{Tick}\}\ \{\texttt{Tick}, \texttt{Tock}\}\ \gamma_1\ \gamma_2\ h$$

where γ_1 needs to be a witness for $\texttt{Unit} \leqslant \texttt{Unit}$ and γ_2 for $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$.

4.2.3 Guaranteed Erasure with Skeletons

One of our main requirements for ExEff is that its effect information and subtyping can be easily erased. The reason is twofold. Firstly, we want to show that neither

plays a role in the runtime behaviour of ExEff programs. Secondly and more importantly, we want to use a conventionally typed (System F-like) functional language such as OCaml as a backend for the Eff compiler.

At first, erasure of both effect information and subtyping seems easy: simply drop that information from types and terms. However, by dropping the effect variables and subtyping constraints from the type of f, we get $\forall\alpha, \alpha'.(\texttt{Unit} \to \alpha) \to \alpha'$ instead of the expected type $\forall\alpha.(\texttt{Unit} \to \alpha) \to \alpha$. In our naive erasure attempt, we have carelessly discarded the connection between α and α'. A more appropriate approach to erasure would be to unify the types in dropped subtyping constraints. However, unifying types may reduce the number of type variables when they become instantiated, so corresponding binders need to be dropped, greatly complicating the erasure procedure and its meta-theory.

Fortunately, there is an easier way by tagging all bound type variables with *skeletons*, which are barebone types without effect information. For example, the skeleton of a function type $A \to B \,!\, \Delta$ is $\tau_1 \to \tau_2$, where τ_1 is the skeleton of A and τ_2 the skeleton of B. In ExEff every well-formed type has an associated skeleton, and any two types $A_1 \leqslant A_2$ share the same skeleton. In particular, binders for type variables are explicitly annotated with skeleton variables ς. For instance, the actual type of f is:

$$\forall\varsigma.\forall(\alpha : \varsigma), (\alpha' : \varsigma).\forall\delta, \delta'.\alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\texttt{Unit} \to \alpha \,!\, \delta) \to \alpha' \,!\, \delta'$$

The skeleton quantifications and annotations also appear at the term-level:

$$\Lambda\varsigma.\Lambda(\alpha : \varsigma).\Lambda(\alpha' : \varsigma).\Lambda\delta.\Lambda\delta'.\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta').\ldots$$

Now erasure is really easy: we drop not only effect and subtyping-related term formers, but also type binders and application. We do retain skeleton binders and applications, which take over the role of (plain) types in the backend language. In terms, we replace types by their skeletons. For instance, for f we get:

$$\Lambda\varsigma.\texttt{fun}\ (g : \texttt{Unit} \to \varsigma) \mapsto g\,() \quad : \quad \forall\varsigma.(\texttt{Unit} \to \varsigma) \to \varsigma$$

4.3 The ImpEff Language

This section presents ImpEff, a basic functional calculus with support for algebraic effect handlers. It is similar to the calculus presented in Chapter 2 but we add type polymorphism which increases the expressivity of the calculus.

Terms

$$
\begin{aligned}
\text{value } v &::= x \mid () \mid \texttt{fun}\ x \mapsto c \mid h \\
\text{handler } h &::= \{\texttt{return}\ x \mapsto c_r, \texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}\} \\
\text{computation } c &::= \texttt{return}\ v \mid \texttt{Op}\ v\ (y.c) \mid \texttt{do}\ x \leftarrow c_1; c_2 \\
&\quad \mid \texttt{handle}\ c\ \texttt{with}\ v \mid v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c
\end{aligned}
$$

Types & Constraints

$$
\begin{aligned}
\text{skeleton } \tau &::= \varsigma \mid \texttt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \\
\text{value type } A, B &::= \alpha \mid \texttt{Unit} \mid A \to \underline{C} \mid \underline{C} \Rrightarrow \underline{D} \\
\text{qualified type } K &::= A \mid \pi \Rightarrow K \\
\text{polytype } S &::= K \mid \forall \varsigma.S \mid \forall \alpha{:}\tau.S \mid \forall \delta.S \\
\text{computation type } \underline{C}, \underline{D} &::= A\,!\,\Delta \\
\text{dirt } \Delta &::= \delta \mid \emptyset \mid \{\texttt{Op}\} \cup \Delta \\
\text{simple constraint } \pi &::= A_1 \leqslant A_2 \mid \Delta_1 \leqslant \Delta_2 \\
\text{constraint } \rho &::= \pi \mid \underline{C} \leqslant \underline{D}
\end{aligned}
$$

Figure 4.1: IMPEFF Syntax

4.3.1 Syntax

Figure 4.1 presents the syntax of the IMPEFF. There are two main kinds of terms: (pure) values v and (dirty) computations c. The latter may call effectful operations. Handlers h are a subsidiary sort of values. We assume a given set of *operations* $\texttt{Op}$, such as `Get` and `Put`. We abbreviate $\texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}$ as $[\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}$, and write $\mathcal{O}$ to denote the set $\{\texttt{Op}_1, \ldots, \texttt{Op}_n\}$.

Similarly, we distinguish between two basic sorts of types: the value types A, B and the computation types $\underline{C}, \underline{D}$. There are four forms of value types: type variables α, function types $A \to \underline{C}$, handler types $\underline{C} \Rrightarrow \underline{D}$ and the `Unit` type. Skeletons τ capture the shape of types, so, by design, their forms are identical. The computation type $A\,!\,\Delta$ is assigned to a computation returning values of type A and potentially calling operations from the *dirt* set Δ. A dirt set is a recursive structure that contains zero or more operations $\texttt{Op}$ and has the base case of either an empty set or a dirt variable δ. Despite this representation of dirt, the intended semantics of dirt sets Δ is that the order of operations $\texttt{Op}$ is irrelevant. Similarly to all HM-based systems, we discriminate between value types (or monotypes) A, qualified types K and polytypes (or type schemes) S. (Simple) subtyping constraints π denote inequalities between either value types or dirts. We also present the more general

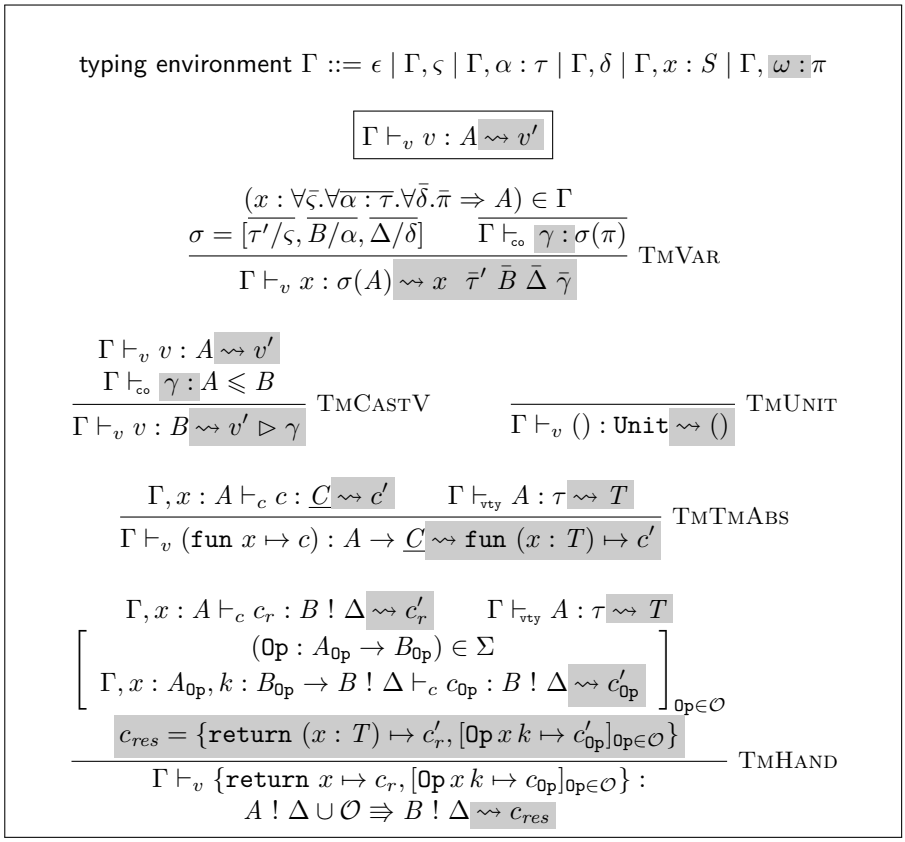

Figure 4.2: ImpEff Typing & Elaboration for values

form of constraints ρ that includes inequalities between computation types (as we illustrate in Section 4.3.2 below, this allows for a single, uniform constraint entailment relation). Finally, polytypes consist of zero or more skeleton, type or dirt abstractions followed by a qualified type.

4.3.2 Typing

Figures 4.2 and 4.3 present the typing rules for values and computations respectively, along with a typing-directed elaboration into our target language ExEff. In order to simplify the presentation, in this section we focus exclusively on typing. The

typing environment $\Gamma ::= \epsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : S \mid \Gamma, \omega : \pi$

$$\boxed{\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}$$

$$\frac{\Gamma \vdash_c c : \underline{C}_1 \rightsquigarrow c' \qquad \Gamma \vdash_{\text{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_c c : \underline{C}_2 \rightsquigarrow c' \rhd \gamma}\ \textsc{TmCastC} \qquad \frac{\Gamma \vdash_v v_1 : A \to \underline{C} \rightsquigarrow v_1' \qquad \Gamma \vdash_v v_2 : A \rightsquigarrow v_2'}{\Gamma \vdash_c v_1\ v_2 : \underline{C} \rightsquigarrow v_1'\ v_2'}\ \textsc{TmTmApp}$$

$$\frac{S = \forall \bar{\varsigma}.\overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow A \qquad \Gamma, \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \overline{\omega : \pi} \vdash_v v : A \rightsquigarrow v' \qquad \Gamma, x : S \vdash_c c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash_c \texttt{let}\ x = v\ \texttt{in}\ c : \underline{C} \rightsquigarrow \texttt{let}\ x = \Lambda\bar{\varsigma}.\Lambda\overline{\alpha : \tau}.\Lambda\bar{\delta}.\Lambda(\overline{\omega : \pi}).v'\ \texttt{in}\ c'}\ \textsc{TmLet}$$

$$\frac{\Gamma \vdash_v v : A \rightsquigarrow v'}{\Gamma \vdash_c \texttt{return}\ v : A\ !\ \emptyset \rightsquigarrow \texttt{return}\ v'}\ \textsc{TmReturn}$$

$$\frac{(\texttt{Op} : A_{\texttt{Op}} \to B_{\texttt{Op}}) \in \Sigma \qquad \Gamma \vdash_v v : A_{\texttt{Op}} \rightsquigarrow v' \qquad \Gamma, y : B_{\texttt{Op}} \vdash_c c : A\ !\ \Delta \rightsquigarrow c' \qquad \Gamma \vdash_{\text{vty}} B_{\texttt{Op}} : \tau \rightsquigarrow T_{\texttt{Op}} \qquad \texttt{Op} \in \Delta}{\Gamma \vdash_c \texttt{Op}\ v\ (y.c) : A\ !\ \Delta \rightsquigarrow \texttt{Op}\ v'\ (y : T_{\texttt{Op}}.c')}\ \textsc{TmOp}$$

$$\frac{\Gamma \vdash_c c_1 : A\ !\ \Delta \rightsquigarrow c_1' \qquad \Gamma, x : A \vdash_c c_2 : B\ !\ \Delta \rightsquigarrow c_2'}{\Gamma \vdash_c \texttt{do}\ x \leftarrow c_1; c_2 : B\ !\ \Delta \rightsquigarrow \texttt{do}\ x \leftarrow c_1'; c_2'}\ \textsc{TmDo}$$

$$\frac{\Gamma \vdash_v v : \underline{C} \Rrightarrow \underline{D} \rightsquigarrow v' \qquad \Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash_c \texttt{handle}\ c\ \texttt{with}\ v : \underline{D} \rightsquigarrow \texttt{handle}\ c'\ \texttt{with}\ v'}\ \textsc{TmHandle}$$

Figure 4.3: ImpEff Typing & Elaboration for computations

parts of the rules that concern elaboration are highlighted in gray and are discussed in Section 4.5.

We capture all defined operations along with their types in a global signature Σ. The typing environment Γ keeps track of several variables during typing. It contains skeleton variables ς, dirt variables δ, type variables with skeleton attached $\alpha : \tau$ and variables with polytype $x : S$. The typing environment also contains coercion variables ω that we will discuss in the next section.

Values Typing for values takes the form $\Gamma \vdash_v v : A$ and, given a typing environment Γ, checks a value v against a value type A.

Rule TmVar handles term variables. Given that x has type $(\forall \overline{\varsigma}.\overline{\alpha : \tau}.\forall \overline{\delta}.\overline{\pi} \Rightarrow A)$, we appropriately instantiate the skeleton $(\overline{\varsigma})$, type $\overline{(\alpha)}$, and dirt $(\overline{\delta})$ variables, and ensure that the instantiated wanted constraints $\overline{\sigma(\pi)}$ are satisfied, via the side condition $\overline{\Gamma \vdash_{\text{co}} \gamma : \sigma(\pi)}$. Rule TmCastV allows casting the type of a value v from A to B, if A is a subtype of B (upcasting). Rule TmTmAbs checks lambda abstractions and omits freshness conditions by adopting the Barendregt convention which states that all bound variables are different from the free variables [4]. Finally, Rule TmHand gives typing for handlers. It requires that the right-hand sides of the return clause and all operation clauses have the same computation type $(B\,!\,\Delta)$ and that all operations mentioned are part of the top-level signature Σ. The result type takes the form $A\,!\,\Delta \cup \mathcal{O} \Rrightarrow B\,!\,\Delta$, capturing the intended handler semantics: given a computation of type $A\,!\,\Delta \cup \mathcal{O}$, the handler (a) produces a result of type B, (b) handles operations $\mathcal{O}$, and (c) propagates unhandled operations Δ to the output.

Computations Typing for computations takes the form $\Gamma \vdash_c c : \underline{C}$ and, given a typing environment Γ, checks a computation c against a type $\underline{C}$.

Rule TmCastC behaves like Rule TmCastV, but for computation types. Rule TmLet handles polymorphic, non-recursive let-bindings. Rule TmReturn handles `return` v computations. The keyword `return` effectively lifts a value v of type A into a computation of type $A\,!\,\emptyset$. Rule TmOp checks operation calls. First, we ensure that v has the appropriate type, as specified by the signature of `Op`. Then, the continuation $(y.c)$ is checked. The side condition `Op` $\in \Delta$ ensures that the called operation `Op` is captured in the result type. Rule TmDo handles sequencing. Given that c_1 has type $A\,!\,\Delta$, the pure part of the result of type A is bound to term variable x, which is brought in scope for checking c_2. As we mentioned in Section 4.2, all computations in a `do`-construct should have the same effect set, Δ. Rule TmHandle eliminates handler types, just as Rule TmTmApp eliminates arrow types.

Constraint Entailment The specification of constraint entailment takes the form $\Gamma \vdash_{\text{co}} \rho$ and is presented in Figure 4.4. Notice that we use ρ instead of π, which allows us to capture subtyping between two value types, computation types or dirts, within the same relation. Subtyping can be established in several ways:

Rule CoVar handles given assumptions. Rules VCoRefl and DCoRefl express that subtyping is reflexive, for both value types and dirts. Notice that we do not have a rule for the reflexivity of computation types since, as we illustrate below, it can be established using the reflexivity of their subparts. Rules VCoTrans, CCoTrans and DCoTrans express the transitivity of subtyping for value types, computation types and dirts, respectively. Rule VCoArr establishes inequality of arrow types. As usual, the arrow type constructor is contravariant in the argument type. Rules VCoArrL and CCoArrR are the inversions of Rule VCoArr, allowing us to establish the relation between the subparts of the arrow types. Rules VCoHand, CCoHL, and CCoHR work similarly, for handler types. Rule CCoComp captures the covariance of the type constructor (!), establishing subtyping between two computation types if subtyping is established for their respective subparts. Rules VCoPure and DCoImpure are its inversions. Finally, Rules DCoNil and DCoOp establish subtyping between dirts. Rule DCoNil captures that the empty dirty set $\emptyset$ is a subdirt of any dirt Δ and Rule DCoOp expresses that dirt subtyping preserved under extension with the same operation `Op`. Notice that some forms of subtyping cannot be derived from the given rules, such as $\delta \leq \{\texttt{Op}\} \cup \delta$. This is not a problem because the constraint generation rules that we discuss later in this chapter do not generate subtyping constraints that lead to those forms during solving. Consequently, the completeness of the constraint solving algorithm is preserved according to Theorem 7.

4.3.3 Well-formedness of Types, Constraints, Dirts, and Skeletons for ImpEff

The relations $\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T$ and $\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}$ check the well-formedness of value and computation types respectively. Similarly, relations $\Gamma \vdash_{\text{ct}} \rho \rightsquigarrow \rho$ and $\Gamma \vdash_{\Delta} \Delta$ check the well-formedness of constraints and dirts, respectively.

Type Well-formedness & Elaboration

Since our system discriminates between value types and computation types, well-formedness of types is checked using two mutually recursive relations: $\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T$ (values), and $\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}$ (computations). We discuss each one separately below.

$\boxed{\Gamma \vdash_{\text{co}} \gamma : \rho}$ **Constraint Entailment**

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\text{co}} \omega : \pi}\ \textsc{CoVar} \qquad \frac{\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{co}} \langle T \rangle : A \leqslant A}\ \textsc{VCoRefl}$$

$$\frac{\Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\text{co}} \langle \Delta \rangle : \Delta \leqslant \Delta}\ \textsc{DCoRefl} \qquad \frac{\Gamma \vdash_{\text{co}} \gamma_1 : A_1 \leqslant A_2 \quad \Gamma \vdash_{\text{co}} \gamma_2 : A_2 \leqslant A_3}{\Gamma \vdash_{\text{co}} \gamma_1 \gg \gamma_2 : A_1 \leqslant A_3}\ \textsc{VCoTrans}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma_1 : \underline{C}_1 \leqslant \underline{C}_2 \quad \Gamma \vdash_{\text{co}} \gamma_2 : \underline{C}_2 \leqslant \underline{C}_3}{\Gamma \vdash_{\text{co}} \gamma_1 \gg \gamma_2 : \underline{C}_1 \leqslant \underline{C}_3}\ \textsc{CCoTrans} \qquad \frac{\Gamma \vdash_{\text{co}} \gamma_1 : \Delta_1 \leqslant \Delta_2 \quad \Gamma \vdash_{\text{co}} \gamma_2 : \Delta_2 \leqslant \Delta_3}{\Gamma \vdash_{\text{co}} \gamma_1 \gg \gamma_2 : \Delta_1 \leqslant \Delta_3}\ \textsc{DCoTrans}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma_1 : B \leqslant A \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \underline{C} \leqslant \underline{D}}{\Gamma \vdash_{\text{co}} \gamma_1 \to \gamma_2 : A \to \underline{C} \leqslant B \to \underline{D}}\ \textsc{VCoArr}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : A \to \underline{C} \leqslant B \to \underline{D}}{\Gamma \vdash_{\text{co}} \mathit{left}(\gamma) : B \leqslant A}\ \textsc{VCoArrL} \qquad \frac{\Gamma \vdash_{\text{co}} \gamma : A \to \underline{C} \leqslant B \to \underline{D}}{\Gamma \vdash_{\text{co}} \mathit{right}(\gamma) : \underline{C} \leqslant \underline{D}}\ \textsc{CCoArrR}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma_1 : \underline{C}_2 \leqslant \underline{C}_1 \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \underline{D}_1 \leqslant \underline{D}_2}{\Gamma \vdash_{\text{co}} \gamma_1 \Rrightarrow \gamma_2 : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}\ \textsc{VCoHand}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\text{co}} \mathit{left}(\gamma) : \underline{C}_2 \leqslant \underline{C}_1}\ \textsc{CCoHL}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\text{co}} \mathit{right}(\gamma) : \underline{D}_1 \leqslant \underline{D}_2}\ \textsc{CCoHR}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma_1 : A_1 \leqslant A_2 \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2}{\Gamma \vdash_{\text{co}} \gamma_1 \mathbin{!} \gamma_2 : A_1 \mathbin{!} \Delta_1 \leqslant A_2 \mathbin{!} \Delta_2}\ \textsc{CCoComp}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : A_1 \mathbin{!} \Delta_1 \leqslant A_2 \mathbin{!} \Delta_2}{\Gamma \vdash_{\text{co}} \mathit{pure}(\gamma) : A_1 \leqslant A_2}\ \textsc{VCoPure}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : A_1 \mathbin{!} \Delta_1 \leqslant A_2 \mathbin{!} \Delta_2}{\Gamma \vdash_{\text{co}} \mathit{impure}(\gamma) : \Delta_1 \leqslant \Delta_2}\ \textsc{DCoImpure} \qquad \frac{}{\Gamma \vdash_{\text{co}} \emptyset_{\Delta} : \emptyset \leqslant \Delta}\ \textsc{DCoNil}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : \Delta_1 \leqslant \Delta_2 \qquad (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma}{\Gamma \vdash_{\text{co}} \{\mathtt{Op}\} \cup \gamma : \{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2}\ \textsc{DCoOp}$$

Figure 4.4: ImpEff Constraint Entailment

Value Type Well-formedness

$$\boxed{\Gamma \vdash_{\text{vty}} S : \tau \rightsquigarrow T}$$

$$\frac{\alpha : \tau \in \Gamma}{\Gamma \vdash_{\text{vty}} \alpha : \tau \rightsquigarrow \alpha}\ \textsc{WfTyVar}$$

$$\frac{\Gamma \vdash_{\text{vty}} A : \tau_1 \rightsquigarrow T \qquad \Gamma \vdash_{\text{cty}} \underline{C} : \tau_2 \rightsquigarrow \underline{C}}{\Gamma \vdash_{\text{vty}} A \to \underline{C} : \tau_1 \to \tau_2 \rightsquigarrow T \to \underline{C}}\ \textsc{WfTyArr}$$

$$\frac{\Gamma \vdash_{\text{cty}} \underline{C} : \tau_1 \rightsquigarrow \underline{C}_1 \qquad \Gamma \vdash_{\text{cty}} \underline{D} : \tau_2 \rightsquigarrow \underline{C}_2}{\Gamma \vdash_{\text{vty}} \underline{C} \Rrightarrow \underline{D} : \tau_1 \Rrightarrow \tau_2 \rightsquigarrow \underline{C}_1 \Rrightarrow \underline{C}_2}\ \textsc{WfHandTy}$$

$$\frac{}{\Gamma \vdash_{\text{vty}} \texttt{Unit} : \texttt{Unit} \rightsquigarrow \texttt{Unit}}\ \textsc{WfUnitTy}$$

$$\frac{\Gamma \vdash_{\text{ct}} \pi \rightsquigarrow \pi \qquad \Gamma \vdash_{\text{vty}} K : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \pi \Rightarrow K : \tau \rightsquigarrow \pi \Rightarrow T}\ \textsc{WfCoAbsTy}$$

$$\frac{\Gamma, \alpha : \tau_1 \vdash_{\text{vty}} S : \tau_2 \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \alpha : \tau_1 . S : \tau_2 \rightsquigarrow \forall \alpha : \tau_1 . T}\ \textsc{WfTyAbs}$$

$$\frac{\Gamma, \delta \vdash_{\text{vty}} S : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \delta . S : \tau \rightsquigarrow \forall \delta . T}\ \textsc{WfDirtAbs} \qquad \frac{\Gamma, \varsigma \vdash_{\text{vty}} S : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \varsigma . S : \forall \varsigma . \tau \rightsquigarrow \forall \varsigma . T}\ \textsc{WfSkelAbs}$$

Figure 4.5: Well-formedness of Value types for ImpEff

Value Types Well-formedness for value types for ImpEff are defined in Figure 4.5. The judgement is syntax-directed on the structure of types; each rule corresponds to a value type syntactic form. Since ExEff types are a superset of ImpEff types, the elaboration-part (highlighted in gray) is the identity transformation to ExEff that we will discuss in the next section. The essence of the judgement is to check the well-scopedness of ImpEff types.

Computation Types Figure 4.6 shows the well-formedness for computation types. We ensure that both parts of a computation type (the value type and the dirt) are well-scoped under Γ, while elaborating the value-type into a proper ExEff representation.

Computation Type Well-formedness

$$\boxed{\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}}$$

$$\frac{\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\text{cty}} A \,!\, \Delta : \tau \rightsquigarrow T \,!\, \Delta}\ \textsc{WfCompTy}$$

Figure 4.6: Well-formedness of Computation types for ImpEff

Constraint Well-formedness

$$\boxed{\Gamma \vdash_{\text{ct}} \rho \rightsquigarrow \rho}$$

$$\frac{\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T_1 \qquad \Gamma \vdash_{\text{vty}} B : \tau \rightsquigarrow T_2}{\Gamma \vdash_{\text{ct}} A \leqslant B \rightsquigarrow T_1 \leqslant T_2}\ \textsc{WfVTyCt}$$

$$\frac{\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C_1} \qquad \Gamma \vdash_{\text{cty}} \underline{D} : \tau \rightsquigarrow \underline{C_2}}{\Gamma \vdash_{\text{ct}} \underline{C} \leqslant \underline{D} \rightsquigarrow \underline{C_1} \leqslant \underline{C_2}}\ \textsc{WfCTyCt}$$

$$\frac{\Gamma \vdash_{\Delta} \Delta_1 \qquad \Gamma \vdash_{\Delta} \Delta_2}{\Gamma \vdash_{\text{ct}} \Delta_1 \leqslant \Delta_2 \rightsquigarrow \Delta_1 \leqslant \Delta_2}\ \textsc{WfDirtCt}$$

Figure 4.7: Well-formedness of Constraints for ImpEff

Constraint Well-formedness

Well-formedness for constraints is given by judgement $\Gamma \vdash_{\text{ct}} \rho \rightsquigarrow \rho$ and is shown in Figure 4.7. All three rules check the constraint components for well-scopedness using the other defined well-formedness relations.

Dirt Well-formedness

Judgement $\Gamma \vdash_{\Delta} \Delta$ checks dirt well-formedness and is defined in Figure 4.8. In addition to checking that the dirt is well-scoped (illustrated by WfDirtVar), we also make sure that all operations in a dirt set are already defined, by looking them up in the globally visible signature Σ (WfOpDirt).

Dirt Well-formedness

$$\boxed{\Gamma \vdash_{\Delta} \Delta}$$

$$\frac{}{\Gamma \vdash_{\Delta} \emptyset}\ \textsc{WfEmpty} \qquad \frac{\delta \in \Gamma}{\Gamma \vdash_{\Delta} \delta}\ \textsc{WfDirtVar}$$

$$\frac{(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\Delta} \{\mathtt{Op}\} \cup \Delta}\ \textsc{WfOpDirt}$$

Figure 4.8: Well-formedness of dirts for ImpEff

Skeleton Well-formedness

$$\boxed{\Gamma \vdash_{\tau} \tau}$$

$$\frac{\varsigma \in \Gamma}{\Gamma \vdash_{\tau} \varsigma}\ \textsc{WfSkelVar} \qquad \frac{}{\Gamma \vdash_{\tau} \mathtt{Unit}}\ \textsc{WfSkelUnit}$$

$$\frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \to \tau_2}\ \textsc{WfSkel}{\to} \qquad \frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \Rrightarrow \tau_2}\ \textsc{WfSkel}{\Rrightarrow}$$

Figure 4.9: Well-formedness of skeletons for ImpEff

Skeleton Well-formedness

Finally, skeleton well-formedness is performed by judgement $\Gamma \vdash_{\tau} \tau$, as given in Figure 4.9. The relation is straightforward. Since the base sort of skeletons is either `Unit` or ς, the relation recursively reduces the arrow and handler types till it reaches either of the base sorts.

4.4 The ExEff Language

This section presents ExEff, the calculus which forms the core language of our optimising compiler. It is designed to explicitly state the type of every term in the language. There is no need for implicit subtyping in this calculus. This allows optimisations to be easily built on top of this calculus.

4.4.1 Syntax

Figure 4.10 presents ExEff's syntax. ExEff is an intensional type theory akin to System F [32], where every term encodes its own typing derivation. In essence, all abstractions and applications that are implicit in ImpEff, are made explicit in ExEff via new syntactic forms. Additionally, ExEff is impredicative, which is reflected in the lack of discrimination between value types, qualified types and type schemes; all non-computation types are denoted by T. While the impredicativity is not strictly required for the purpose at hand, it makes for a cleaner system.

Coercions Of particular interest is the use of explicit *subtyping coercions*, denoted by γ. ExEff uses these to replace the implicit casts of ImpEff (Rules TmCastV and TmCastC in Figures 4.2 and 4.3) with explicit casts $(v \rhd \gamma)$ and $(c \rhd \gamma)$ respectively.

Essentially, coercions γ are explicit witnesses of subtyping derivations: each coercion form corresponds to a subtyping rule. Subtyping forms a partial order, which is reflected in coercion forms $\gamma_1 \gg \gamma_2$, $\langle T \rangle$, and $\langle \Delta \rangle$. Coercion form $\gamma_1 \gg \gamma_2$ captures transitivity, while forms $\langle T \rangle$ and $\langle \Delta \rangle$ capture reflexivity for value types and dirts (reflexivity for computation types can be derived from these).

Subtyping for skeleton abstraction, type abstraction, dirt abstraction, and qualification is witnessed by coercion forms $\forall \varsigma.\gamma$, $\forall \alpha.\gamma$, $\forall \delta.\gamma$, and $\pi \Rightarrow \gamma$, respectively. Similarly, forms $\gamma[\tau]$, $\gamma[T]$, $\gamma[\Delta]$, and $\gamma_1 @ \gamma_2$ witness subtyping of skeleton instantiation, type instantiation, dirt instantiation, and coercion application, respectively.

Syntactic forms $\gamma_1 \to \gamma_2$ and $\gamma_1 \Rrightarrow \gamma_2$ capture injection for the arrow and the handler type constructor, respectively. Similarly, inversion forms *left*(γ) and *right*(γ) capture projection, following from the injectivity of both type constructors.

Coercion form $\gamma_1 \mathbin{!} \gamma_2$ witnesses subtyping for computation types, using proofs for their components. Inversely, syntactic forms *pure*(γ) and *impure*(γ) witness subtyping between the value- and dirt-components of a computation coercion.

Coercion forms $\emptyset_\Delta$ and $\{\mathtt{Op}\} \cup \gamma$ are concerned with dirt subtyping. Form $\emptyset_\Delta$ witnesses that the empty dirt $\emptyset$ is a subdirt of any dirt Δ. Lastly, coercion form $\{\mathtt{Op}\} \cup \gamma$ witnesses that subtyping between dirts is preserved under extension with a new operation. Note that we do not have an inversion form to extract a witness for $\Delta_1 \leqslant \Delta_2$ from a coercion for $\{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2$. The reason is that dirt sets are sets and not inductive structures. The set$\{\mathtt{Op}\}$ contains the operations and Δ in this case is either $\emptyset$ or δ. For instance, for $\Delta_1 = \{\mathtt{Op}\}$ and $\Delta_2 = \emptyset$ the latter subtyping holds, but the former does not.

Terms

$$\begin{aligned}
\text{value } v &::= x \mid () \mid \texttt{fun}\ (x : T) \mapsto c \mid h \\
&\mid \Lambda\varsigma.v \mid v\ \tau \mid \Lambda\alpha : \tau.v \mid v\ T \\
&\mid \Lambda\delta.v \mid v\ \Delta \mid \Lambda(\omega : \pi).v \mid v\ \gamma \mid v \rhd \gamma
\end{aligned}$$

$$\begin{aligned}
\text{handler } h ::= \{&\texttt{return}\ (x : T) \mapsto c_r, \\
&\texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}\}
\end{aligned}$$

$$\begin{aligned}
\text{computation } c ::= &\ \texttt{return}\ v \mid \texttt{Op}\ v\ (y : T.c) \mid \texttt{do}\ x \leftarrow c_1; c_2 \\
\mid &\ \texttt{handle}\ c\ \texttt{with}\ v \mid v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c \mid c \rhd \gamma
\end{aligned}$$

Types

$$\text{skeleton } \tau ::= \varsigma \mid \texttt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \mid \forall\varsigma.\tau$$

$$\begin{aligned}
\text{value type } T ::= &\ \alpha \mid \texttt{Unit} \mid T \to \underline{C} \mid \underline{C}_1 \Rrightarrow \underline{C}_2 \\
\mid &\ \forall\varsigma.T \mid \forall\alpha{:}\tau.T \mid \forall\delta.T \mid \pi \Rightarrow T
\end{aligned}$$

$$\text{simple coercion type } \pi ::= T_1 \leqslant T_2 \mid \Delta_1 \leqslant \Delta_2$$

$$\text{coercion type } \rho ::= \pi \mid \underline{C}_1 \leqslant \underline{C}_2$$

$$\begin{aligned}
\text{computation type } \underline{C} &::= T\ !\ \Delta \\
\text{dirt } \Delta &::= \delta \mid \emptyset \mid \{\texttt{Op}\} \cup \Delta
\end{aligned}$$

Coercions

$\gamma ::= \omega$	*Coercion Variable*
$\mid\ \gamma_1 \gg \gamma_2$	*Transitivity*
$\mid\ \langle T \rangle$	*Reflexivity over types*
$\mid\ \gamma_1 \to \gamma_2$	*Arrow type congruence*
$\mid\ \gamma_1 \Rrightarrow \gamma_2$	*Handler type*
$\mid\ \mathit{left}(\gamma)$	*Arrow left inversion*
$\mid\ \mathit{right}(\gamma)$	*Arrow right inversion*
$\mid\ \langle \Delta \rangle$	*Reflexivity over dirts*
$\mid\ \emptyset_\Delta$	*Empty set of constraints*
$\mid\ \{\texttt{Op}\} \cup \gamma$	*Operation congruence*
$\mid\ \forall\varsigma.\gamma$	*Skeleton quantification*
$\mid\ \gamma[\tau]$	*Skeleton instantiation*
$\mid\ \forall\alpha.\gamma$	*Type quantification*
$\mid\ \gamma[T]$	*Type instantiation*
$\mid\ \forall\delta.\gamma$	*Dirt quantification*
$\mid\ \gamma[\Delta]$	*Dirt instantiation*
$\mid\ \pi \Rightarrow \gamma$	*Constraint abstraction*
$\mid\ \gamma_1 @ \gamma_2$	*Constraint instantiation*
$\mid\ \gamma_1\ !\ \gamma_2$	*Computation type congruence*
$\mid\ \mathit{pure}(\gamma)$	*Computation type left inversion*
$\mid\ \mathit{impure}(\gamma)$	*Computation type right inversion*

Figure 4.10: EXEFF Syntax

$$\boxed{\Gamma \vdash_{\mathsf{v}} v : T}$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash_{\mathsf{v}} x : T}\ \textsc{ExVar} \qquad \frac{}{\Gamma \vdash_{\mathsf{v}} () : \mathtt{Unit}}\ \textsc{ExUnit}$$

$$\frac{\Gamma, x : T \vdash_{\mathsf{c}} c : \underline{C} \qquad \Gamma \vdash_{T} T : \tau}{\Gamma \vdash_{\mathsf{v}} (\mathtt{fun}\ x : T \mapsto c) : T \to \underline{C}}\ \textsc{ExTmAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} v : T_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathsf{v}} v \rhd \gamma : T_2}\ \textsc{ExCastV} \qquad \frac{\Gamma, \varsigma \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\varsigma.v : \forall\varsigma.T}\ \textsc{ExSkelAbs}$$

$$\frac{\Gamma, \alpha : \tau \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\alpha : \tau.v : \forall\alpha : \tau.T}\ \textsc{ExTyAbs} \qquad \frac{\Gamma, \delta \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\delta.v : \forall\delta.T}\ \textsc{ExDirtAbs}$$

$$\frac{\Gamma, \omega : \pi \vdash_{\mathsf{v}} v : T \qquad \Gamma \vdash_{\rho} \pi}{\Gamma \vdash_{\mathsf{v}} \Lambda(\omega : \pi).v : \pi \Rightarrow T}\ \textsc{ExCoAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} v : \pi \Rightarrow T \qquad \Gamma \vdash_{\mathsf{co}} \gamma : \pi}{\Gamma \vdash_{\mathsf{v}} v\ \gamma : T}\ \textsc{ExCoApp}$$

ExHandler

$$\frac{\begin{array}{c}\Gamma, x : T_x \vdash_{\mathsf{c}} c_r : T\,!\,\Delta \\ \left[(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma, x : T_1, k : T_2 \to T\,!\,\Delta \vdash_{\mathsf{c}} c_{\mathtt{Op}} : T\,!\,\Delta\right]_{\mathtt{Op} \in \mathcal{O}}\end{array}}{\Gamma \vdash_{\mathsf{v}} \{\mathtt{return}\ (x : T_x) \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : T_x\ !\ \Delta \cup \mathcal{O} \Rrightarrow T\ !\ \Delta}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} v : \forall\varsigma.T \qquad \Gamma \vdash_{\tau} \tau}{\Gamma \vdash_{\mathsf{v}} v\ \tau : T[\tau/\varsigma]}\ \textsc{ExSkelApp} \qquad \frac{\Gamma \vdash_{\mathsf{v}} v : \forall\alpha : \tau.T_1 \qquad \Gamma \vdash_{T} T_2 : \tau}{\Gamma \vdash_{\mathsf{v}} v\ T_2 : T_1[T_2/\alpha]}\ \textsc{ExTyApp}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} v : \forall\delta.T \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\mathsf{v}} v\ \Delta : T[\Delta/\delta]}\ \textsc{ExDirtApp}$$

Figure 4.11: ExEff Value Typing

$$\boxed{\Gamma \vdash_{\mathrm{c}} c : \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} v_1 : T \to \underline{C} \qquad \Gamma \vdash_{\mathrm{v}} v_2 : T}{\Gamma \vdash_{\mathrm{c}} v_1\ v_2 : \underline{C}}\ \textsc{ExApp}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} v : T \qquad \Gamma, x : T \vdash_{\mathrm{c}} c : \underline{C}}{\Gamma \vdash_{\mathrm{c}} \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C}}\ \textsc{ExLet} \qquad \frac{\Gamma \vdash_{\mathrm{c}} v : T}{\Gamma \vdash_{\mathrm{c}} \mathtt{return}\ v : T\ !\ \emptyset}\ \textsc{ExReturn}$$

$$\frac{\Gamma \vdash_{\mathrm{c}} c_1 : T_1\ !\ \Delta \qquad \Gamma, x : T_1 \vdash_{\mathrm{c}} c_2 : T_2\ !\ \Delta}{\Gamma \vdash_{\mathrm{c}} \mathtt{do}\ x \leftarrow c_1; c_2 : T_2\ !\ \Delta}\ \textsc{ExDo}$$

$$\frac{\begin{array}{c}(\mathtt{Op} : T_1 \to T_2) \in \Sigma \\ \Gamma \vdash_{\mathrm{v}} v : T_1 \qquad \Gamma, y : T_2 \vdash_{\mathrm{c}} c : T\ !\ \Delta \qquad \mathtt{Op} \in \Delta\end{array}}{\Gamma \vdash_{\mathrm{c}} \mathtt{Op}\ v\ (y : T_2.c) : T\ !\ \Delta}\ \textsc{ExOp}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} v : \underline{C}_1 \Rrightarrow \underline{C}_2 \qquad \Gamma \vdash_{\mathrm{c}} c : \underline{C}_1}{\Gamma \vdash_{\mathrm{c}} \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{C}_2}\ \textsc{ExHandle}$$

$$\frac{\Gamma \vdash_{\mathrm{c}} c : \underline{C}_1 \qquad \Gamma \vdash_{\mathrm{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_{\mathrm{c}} c \rhd \gamma : \underline{C}_2}\ \textsc{ExCastC}$$

Figure 4.12: ExEff Computation Typing

4.4.2 Typing

Value & Computation Typing Typing for ExEff values and computations is presented in Figures 4.11 and 4.12 respectively and is given by two mutually recursive relations of the form $\Gamma \vdash_{\mathrm{v}} v : T$ (values) and $\Gamma \vdash_{\mathrm{c}} c : \underline{C}$ (computations). ExEff typing environments Γ contain bindings for variables of all sorts:

$$\Gamma ::= \epsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : T \mid \Gamma, \omega : \pi$$

Typing is entirely syntax-directed. Apart from the typing rules for skeleton, type, dirt, and coercion abstraction (and, subsequently, skeleton, type, dirt, and coercion application), the main difference between typing for ImpEff and ExEff lies in the explicit cast forms, $(v \rhd \gamma)$ and $(c \rhd \gamma)$. Given that a value v has type T_1 and that γ is a witness that T_1 is a subtype of T_2, we can upcast v with an explicit cast operation $(v \rhd \gamma)$. Upcasting for computations works analogously.

4.4.3 Well-formedness of Types, Constraints, Dirts & Skeletons for ExEff

The definitions of the judgements that express the well-formedness of ExEff value types ($\Gamma \vdash_T T : \tau$), computation types ($\Gamma \vdash_{\underline{C}} \underline{C} : \tau$), dirts ($\Gamma \vdash_{\underline{\Delta}} \underline{\Delta}$), and skeletons ($\Gamma \vdash_\tau \tau$) shown in Figure 4.13 are equally straightforward as those for ImpEff.

Coercion Typing Coercion typing formalises the intuitive interpretation of coercions that we gave in Section 4.4.1 and takes the form $\Gamma \vdash_{\text{co}} \gamma : \rho$. It is essentially an extension of the constraint entailment relation of Figure 4.4.

Figure 4.14 shows the typing rules for ExEff coercions. The extended rules in this figure deal with coercions for skeleton abstraction(VCoSkAbs) and instantiation(VCoSkInst) for polymorphic types. The rules also deal with types and dirts abstractions, and their instantiations. They also deal with coercions abstractions and instantiation.

4.4.4 Operational Semantics

Figures 4.15 and 4.16 present ExEff's small-step, call-by-value operational semantics for values and computations respectively.

One of the non-conventional features of our system is in the stratification of results in plain results and cast results:

$$
\begin{aligned}
\text{terminal value } v^T &::= () \mid h \mid \texttt{fun}\ x : T \mapsto c \mid \Lambda\alpha : \tau.v \mid \Lambda\delta.v \mid \Lambda\omega : \pi.v \\
\text{value result } v^R &::= v^T \mid v^T \rhd \gamma \\
\text{computation result } c^R &::= \texttt{return}\ v^T \mid (\texttt{return}\ v^T) \rhd \gamma \mid \texttt{Op}\ v^R\ (y : T.c)
\end{aligned}
$$

Terminal values v^T represent conventional values, and value results v^R can either be plain terminal values v^T or terminal values with a cast: $v^T \rhd \gamma$. The same applies to computation results c^R. Operation values do not feature an outermost cast operation, as the coercion can always be pushed into its continuation.

Although unusual, this stratification can also be found in Crary's coercion calculus for inclusive subtyping [16], and, more recently, in System F_C [88]. Stratification is crucial for ensuring type preservation. Consider for example the expression $(\texttt{return}\ 5 \rhd \langle \texttt{int} \rangle\,!\,\emptyset_{\{\texttt{Op}\}})$, of type $\texttt{int}\,!\,\{\texttt{Op}\}$. We can not reduce the expression further without losing effect information; removing the cast would result in

$\boxed{\Gamma \vdash_{\tau} T : \tau}$ **Value Types**

$$\frac{(\alpha : \tau) \in \Gamma}{\Gamma \vdash_{\tau} \alpha : \tau}\ \textsc{WfVTyVar} \qquad \frac{\Gamma \vdash_{\tau} T : \tau_1 \qquad \Gamma \vdash_{\underline{C}} \underline{C} : \tau_2}{\Gamma \vdash_{\tau} T \to \underline{C} : \tau_1 \to \tau_2}\ \textsc{WfVTyArr}$$

$$\frac{\Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau_1 \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau_2}{\Gamma \vdash_{\tau} \underline{C}_1 \Rrightarrow \underline{C}_2 : \tau_1 \Rrightarrow \tau_2}\ \textsc{WfVTyHandler}$$

$$\frac{}{\Gamma \vdash_{\tau} \mathtt{Unit} : \mathtt{Unit}}\ \textsc{WfVTyUnit} \qquad \frac{\Gamma \vdash_{\rho} \pi \qquad \Gamma \vdash_{\tau} T : \tau}{\Gamma \vdash_{\tau} \pi \Rightarrow T : \tau}\ \textsc{WfVTyCoAbs}$$

$$\frac{\Gamma, \varsigma \vdash_{\tau} T : \tau}{\Gamma \vdash_{\tau} \forall \varsigma . T : \forall \varsigma . \tau}\ \textsc{WfVTyCoAbs} \qquad \frac{\Gamma, \alpha : \tau_1 \vdash_{\tau} T : \tau_2}{\Gamma \vdash_{\tau} \forall \alpha : \tau_1 . T : \tau_2}\ \textsc{WfVTyTyAbs}$$

$$\frac{\Gamma, \delta \vdash_{\tau} T : \tau}{\Gamma \vdash_{\tau} \forall \delta . T : \tau}\ \textsc{WfVTyDirtAbs}$$

$\boxed{\Gamma \vdash_{\underline{C}} \underline{C} : \tau}$ **Computation Types**

$$\frac{\Gamma \vdash_{\tau} T : \tau \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\underline{C}} T \,!\, \Delta : \tau}\ \textsc{WfCTy}$$

$\boxed{\Gamma \vdash_{\rho} \rho}$ **Coercion Types**

$$\frac{\Gamma \vdash_{\tau} T_1 : \tau \qquad \Gamma \vdash_{\tau} T_2 : \tau}{\Gamma \vdash_{\rho} T_1 \leqslant T_2}\ \textsc{WfVTyCt}$$

$$\frac{\Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau}{\Gamma \vdash_{\rho} \underline{C}_1 \leqslant \underline{C}_2}\ \textsc{WfCTyCt} \qquad \frac{\Gamma \vdash_{\Delta} \Delta_1 \qquad \Gamma \vdash_{\Delta} \Delta_2}{\Gamma \vdash_{\rho} \Delta_1 \leqslant \Delta_2}\ \textsc{WfDTyCt}$$

$\boxed{\Gamma \vdash_{\Delta} \Delta}$ **Dirts**

$$\frac{}{\Gamma \vdash_{\Delta} \emptyset}\ \textsc{WfΔ Empty} \qquad \frac{\delta \in \Gamma}{\Gamma \vdash_{\Delta} \delta}\ \textsc{WfΔVar} \qquad \frac{(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\Delta} \{\mathtt{Op}\} \cup \Delta}\ \textsc{WfΔOp}$$

$\boxed{\Gamma \vdash_{\tau} \tau}$ **Skeletons**

$$\frac{\varsigma \in \Gamma}{\Gamma \vdash_{\tau} \varsigma} \qquad \frac{}{\Gamma \vdash_{\tau} \mathtt{Unit}} \qquad \frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \Rrightarrow \tau_2}$$

$$\frac{\Gamma, \varsigma \vdash_{\tau} \tau}{\Gamma \vdash_{\tau} \forall \varsigma . \tau}$$

Figure 4.13: ExEff Well-formedness of Types, Constraints, Dirts & Skeletons

Coercion Typing

$$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \rho}$$

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \omega : \pi}\ \textsc{VCoVar} \qquad \frac{\Gamma \vdash_T T : \tau}{\Gamma \vdash_{\mathsf{co}} \langle T \rangle : T \leqslant T}\ \textsc{VCoRefl} \qquad \frac{\Gamma \vdash_{\mathbb{A}} \Delta}{\Gamma \vdash_{\mathsf{co}} \langle \Delta \rangle : \Delta \leqslant \Delta}\ \textsc{DCoRefl}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_1 \leqslant T_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : T_2 \leqslant T_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : T_1 \leqslant T_3}\ \textsc{VCoTrans} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_1 \leqslant \underline{C}_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_2 \leqslant \underline{C}_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \underline{C}_1 \leqslant \underline{C}_3}\ \textsc{CCoTrans}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \Delta_1 \leqslant \Delta_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_2 \leqslant \Delta_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \Delta_1 \leqslant \Delta_3}\ \textsc{DCoTrans} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_2 \leqslant T_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \to \gamma_2 : T_1 \to \underline{C}_1 \leqslant T_2 \to \underline{C}_2}\ \textsc{VCoArr}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : T_1 \to \underline{C}_1 \leqslant T_2 \to \underline{C}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : T_2 \leqslant T_1}\ \textsc{VCoLeftArr} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : T_1 \to \underline{C}_1 \leqslant T_2 \to \underline{C}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{C}_1 \leqslant \underline{C}_2}\ \textsc{CCoRightArr}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_3 \leqslant \underline{C}_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_2 \leqslant \underline{C}_4}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \Rrightarrow \gamma_2 : \underline{C}_1 \Rrightarrow \underline{C}_2 \leqslant \underline{C}_3 \Rrightarrow \underline{C}_4}\ \textsc{VCoHandler}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{C}_2 \leqslant \underline{C}_3 \Rrightarrow \underline{C}_4}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : \underline{C}_3 \leqslant \underline{C}_1}\ \textsc{CCoLeftHandler}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{C}_2 \leqslant \underline{C}_3 \Rrightarrow \underline{C}_4}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{C}_2 \leqslant \underline{C}_4}\ \textsc{CCoRightHandler} \qquad \frac{\Gamma, \varsigma \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathsf{co}} \forall \varsigma.\gamma : \forall \varsigma. T_1 \leqslant \forall \varsigma. T_2}\ \textsc{VCoSkAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \forall \varsigma. T_1 \leqslant \forall \varsigma. T_2 \qquad \Gamma \vdash_\tau \tau}{\Gamma \vdash_{\mathsf{co}} \gamma[\tau] : T_1[\tau/\varsigma] \leqslant T_2[\tau/\varsigma]}\ \textsc{VCoSkInst}$$

$$\frac{\Gamma, \alpha : \tau \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathsf{co}} \forall \alpha : \tau.\gamma : \forall \alpha : \tau. T_1 \leqslant \forall \alpha : \tau. T_2}\ \textsc{VCoTyAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \forall \alpha : \tau. T_1 \leqslant \forall \alpha : \tau. T_2 \qquad \Gamma \vdash_T T : \tau}{\Gamma \vdash_{\mathsf{co}} \gamma[T] : T_1[T/\alpha] \leqslant T_2[T/\alpha]}\ \textsc{VCoTyInst}$$

$$\frac{\Gamma, \delta \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathsf{co}} \forall \delta.\gamma : \forall \delta. T_1 \leqslant \forall \delta. T_2}\ \textsc{VCoDirtAbs} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \forall \delta. T_1 \leqslant \forall \delta. T_2 \qquad \Gamma \vdash_{\mathbb{A}} \Delta}{\Gamma \vdash_{\mathsf{co}} \gamma[\Delta] : T_1[\Delta/\delta] \leqslant T_2[\Delta/\delta]}\ \textsc{VCoDirtInst}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2 \quad \Gamma \vdash_\rho \pi}{\Gamma \vdash_{\mathsf{co}} \pi \Rightarrow \gamma : \pi \Rightarrow T_1 \leqslant \pi \Rightarrow T_2}\ \textsc{VCoCoAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \pi \Rightarrow T_1 \leqslant \pi \Rightarrow T_2 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \pi}{\Gamma \vdash_{\mathsf{co}} \gamma_1 @ \gamma_2 : T_1 \leqslant T_2}\ \textsc{VCoCoInst}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_1 \leqslant T_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \,!\, \gamma_2 : T_1 \,!\, \Delta_1 \leqslant T_2 \,!\, \Delta_2}\ \textsc{CCoComp} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : T_1 \,!\, \Delta_1 \leqslant T_2 \,!\, \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{pure}(\gamma) : T_1 \leqslant T_2}\ \textsc{VCoPure}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : T_1 \,!\, \Delta_1 \leqslant T_2 \,!\, \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{impure}(\gamma) : \Delta_1 \leqslant \Delta_2}\ \textsc{DCoImpure} \qquad \frac{\Gamma \vdash_{\mathbb{A}} \Delta}{\Gamma \vdash_{\mathsf{co}} \emptyset_\Delta : \emptyset \leqslant \Delta}\ \textsc{DCoEmpty}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \Delta_1 \leqslant \Delta_2 \qquad (\mathtt{Op} : T_1 \to T_2) \in \Sigma}{\Gamma \vdash_{\mathsf{co}} \{\mathtt{Op}\} \cup \gamma : \{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2}\ \textsc{DCoOp}$$

Figure 4.14: ExEff Coercion Typing

Values

$$\boxed{v \rightsquigarrow_{\mathrm{v}} v'}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v \rhd \gamma \rightsquigarrow_{\mathrm{v}} v' \rhd \gamma}\ (\textsc{VCast}) \qquad (v^T \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{v}} v^T \rhd (\gamma_1 \gg \gamma_2)\ (\textsc{VPushTrans})$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v\ T \rightsquigarrow_{\mathrm{v}} v'\ T}\ (\textsc{VTyApp}) \qquad \frac{v \rightsquigarrow_{\mathrm{v}} v'}{v\ \Delta \rightsquigarrow_{\mathrm{v}} v'\ \Delta}\ (\textsc{VDirtApp})$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v\ \gamma \rightsquigarrow_{\mathrm{v}} v'\ \gamma}\ (\textsc{VCoerApp}) \qquad (\Lambda\alpha : \tau.v)\ T \rightsquigarrow_{\mathrm{v}} v[T/\alpha]\ (\textsc{VBredTy})$$

$$(\Lambda\delta.v)\ \Delta \rightsquigarrow_{\mathrm{v}} v[\Delta/\delta]\ (\textsc{VBredDirt}) \qquad (\Lambda(\omega : \pi).v)\ \gamma \rightsquigarrow_{\mathrm{v}} v[\gamma/\omega]\ (\textsc{VBredCoer})$$

$$(v^T \rhd \gamma)\ T \rightsquigarrow_{\mathrm{v}} (v^T\ T) \rhd \gamma[T]\ (\textsc{VPushTyapp})$$

$$(v^T \rhd \gamma)\ \Delta \rightsquigarrow_{\mathrm{v}} (v^T\ \Delta) \rhd \gamma[\Delta]\ (\textsc{VPushDirtApp})$$

$$(v^T \rhd \gamma_1)\ \gamma_2 \rightsquigarrow_{\mathrm{v}} (v^T\ \gamma_2) \rhd \gamma_1 @ \gamma_2\ (\textsc{VPushCoer})$$

Figure 4.15: ExEff Operational Semantics for Values

computation (`return` 5), of type `int` $!\,\emptyset$. Even if we consider type preservation only up to subtyping, the redex may still occur as a subterm in a context that expects solely the larger type.

We also need to make sure that casts do not stand in the way of evaluation. This is captured in the so-called "push" rules, all of which appear in the two figures.

Operational semantics for Values Figure 4.15 shows the relation $v \rightsquigarrow_{\mathrm{v}} v'$ that evaluates values. The rule (VCast) evaluates a value inside a cast. The rule (VpushTrans) groups nested casts into a single cast, by means of transitivity. The rules (VtyApp), (VdirtApp) and (VCoerApp) are the same as (VCast) but for types, dirts and coercions respectively.

The rules (VBredTy), (VBredDirt) and (VBredCoer) beta reduces an application by substitution of the applied type, dirt or coercion respectively into the value v itself. The last rules capture the essence of push rules: whenever a redex is "blocked" due to a cast, we take the coercion apart and redistribute it (in a type-preserving manner) over the subterms, so that evaluation can proceed.

Computations

$$\boxed{c \rightsquigarrow_{\mathrm{c}} c'}$$

$$\frac{c \rightsquigarrow_{\mathrm{c}} c'}{c \rhd \gamma \rightsquigarrow_{\mathrm{c}} c' \rhd \gamma}\ \textsc{CCast} \qquad (c^R \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{c}} c^R \rhd (\gamma_1 \gg \gamma_2)\ \textsc{CPushTrans}$$

$$\frac{v_1 \rightsquigarrow_{\mathrm{v}} v_1'}{v_1\ v_2 \rightsquigarrow_{\mathrm{c}} v_1'\ v_2}\ \textsc{CVarApp}$$

$$(v_1^T \rhd \gamma)\ v_2 \rightsquigarrow_{\mathrm{c}} (v_1^T\ (v_2 \rhd \mathit{left}(\gamma))) \rhd \mathit{right}(\gamma)\ \textsc{CPushApp}$$

$$\frac{v_2 \rightsquigarrow_{\mathrm{v}} v_2'}{v_1^T\ v_2 \rightsquigarrow_{\mathrm{c}} v_1^T\ v_2'}\ \textsc{CTermValApp} \qquad (\mathtt{fun}\ (x : T) \mapsto c)\ v^R \rightsquigarrow_{\mathrm{c}} c[v^R/x]\ \textsc{CFunBred}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\mathtt{let}\ x = v\ \mathtt{in}\ c \rightsquigarrow_{\mathrm{c}} \mathtt{let}\ x = v'\ \mathtt{in}\ c}\ \textsc{CLetValApp}$$

$$\mathtt{let}\ x = v^R\ \mathtt{in}\ c \rightsquigarrow_{\mathrm{c}} c[v^R/x]\ \textsc{CLetValBred} \qquad \frac{v \rightsquigarrow_{\mathrm{v}} v'}{\mathtt{return}\ v \rightsquigarrow_{\mathrm{c}} \mathtt{return}\ v'}\ \textsc{CReturnVal}$$

$$\mathtt{return}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{return}\ v^T) \rhd (\gamma\ !\ \emptyset_{\emptyset})\ \textsc{CEmptyDirt}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\mathtt{Op}\ v\ (y : T.c) \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v'\ (y : T.c)}\ \textsc{COpVal}$$

$$(\mathtt{Op}\ v^R\ (y : T.c)) \rhd \gamma \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.(c \rhd \gamma))\ \textsc{CPushOpCoer}$$

$$\frac{c_1 \rightsquigarrow_{\mathrm{c}} c_1'}{\mathtt{do}\ x \leftarrow c_1; c_2 \rightsquigarrow_{\mathrm{c}} \mathtt{do}\ x \leftarrow c_1'; c_2}\ \textsc{CDocRed}$$

$$\mathtt{do}\ x \leftarrow ((\mathtt{return}\ v^T) \rhd \gamma); c_2 \rightsquigarrow_{\mathrm{c}} c_2[(v^T \rhd \mathit{pure}(\gamma))/x]\ \textsc{CDoRet}$$

$$\mathtt{do}\ x \leftarrow \mathtt{Op}\ v^R\ (y : T.c_1); c_2 \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.\mathtt{do}\ x \leftarrow c_1; c_2)\ \textsc{CDoOp}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\mathtt{handle}\ c\ \mathtt{with}\ v \rightsquigarrow_{\mathrm{c}} \mathtt{handle}\ c\ \mathtt{with}\ v'}\ \textsc{CHandleVal}$$

$$\mathtt{handle}\ c\ \mathtt{with}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{handle}\ (c \rhd \mathit{left}(\gamma))\ \mathtt{with}\ v^T) \rhd \mathit{right}(\gamma)\ \textsc{CHandlePush}$$

$$\frac{c \rightsquigarrow_{\mathrm{c}} c'}{\mathtt{handle}\ c\ \mathtt{with}\ v^T \rightsquigarrow_{\mathrm{c}} \mathtt{handle}\ c'\ \mathtt{with}\ v^T}\ \textsc{CHandleComp}$$

$$\mathtt{handle}\ ((\mathtt{return}\ v^T) \rhd \gamma)\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} c_r[v^T \rhd \mathit{pure}(\gamma)/x]\ \textsc{CHandleRet}$$

$$\mathtt{handle}\ (\mathtt{Op}\ v^R\ (y : T.c))\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} \begin{array}{c} c_{\mathtt{Op}}[v^R/x, (\mathtt{fun}\ (y : T) \mapsto \\ \mathtt{handle}\ c\ \mathtt{with}\ h)/k] \end{array}\ \textsc{CHandleOp}$$

$$\mathtt{handle}\ (\mathtt{Op}\ v^R\ (y : T.c))\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.\mathtt{handle}\ c\ \mathtt{with}\ h)\ \textsc{CHandleNoOp}$$

Figure 4.16: ExEff Operational Semantics For Computations

Operational semantics for Computations Figure 4.16 shows the relation $c \rightsquigarrow_{\mathrm{v}} c'$ that evaluates computations.

The rule CCast evaluates a computation inside a cast. The rule CPushTrans uses transitivity to group nested casts into a single cast. CVarApp reduces the function of the application before executing the application itself. CPushApp is the push rule for β-reduction, it pushes the left part of the coercion into the input value of the function and pushes the right part of the coercion outside to coerce the result of the application. The rule CTermValApp reduces the input value in an application, only after the function is already reduced to a terminal value (after applying CVarApp multiple times). The rule CFunBred beta reduces an application.

The rule CLetValApp reduces the value in the `Let..in..` computation and then CLetValBred beta-reduces the whole computation. The rule CReturnVal reduces the value inside the `return`-computation and CEmptyDirt pushes a cast out of a `return`-computation and applies an empty dirt coercion to it.

The rule COpVal reduces the value inside an operation call and CPushOpCoer pushes the outer cast of an operation inside the operation's computation, illustrating why the syntax for c^R does not require casts on operation-computations. The rule CDocRed reduces the first computation in sequencing. CDocRet is a push rule for sequencing computations and performs two tasks at once. Since we know that the computation bound to x calls no operations, we (a) safely "drop" the impure part of γ, and (b) substitute x with v^T, cast with the pure part of γ (so that types are preserved). CDoOp handles operation calls in sequencing computations. If an operation is called in a sequencing computation, evaluation is suspended and the rest of the computation is captured in the continuation.

The last six rules are concerned with handlers. CHandleVal and CHandleComp reduce the handler and the computation the handler is handling respectively. CHandlePush pushes a coercion on the handler "outwards", so that the handler can be exposed and evaluation is not stuck (similarly to the push rule for term application). CHandleRet behaves similarly to the push/beta rule for sequencing computations. CHandleOp and CHandleNoOp are concerned with handling of operations. The first captures cases where the called operation is handled by the handler, in which case the respective clause of the handler is called. The second rule captures cases where the operation is not covered by the handler and thus remains unhandled.

Type safety of ExEff ExEff is type safe:

Theorem 4 (Type Safety)**.** ▪ *If* $\Gamma \vdash_{\mathrm{v}} v : T$ *then either* v *is a result value or* $v \rightsquigarrow_{\mathrm{v}} v'$ *and* $\Gamma \vdash_{\mathrm{v}} v' : T$.

- *If* $\Gamma \vdash_{\mathrm{c}} c : \underline{C}$ *then either* c *is a result computation or* $c \rightsquigarrow_{\mathrm{c}} c'$ *and* $\Gamma \vdash_{\mathrm{c}} c' : \underline{C}$.

4.5 Type Inference & Elaboration

This section presents the typing-directed elaboration of ImpEff into ExEff. This elaboration makes all the implicit type and effect information explicit, and introduces explicit term-level coercions to witness the use of subtyping.

After covering the declarative specification of this elaboration, we present a constraint-based algorithm to infer ImpEff types and at the same time elaborate into ExEff. This algorithm alternates between two phases: 1) the syntax-directed generation of constraints from the ImpEff term, and 2) solving these constraints.

4.5.1 Elaboration of ImpEff into ExEff

The grayed parts of Figures 4.2 and 4.3 augment the typing rules for ImpEff value and computation terms with typing-directed elaboration to corresponding ExEff terms. The elaboration is mostly straightforward, mapping every ImpEff construct onto its corresponding ExEff construct while adding explicit type annotations to binders in Rules TmTmAbs, TmHandler and TmOp. Implicit appeals to subtyping are turned into explicit casts with coercions in Rules TmCastV and TmCastC. Rule TmLet introduces explicit binders for skeleton, type, and dirt variables, as well as for constraints. These last also introduce coercion variables ω that can be used in casts. The binders are eliminated in rule TmVar by means of explicit application with skeletons, types, dirts and coercions. The coercions are produced by the auxiliary judgement $\Gamma \vdash_{\mathrm{co}} \gamma : \pi$, defined in Figure 4.4, which provides a coercion witness for every subtyping proof.

We have also shown that elaboration preserves types.

Theorem 5 (Type Preservation).

- *If* $\Gamma \vdash_{v} v : A \rightsquigarrow v'$ *then* $elab_{\Gamma}(\Gamma) \vdash_{\mathrm{v}} v' : elab_{s}(A)$.
- *If* $\Gamma \vdash_{c} c : \underline{C} \rightsquigarrow c'$ *then* $elab_{\Gamma}(\Gamma) \vdash_{\mathrm{c}} c' : elab_{\underline{c}}(\underline{C})$.

Here $elab_{\Gamma}(\Gamma)$, $elab_{s}(A)$ and $elab_{\underline{c}}(\underline{C})$ convert ImpEff environments and types into their ExEff counterparts. All four functions are entirely straightforward and essentially traverse each sort, so that ImpEff value types A are replaced with ExEff value types T, as shown in Figure 4.17.

$$
\begin{array}{lcl}
elab_S(\alpha) & = & \alpha \\
elab_S(A \to \underline{C}) & = & elab_S(A) \to elab_{\underline{C}}(\underline{C}) \\
elab_S(\underline{C} \Rrightarrow \underline{D}) & = & elab_{\underline{C}}(\underline{C}) \to elab_{\underline{C}}(\underline{D}) \\
elab_S(\texttt{Unit}) & = & \texttt{Unit} \\
elab_S(\forall \varsigma.S) & = & \forall \varsigma.elab_S(S) \\
elab_S(\forall \alpha : \tau.S) & = & \forall \alpha : \tau.elab_S(S) \\
elab_S(\forall \delta.S) & = & \forall \delta.elab_S(S) \\
elab_S(\pi \Rightarrow K) & = & elab_\rho(\pi) \Rightarrow elab_S(K)
\end{array}
\qquad
\begin{array}{lcl}
elab_{\underline{C}}(A \,!\, \Delta) & = & elab_S(A) \,!\, \Delta \\
elab_\Gamma(\epsilon) & = & \epsilon \\
elab_\Gamma(\Gamma, \varsigma) & = & elab_\Gamma(\Gamma), \varsigma \\
elab_\Gamma(\Gamma, \alpha : \tau) & = & elab_\Gamma(\Gamma), \alpha : \tau \\
elab_\Gamma(\Gamma, \delta) & = & elab_\Gamma(\Gamma), \delta \\
elab_\Gamma(\Gamma, x : S) & = & elab_\Gamma(\Gamma), x : elab_S(S) \\
elab_\Gamma(\Gamma, \omega : \rho) & = & elab_\Gamma(\Gamma), \omega : elab_\rho(\rho)
\end{array}
$$

$$
\begin{array}{lcl}
elab_\rho(A \leqslant B) & = & elab_S(A) \leqslant elab_S(B) \\
elab_\rho(\underline{C} \leqslant \underline{D}) & = & elab_{\underline{C}}(\underline{C}) \leqslant elab_{\underline{C}}(\underline{D}) \\
elab_\rho(\Delta_1 \leqslant \Delta_2) & = & \Delta_1 \leqslant \Delta_2
\end{array}
$$

Figure 4.17: Elaboration for value & computation types, constraints, and typing environments.

4.5.2 Constraint Generation & Elaboration

Constraint generation with elaboration into ExEff is presented in Figures 4.18 (values) and 4.19 (computations). Before going into the details of each, we first introduce the three auxiliary constructs they use.

$$
\begin{array}{rl}
\text{constraint set } \mathcal{P}, \mathcal{Q} & ::= \bullet \mid \tau_1 = \tau_2, \mathcal{P} \mid \alpha : \tau, \mathcal{P} \mid \omega : \pi, \mathcal{P} \\
\text{typing environment } \Gamma & ::= \epsilon \mid \Gamma, x : S \\
\text{substitution } \sigma & ::= \bullet \mid \sigma \cdot [\tau/\varsigma] \mid \sigma \cdot [A/\alpha] \mid \sigma \cdot [\Delta/\delta] \mid \sigma \cdot [\gamma/\omega]
\end{array}
$$

At the heart of our algorithm are sets $\mathcal{P}$, containing three different kinds of constraints: (a) skeleton equalities of the form $\tau_1 = \tau_2$, (b) skeleton constraints of the form $\alpha : \tau$, and (c) wanted subtyping constraints of the form $\omega : \pi$. The purpose of the first two becomes clear when we discuss constraint solving, in Section 4.5.3. Next, typing environments Γ only contain term variable bindings, while other variables represent unknowns of their sort and may end up being instantiated after constraint solving. Finally, during type inference we compute substitutions σ, for refining as of yet unknown skeletons, types, dirts, and coercions. The last one is essential, since our algorithm simultaneously performs type inference and elaboration into ExEff.

Values

$$\boxed{\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'}$$

GVar

$$\frac{(x : \forall \overline{\varsigma}.\forall \overline{\alpha : \tau}.\forall \overline{\delta}.\overline{\pi} \Rightarrow A) \in \Gamma \qquad \sigma = [\overline{\varsigma'/\varsigma}, \overline{\alpha'/\alpha}, \overline{\delta'/\delta}]}{\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} x : \sigma(A) \mid \overline{\omega : \sigma(\pi)}, \overline{\alpha' : \sigma(\tau)}, \mathcal{Q}; \bullet \rightsquigarrow x\ \overline{\varsigma'}\ \overline{\alpha'}\ \overline{\delta'}\ \overline{\omega}}$$

GUnit

$$\frac{}{\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} () : \mathtt{Unit} \mid \mathcal{Q}; \bullet \rightsquigarrow ()}$$

GAbs

$$\frac{\alpha : \varsigma, \mathcal{Q}; \Gamma, x : \alpha \vdash_{\mathrm{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'}{\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} (\mathtt{fun}\ x \mapsto c) : \sigma(\alpha) \to \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow \mathtt{fun}\ x : \sigma(\alpha) \mapsto c'}$$

GHand

$$\frac{\begin{array}{l}\alpha_r : \varsigma_r, \mathcal{Q}; \Gamma, x : \alpha_r \vdash_{\mathrm{c}} c_r : B_r\ !\ \Delta_r \mid \mathcal{Q}_0; \sigma_r \rightsquigarrow c'_r \qquad \sigma^i = \sigma_i \cdot \sigma_{i-1} \cdot \ldots \cdot \sigma_1 \\ \quad \mathtt{Op}_i \in \mathcal{O} : \\ \quad\quad (\mathtt{Op}_i : A_i \to B_i) \in \Sigma \\ \quad\quad \alpha_i : \varsigma_i, \mathcal{Q}_{i-1}; \sigma^{i-1}(\sigma_r(\Gamma)), x : A_i, k : B_i \to \alpha_i\ !\ \delta_i \vdash_{\mathrm{v}} \\ \quad\quad\quad c_{\mathtt{Op}_i} : B_{\mathtt{Op}_i}\ !\ \Delta_{\mathtt{Op}_i} \mid \mathcal{Q}_i; \sigma_i \rightsquigarrow c'_{\mathtt{Op}_i} \\ \quad \mathcal{Q}' = \alpha_{in} : \varsigma_{in}, \alpha_{out} : \varsigma_{out}, \overline{\omega_1 : \sigma^n(B_r) \leqslant \alpha_{out}}, \overline{\omega_2 : \sigma^n(\Delta_r) \leqslant \delta_{out}}^{n}, \\ \quad\quad \overline{\omega_{3_i} : \sigma^n(B_{\mathtt{Op}_i}) \leqslant \alpha_{out}}^{n}, \overline{\omega_{4_i} : \sigma^n(\Delta_{\mathtt{Op}_i}) \leqslant \delta_{out}}^{n}, \\ \quad\quad \overline{\omega_{5_i} : B_i \to \alpha_{out}\ !\ \delta_{out} \leqslant B_i \to \sigma^n(\alpha_i\ !\ \delta_i)}^{n}, \\ \quad\quad \omega_6 : \alpha_{in} \leqslant \sigma^n(\sigma_r(\alpha_r)), \omega_7 : \delta_{in} \leqslant \delta_{out} \cup \mathcal{O}, \mathcal{Q}_n \\ \quad\quad c_{res} = \{\, \mathtt{return}\ y : \sigma^n(\sigma_r(\alpha_r)) \mapsto \sigma^n(c'_r)[y \rhd \omega_6/x] \rhd \omega_1\ !\ \omega_2 \\ \quad\quad\quad , \left[\mathtt{Op}_i\ x\ l \mapsto \sigma^n(c'_{\mathtt{Op}_i})[l \rhd \omega_{5_i}/k] \rhd \omega_{3_i}\ !\ \omega_{4_i}\right]_{\mathtt{Op}_i \in \mathcal{O}} \} \\ \quad\quad\quad \rhd (\langle \alpha_{in} \rangle\ !\ \omega_7 \Rrightarrow \langle \alpha_{out} \rangle\ !\ \langle \delta_{out} \rangle)\end{array}}{\begin{array}{c}\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ x\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : \\ \alpha_{in}\ !\ \delta_{in} \Rrightarrow \alpha_{out}\ !\ \delta_{out} \mid \mathcal{Q}'; (\sigma^n \cdot \sigma_r) \rightsquigarrow c_{res}\end{array}}$$

Figure 4.18: Constraint Generation with Elaboration (Values)

A substitution σ is a solution of the set $\mathcal{P}$, written as $\sigma \models \mathcal{P}$, if we get derivable judgements after applying σ to all constraints in $\mathcal{P}$.

Values. Constraint generation for values takes the form $\mathcal{Q}; \Gamma \vdash_{\mathrm{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'$. It takes as inputs a set of wanted constraints $\mathcal{Q}$, a typing environment Γ, and a ImpEff value v, and produces a value type A, a new set of wanted constraints $\mathcal{Q}'$, a substitution σ, and a ExEff value v'.

Unlike standard Hindley-Milner inference algorithm, our inference algorithm does not keep constraint generation and solving separate. Instead, the two are interleaved, as indicated by the additional arguments of our relation: (a) constraints $\mathcal{Q}$ are passed around in a stateful manner (i.e., they are input and output), and (b) substitutions σ generated from constraint solving constitute part of the relation's output. We discuss the reason for this interleaved approach in Section 4.5.4; we now focus on the algorithm.

The rules are syntax-directed on the input ImpEff value. Rule GVar handles term variables x: as usual for constraint-based type inference the rule instantiates the polymorphic type $(\forall \bar{\varsigma}.\overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow A)$ of x with fresh variables; these are placeholders that are determined during constraint solving. Moreover, the rule extends the wanted constraints $\mathcal{P}$ with $\bar{\pi}$, appropriately instantiated. In ExEff, this corresponds to explicit skeleton, type, dirt, and coercion applications. The rule GUnit elaborates ImpEff's $()$ to ExEff and does not generate any new constraints and returns an empty substitution.

More interesting is the GAbs, for term abstractions. Like in standard Hindley-Milner [18], it generates a fresh type variable α for the type of the abstracted term variable x. In addition, it generates a fresh skeleton variable ς, to capture the (yet unknown) shape of α.

As explained in detail in Section 4.5.3, the constraint solver instantiates type variables only through their skeleton annotations, meaning that the shape of the skeleton variable defines the shape of the type variable. Because we want to allow local constraint solving for the body c of the term abstraction the opportunity to produce a substitution σ that instantiates α, we have to pass in the annotation constraint $\alpha : \varsigma$.[1] We apply the resulting substitution σ to the result type $\sigma(\alpha) \to \underline{C}$. Notice that though σ refers to ImpEff types, we abuse notation to save clutter and apply it directly to ExEff entities too.

Finally, the GHand is concerned with handlers. Since it is the most complex of the rules, we discuss each of its premises separately:

Firstly, we infer a type $B_r \,!\, \Delta_r$ for the right hand side of the `return`-clause. Since α_r is a fresh unification variable, just like for term abstraction we require $\alpha_r : \varsigma_r$, for a fresh skeleton variable ς_r.

Secondly, we check every operation clause in $\mathcal{O}$ in order. For each clause, we

[1] This hints at why we need to pass constraints in a stateful manner.

generate fresh skeleton, type, and dirt variables (ς_i, α_i, and δ_i), to account for the (yet unknown) result type $\alpha_i \,!\, \delta_i$ of the continuation k, while inferring type $B_{\mathtt{Op}_i} \,!\, \Delta_{\mathtt{Op}_i}$ for the right-hand-side $c_{\mathtt{Op}_i}$.

More interesting is the (final) set of wanted constraints $\mathcal{Q}'$. First, we assign to the handler the overall type

$$\alpha_{in} \,!\, \delta_{in} \Rrightarrow \alpha_{out} \,!\, \delta_{out}$$

where $\varsigma_{in}, \alpha_{in}, \delta_{in}, \varsigma_{out}, \alpha_{out}, \delta_{out}$ are fresh variables of the respective sorts. In turn, we require that (a) the type of the return clause is a subtype of $\alpha_{out} \,!\, \delta_{out}$ (given by the combination of ω_1 and ω_2), (b) the right-hand-side type of each operation clause is a subtype of the overall result type: $\sigma^n(B_{\mathtt{Op}_i} \,!\, \Delta_{\mathtt{Op}_i}) \leqslant \alpha_{out} \,!\, \delta_{out}$ (witnessed by $\omega_{3_i} \,!\, \omega_{4_i}$), (c) the actual types of the continuations $B_i \to \alpha_{out} \,!\, \delta_{out}$ in the operation clauses should be subtypes of their assumed types $B_i \to \sigma^n(\alpha_i \,!\, \delta_i)$ (witnessed by ω_{5_i}). (d) the overall argument type α_{in} is a subtype of the assumed type of x: $\sigma^n(\sigma_r(\alpha_r))$ (witnessed by ω_6), and (e) the input dirt set δ_{in} is a subtype of the resulting dirt set δ_{out}, extended with the handled operations $\mathcal{O}$ (witnessed by ω_7).

All the aforementioned implicit subtyping relations become explicit in the elaborated term c_{res}, via explicit casts.

Computations. The judgement $\mathcal{Q}; \Gamma \vdash_{\mathrm{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'$ generates constraints for computations.

GApp rule handles term applications of the form $v_1\, v_2$. After inferring a type for each subterm (A_1 for v_1 and A_2 for v_2), we generate the wanted constraint $\sigma_2(A_1) \leqslant A_2 \to \alpha \,!\, \delta$, with fresh type and dirt variables α and δ, respectively. Associated coercion variable ω is then used in the elaborated term to explicitly (up)cast v_1' to the expected type $A_2 \to \alpha \,!\, \delta$. The rule GReturn is entirely straightforward as it evaluates the value inside the `return` computation and returns the resulted constraints and the substitution.

The rule GLet handles polymorphic let-bindings. First, we infer a type A for v, as well as wanted constraints $\mathcal{Q}_v$. Then, we simplify wanted constraints $\mathcal{Q}_v$ by means of function `solve` (which we explain in detail in Section 4.5.3 below), obtaining a substitution σ_1' and a set of *residual constraints* $\mathcal{Q}_v'$.

Generalisation of x's type is performed by auxiliary function $split$, given by the following clause:

$$\frac{\begin{array}{c}\bar{\varsigma} = \{\varsigma \mid (\alpha : \varsigma) \in \mathcal{Q}, \nexists \alpha'.\alpha' \notin \bar{\alpha} \wedge (\alpha' : \varsigma) \in \mathcal{Q}\} \\ \bar{\alpha} = fv_\alpha(\mathcal{Q}) \cup fv_\alpha(A) \setminus fv_\alpha(\Gamma) \qquad \mathcal{Q}_1 = \{(\omega : \pi) \mid (\omega : \pi) \in \mathcal{Q}, fv(\pi) \not\subseteq fv(\Gamma)\} \\ \bar{\delta} = fv_\delta(\mathcal{Q}) \cup fv_\delta(A) \setminus fv_\delta(\Gamma) \qquad \mathcal{Q}_2 = \mathcal{Q} - \mathcal{Q}_1\end{array}}{split(\Gamma, \mathcal{Q}, A) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \mathcal{Q}_1, \mathcal{Q}_2 \rangle}$$

Computations

$$\boxed{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}';\sigma \rightsquigarrow c'}$$

GApp

$$\frac{\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v_1 : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v_1' \qquad \mathcal{Q}_1;\sigma_1(\Gamma) \vdash_{\mathsf{v}} v_2 : A_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow v_2'}{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{c}} v_1\ v_2 : \alpha\,!\,\delta \mid \alpha : \varsigma, \omega : \sigma_2(A_1) \leqslant A_2 \to \alpha\,!\,\delta, \mathcal{Q}_2;(\sigma_2 \cdot \sigma_1) \\ \rightsquigarrow (\sigma_2(v_1') \rhd \omega)\ v_2'\end{array}}$$

GReturn

$$\frac{\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}';\sigma \rightsquigarrow v'}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{return}\ v : A\ !\ \emptyset \mid \mathcal{Q}';\sigma \rightsquigarrow \mathtt{return}\ v'}$$

GLet

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}_v;\sigma_1 \rightsquigarrow v' \qquad \mathtt{solve}(\bullet;\bullet;\mathcal{Q}_v) = (\sigma_1', \mathcal{Q}_v') \\ \mathit{split}(\sigma_1'(\sigma_1(\Gamma)), \mathcal{Q}_v', \sigma_1'(A)) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \overline{\omega : \pi}, \mathcal{Q}_1 \rangle \\ \mathcal{Q}_1;\sigma_1'(\sigma_1(\Gamma)), x : \forall \bar{\varsigma}.\forall \overline{\alpha : \tau}.\forall \bar{\delta}.\overline{\pi} \Rightarrow \sigma_1'(A) \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ c_{res} = \mathtt{let}\ x = \sigma_2(\Lambda\bar{\varsigma}.\Lambda\overline{\alpha : \tau}.\Lambda\bar{\delta}.\Lambda\overline{(\omega : \mathit{elab}_{\rho}(\pi))}.v')\ \mathtt{in}\ c'\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C} \mid \mathcal{Q}_2;(\sigma_2 \cdot \sigma_1' \cdot \sigma_1) \rightsquigarrow c_{res}}$$

GOp

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v' \qquad \mathcal{Q}_1;\sigma_1(\Gamma), y : B_{\mathtt{Op}} \vdash_{\mathsf{c}} c : A_2\ !\ \Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad c_{res} = \mathtt{Op}\ (\sigma_2(v') \rhd \omega)\ (y : \mathit{elab}_s(B_{\mathtt{Op}}).c')\end{array}}{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{Op}\ v\ (y : B_{\mathtt{Op}}.c) : A_2\ !\ \{\mathtt{Op}\} \cup \Delta_2 \mid \omega : \sigma_2(A_1) \leqslant A_{\mathtt{Op}}, \mathcal{Q}_2;(\sigma_2 \cdot \sigma_1) \\ \rightsquigarrow c_{res}\end{array}}$$

GDo

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{c}} c_1 : A_1\,!\,\Delta_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow c_1' \\ \mathcal{Q}_1;\sigma_1(\Gamma), x : A_1 \vdash_{\mathsf{c}} c_2 : A_2\,!\,\Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c_2' \\ c_{res} = \mathtt{do}\ x \leftarrow (\sigma_2(c_1') \rhd \langle \sigma_2(A_1) \rangle\,!\,\omega_1); (c_2' \rhd \langle A_2 \rangle\,!\,\omega_2)\end{array}}{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{do}\ x \leftarrow c_1; c_2 : A_2\ !\ \delta \mid \omega_1 : \sigma_2(\Delta_1) \leqslant \delta, \omega_2 : \Delta_2 \leqslant \delta, \mathcal{Q}_2;(\sigma_2 \cdot \sigma_1) \\ \rightsquigarrow c_{res}\end{array}}$$

GHandle

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v' \qquad \mathcal{Q}_1;\sigma_1(\Gamma) \vdash_{\mathsf{c}} c : A_2\,!\,\Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ \mathcal{Q}' = \quad \alpha_1 : \varsigma_1, \alpha_2 : \varsigma_2, \omega_1 : \sigma_2(A_1) \leqslant (\alpha_1\,!\,\delta_1 \Rrightarrow \alpha_2\,!\,\delta_2), \\ \omega_2 : A_2 \leqslant \alpha_1, \omega_3 : \Delta_2 \leqslant \delta_1, \mathcal{Q}_2 \\ c_{res} = \mathtt{handle}\ (c' \rhd (\omega_2\ !\ \omega_3))\ \mathtt{with}\ (\sigma_2(v') \rhd \omega_1)\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{handle}\ c\ \mathtt{with}\ v : \alpha_2\ !\ \Delta_2 \mid \mathcal{Q}';(\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}}$$

Figure 4.19: Constraint Generation with Elaboration (Computations)

In essence, *split* generates the type (scheme) of x in parts. Additionally, it computes the subset $\mathcal{Q}_2$ of the input constraints $\mathcal{Q}$ that do not depend on locally-bound variables. Such constraints can be floated "upwards", and are passed as input when inferring a type for c. The set $\bar{\alpha}$ contains the set of type variables that locally appear as free variables in $\mathcal{Q}$ and the input type A. This set is used to extract the set of skeletons $\bar{\varsigma}$ which contains the skeletons needed for the generalisation.

The rule $\textsc{Op}$ handles operation calls. Observe that in the elaborated term, we upcast the inferred type to match the expected type in the signature. $\textsc{Do}$ handles sequences. The requirement that all computations in a `do`-construct have the same dirt set is expressed in the wanted constraints $\sigma_2(\Delta_1) \leqslant \delta$ and $\Delta_2 \leqslant \delta$ (where δ is a fresh dirt variable; the resulting dirt set), witnessed by coercion variables ω_1 and ω_2. Both coercion variables are used in the elaborated term to upcast c_1 and c_2, such that both draw effects from the same dirt set δ.

Finally, $\textsc{Handle}$ is concerned with effect handling. After inferring type A_1 for the handler v, we require that it takes the form of a handler type, witnessed by coercion variable $\omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 \,!\, \delta_1 \Rrightarrow \alpha_2 \,!\, \delta_2)$, for fresh $\alpha_1, \alpha_2, \delta_1, \delta_2$. To ensure that the type $A_2 \,!\, \Delta_2$ of c matches the expected type, we require that $A_2 \,!\, \Delta_2 \leqslant \alpha_1 \,!\, \delta_1$. Our syntax does not include coercion variables for computation subtyping; we achieve the same effect by combining $\omega_2 : A_2 \leqslant \alpha_1$ and $\omega_3 : \Delta_2 \leqslant \delta_1$.

The soundenss and completeness of the inference elaboration are proved in the following two theorms:

Theorem 6 (Soundness of Inference)**.** *If* $\bullet; \Gamma \vdash_{\text{v}} v : A \mid \mathcal{Q}; \sigma \rightsquigarrow v'$ *then for any* $\sigma' \models \mathcal{Q}$*, we have* $(\sigma' \cdot \sigma)(\Gamma) \vdash_v v : \sigma'(A) \rightsquigarrow \sigma'(v')$*, and analogously for computations.*

Theorem 7. *[Completeness of Inference] If* $\Gamma \vdash_v v : A \rightsquigarrow v'$ *then we have* $\bullet; \Gamma \vdash_{\text{v}} v : A' \mid \mathcal{Q}; \sigma \rightsquigarrow v''$ *and there exists* $\sigma' \models \mathcal{Q}$ *and* γ*, such that* $\sigma'(v'') = v'$ *and* $\Gamma \vdash_{\text{co}} \gamma : \sigma'(A') \leqslant A$*. An analogous statement holds for computations.*

4.5.3 Constraint Solving

The second phase of our inference-and-elaboration algorithm is the constraint solver. It is defined by the `solve` function signature:

$$\boxed{\texttt{solve}(\sigma;\, \mathcal{P};\, \mathcal{Q}) = (\sigma',\, \mathcal{P}')}$$

It takes three inputs: the substitution σ accumulated so far, a list of already processed constraints $\mathcal{P}$, and a queue of still to be processed constraints $\mathcal{Q}$. There are two outputs: the substitution σ' that solves the constraints and the residual constraints $\mathcal{P}'$. The substitutions σ and σ' contain four kinds of mappings: $\varsigma \mapsto \tau$,

$\alpha \mapsto A$, $\delta \mapsto \Delta$ and $\omega \to \gamma$ which instantiate respectively skeleton variables, type variables, dirt variables and coercion variables.

Theorem 8 (Correctness of Solving)**.** *For any set $\mathcal{Q}$, the call* $\texttt{solve}(\bullet; \bullet; \mathcal{Q})$ *either results in a failure, in which case $\mathcal{Q}$ has no solutions, or returns $(\sigma, \mathcal{P})$ such that for any $\sigma' \models \mathcal{Q}$, there exists $\sigma'' \models \mathcal{P}$ where $\sigma' = \sigma'' \cdot \sigma$.*

The solver is invoked with $\texttt{solve}(\bullet; \bullet; \mathcal{Q})$, to process the constraints $\mathcal{Q}$ generated in the first phase of the algorithm, i.e., with an empty substitution and no processed constraints. The `solve` function is defined by case analysis on the queue.

Empty Queue When the queue is empty, all constraints have been processed. What remains are the residual constraints and the solving substitution σ, which are both returned as the result of the solver.

$$\texttt{solve}(\sigma;\, \mathcal{P};\, \bullet) = (\sigma,\, \mathcal{P})$$

Skeleton Equalities The next set of cases we consider are those where the queue is non-empty and its first element is an equality between skeletons $\tau_1 = \tau_2$. We consider seven possible cases based on the structure of τ_1 and τ_2 that together essentially implement conventional unification as used in Hindley-Milner type inference [18].

$$
\begin{aligned}
&\texttt{solve}(\sigma;\, \mathcal{P}; \tau_1 = \tau_2, \mathcal{Q}) = \\
&\quad \texttt{match}\ \tau_1 = \tau_2\ \texttt{with} \\
&\quad \mid \varsigma = \varsigma \mapsto \texttt{solve}(\sigma;\, \mathcal{P};\, \mathcal{Q}) \\
&\quad \mid \varsigma = \tau \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_{\varsigma}(\tau)\ \ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\, \bullet; \sigma'(\mathcal{Q}, \mathcal{P})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \texttt{else fail} \\
&\quad \mid \tau = \varsigma \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_{\varsigma}(\tau)\ \ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\, \bullet; \sigma'(\mathcal{Q}, \mathcal{P})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \texttt{else fail} \\
&\quad \mid \texttt{Unit} = \texttt{Unit} \mapsto \texttt{solve}(\sigma;\, \mathcal{P};\, \mathcal{Q}) \\
&\quad \mid (\tau_1 \to \tau_2) = (\tau_3 \to \tau_4) \mapsto \texttt{solve}(\sigma;\, \mathcal{P};\, \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q}) \\
&\quad \mid (\tau_1 \Rrightarrow \tau_2) = (\tau_3 \Rrightarrow \tau_4) \mapsto \texttt{solve}(\sigma;\, \mathcal{P};\, \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q}) \\
&\quad \mid \texttt{otherwise} \mapsto \texttt{fail}
\end{aligned}
$$

The first case applies when both skeletons are the same type variable ς. Then the equality trivially holds. Hence we drop it and proceed with solving the remaining constraints. The next two cases apply when either τ_1 or τ_2 is a skeleton variable ς. If the occurs check fails[2], there is no finite solution and the algorithm signals failure. Otherwise, the constraint is solved by instantiating the ς. This additional substitution is accumulated and applied to all other constraints $\mathcal{P}, \mathcal{Q}$. Because the substitution might have modified some of the already processed constraints $\mathcal{P}$, we have to revisit them. Hence, they are all pushed back onto the queue, which is processed recursively.

The next three cases consider three different ways in which the two skeletons can have the same instantiated top-level structure. In those cases the equality is decomposed into equalities on the subterms, which are pushed onto the queue and processed recursively.

The last catch-all case deals with all ways in which the two skeletons can be instantiated to different structures. Then there is no solution.

Skeleton Annotations The next four cases consider a skeleton annotation $\alpha : \tau$ at the head of the queue, and propagate the skeleton instantiation to the type

[2]Meaning that $\varsigma \in \mathit{fv}_{\varsigma}(\tau)$, which leads to a cyclic structure.

variable. The first case, where the skeleton is a variable ς does nothing other than moving the annotation to the processed constraints and proceeds with the remainder of the queue. In the other three cases, the skeleton is instantiated and the solver instantiates the type variable with the corresponding structure, introducing fresh variables for any subterms. The instantiating substitution is accumulated and applied to the remaining constraints, which are processed recursively.

$$
\begin{array}{l}
\texttt{solve}(\sigma;\, \mathcal{P};\, \alpha : \tau, \mathcal{Q}) = \\
\quad \texttt{match}\ \tau\ \texttt{with} \\
\quad \mid \varsigma \mapsto \texttt{solve}(\sigma;\, \mathcal{P}, \alpha : \tau;\, \mathcal{Q}) \\
\quad \mid \texttt{Unit} \mapsto \texttt{let}\ \sigma' = [\texttt{Unit}/\alpha]\ \texttt{in}\ \texttt{solve}(\sigma' \cdot \sigma;\, \bullet;\, \sigma'(\mathcal{Q}, \mathcal{P})) \\
\quad \mid \tau_1 \to \tau_2 \mapsto \ \texttt{let}\ \sigma' = [(\alpha_1^{\tau_1} \to \alpha_2^{\tau_2} \,!\, \delta)/\alpha]\ \texttt{in} \\
\qquad\qquad\qquad\qquad \texttt{solve}(\sigma' \cdot \sigma;\, \bullet;\, \alpha_1 : \tau_1, \alpha_2 : \tau_2, \sigma'(\mathcal{Q}, \mathcal{P})) \\
\quad \mid \tau_1 \Rrightarrow \tau_2 \mapsto \ \texttt{let}\ \sigma' = [(\alpha_1^{\tau_1} \,!\, \delta_1 \Rrightarrow \alpha_2^{\tau_2} \,!\, \delta_2)/\alpha]\ \texttt{in} \\
\qquad\qquad\qquad\qquad \texttt{solve}(\sigma' \cdot \sigma;\, \bullet;\, \alpha_1 : \tau_1, \alpha_2 : \tau_2, \sigma'(\mathcal{Q}, \mathcal{P}))
\end{array}
$$

Value Type Subtyping Next are the cases where a subtyping constraint between two value types $A_1 \leqslant A_2$, with the coercion variable ω as evidence, is at the head of the queue. We consider six different situations.

$$
\begin{array}{l}
\texttt{solve}(\sigma;\, \mathcal{P};\, \omega : A_1 \leqslant A_2, \mathcal{Q}) = \\
\quad \texttt{match}\ A_1 \leqslant A_2\ \texttt{with} \\
\quad \mid A \leqslant A \mapsto \texttt{let}\ T = \mathit{elab}_s(A)\ \texttt{in}\ \texttt{solve}([\langle T \rangle / \omega] \cdot \sigma;\, \mathcal{P};\, \mathcal{Q}) \\
\quad \mid \alpha^{\tau_1} \leqslant A \mapsto \texttt{let}\ \tau_2 = \mathit{skeleton}(A)\ \texttt{in}\ \texttt{solve}(\sigma;\, \mathcal{P}, \omega : \alpha^{\tau_1} \leqslant A;\, \tau_1 = \tau_2, \mathcal{Q}) \\
\quad \mid A \leqslant \alpha^{\tau_1} \mapsto \texttt{let}\ \tau_2 = \mathit{skeleton}(A)\ \texttt{in}\ \texttt{solve}(\sigma;\, \mathcal{P}, \omega : A \leqslant \alpha^{\tau_1};\, \tau_2 = \tau_1, \mathcal{Q}) \\
\quad \mid (A_1 \to B_1 \,!\, \Delta_1) \leqslant (A_2 \to B_2 \,!\, \Delta_2) \mapsto \texttt{let}\ \sigma' = [(\omega_1 \to \omega_2 \,!\, \omega_3)/\omega]\ \texttt{in} \\
\qquad \texttt{solve}(\sigma' \cdot \sigma;\, \mathcal{P};\, \omega_1 : A_2 \leqslant A_1, \omega_2 : B_1 \leqslant B_2, \omega_3 : \Delta_1 \leqslant \Delta_2, \mathcal{Q}) \\
\quad \mid (A_1 \,!\, \Delta_1 \Rrightarrow A_2 \,!\, \Delta_2) \leqslant (A_3 \,!\, \Delta_3 \Rrightarrow A_4 \,!\, \Delta_4) \mapsto \texttt{let}\ \sigma' = [(\omega_1 \,!\, \omega_2 \Rrightarrow \omega_3 \,!\, \omega_4)/\omega]\ \texttt{in} \\
\qquad \texttt{solve}(\sigma' \cdot \sigma;\, \mathcal{P};\, \omega_1 : A_3 \leqslant A_1, \omega_2 : \Delta_3 \leqslant \Delta_1, \omega_3 : A_2 \leqslant A_4, \omega_4 : \Delta_2 \leqslant \Delta_4, \mathcal{Q}) \\
\quad \mid \texttt{otherwise} \mapsto \texttt{fail}
\end{array}
$$

If the two types are equal, the subtyping holds trivially through reflexivity. The solver thus drops the constraint and instantiates ω with the reflexivity coercion $\langle T \rangle$. Note that each coercion variable only appears in one constraint. Hence, we only accumulate the substitution and do not have to apply it to the other constraints. In the next two cases, one of the two types is a type variable α. Then we move the constraint to the processed set. We also add an equality constraint between the skeletons to the queue. This enforces the invariant that only types with the same skeleton are compared. With this skeleton equality the type structure (if any) from the type is also transferred to the type variable. The next two cases concern two types with the same top-level instantiation. The solver then decomposes the constraint into constraints on the corresponding subterms and appropriately relates the evidence of the old constraint to the new ones. The final case catches all situations where the two types are instantiated with a different structure and thus there is no solution.

Auxiliary function $\mathit{skeleton}(A)$, defined in Figure 4.20, computes the skeleton of A. A skeleton of a type captures its structure (modulo the dirt information), which is directly expressed in clauses 2, 3, and 4. Hence, in order to capture the whole skeleton of a type, the only missing piece of information is the skeleton of all type variables appearing in the type.

$$\boxed{skeleton(A) = \tau}$$

$$\begin{array}{lcl} skeleton(\alpha^{\tau}) & = & \tau \\ skeleton(\texttt{Unit}) & = & \texttt{Unit} \\ skeleton(A \to B\,!\,\Delta) & = & skeleton(A) \to skeleton(B) \\ skeleton(A\,!\,\Delta_1 \Rrightarrow B\,!\,\Delta_2) & = & skeleton(A) \Rrightarrow skeleton(B) \end{array}$$

Figure 4.20: Skeleton extraction function from value types

As we mentioned earlier, each type variable is implicitly annotated with its skeleton, which allows for the complete determination of the skeleton of a type (clause 1).

Dirt Subtyping The final six cases deal with subtyping constraints between dirts.

$$\begin{array}{l} \texttt{solve}(\sigma;\ \mathcal{P}; \omega : \Delta \leqslant \Delta', \mathcal{Q}) = \\ \quad \texttt{match}\ \Delta \leqslant \Delta'\ \texttt{with} \\ \quad \mid \mathcal{O} \cup \delta \leqslant \mathcal{O}' \cup \delta' \mapsto \texttt{if}\ \mathcal{O} \neq \emptyset\ \ \texttt{then}\ \ \texttt{let}\ \sigma' = [((\mathcal{O} \backslash \mathcal{O}') \cup \delta'')/\delta', \mathcal{O} \cup \omega'/\omega]\ \texttt{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{solve}(\sigma' \cdot \sigma;\ \bullet; (\omega' : \delta \leq \sigma'(\Delta')), \sigma'(\mathcal{Q}, \mathcal{P})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \texttt{solve}(\sigma;\ \mathcal{P}, (\omega : \Delta \leqslant \Delta');\ \mathcal{Q}) \\ \quad \mid \emptyset \leqslant \Delta' \mapsto \texttt{solve}([\emptyset_{\Delta'}/\omega] \cdot \sigma;\ \mathcal{P};\ \mathcal{Q}) \\ \quad \mid \delta \leqslant \emptyset \mapsto \texttt{let}\ \sigma' = [\emptyset/\delta;\ \emptyset_{\emptyset}/\omega]\ \texttt{in}\ \texttt{solve}(\sigma' \cdot \sigma;\ \bullet;\ \sigma'(\mathcal{Q}, \mathcal{P})) \\ \quad \mid \mathcal{O} \cup \delta \leqslant \mathcal{O}' \mapsto \\ \qquad \texttt{if}\ \mathcal{O} \subseteq \mathcal{O}'\ \texttt{then}\ \texttt{let}\ \sigma' = [\mathcal{O} \cup \omega'/\omega]\ \texttt{in}\ \texttt{solve}(\sigma' \cdot \sigma;\ \mathcal{P}, (\omega' : \delta \leqslant \mathcal{O}');\ \mathcal{Q})\ \texttt{else}\ \texttt{fail} \\ \quad \mid \mathcal{O} \leqslant \mathcal{O}' \mapsto \texttt{if}\ \mathcal{O} \subseteq \mathcal{O}'\ \texttt{then}\ \texttt{let}\ \sigma' = [\mathcal{O} \cup \emptyset_{\mathcal{O}' \backslash \mathcal{O}}/\omega]\ \texttt{in}\ \texttt{solve}(\sigma' \cdot \sigma;\ \mathcal{P};\ \mathcal{Q})\ \texttt{else}\ \texttt{fail} \\ \quad \mid \mathcal{O} \leqslant \mathcal{O}' \cup \delta' \mapsto\ \ \texttt{let}\ \sigma' = [(\mathcal{O} \backslash \mathcal{O}') \cup \delta''/\delta';\ \mathcal{O}' \cup \emptyset_{(\mathcal{O}' \backslash \mathcal{O}) \cup \delta''}/\omega]\ \texttt{in} \\ \qquad\qquad\qquad\qquad\quad \texttt{solve}(\sigma' \cdot \sigma;\ \bullet;\ \sigma'(\mathcal{Q}, \mathcal{P})) \end{array}$$

If the two dirts are of the general form $\mathcal{O} \cup \delta$ and $\mathcal{O}' \cup \delta'$, we distinguish two subcases. Firstly, if $\mathcal{O}$ is empty, there is nothing to be done and we move the constraint to the processed set. Secondly, if $\mathcal{O}$ is non-empty, we partially instantiate δ' with any of the operations that appear in $\mathcal{O}$ but not in $\mathcal{O}'$. We then drop $\mathcal{O}$ from

the constraint, and, after substitution, proceed with processing all constraints. For instance, for $\{\mathtt{Op}_1\} \cup \delta \leqslant \{\mathtt{Op}_2\} \cup \delta'$, we instantiate δ' to $\{\mathtt{Op}_1\} \cup \delta''$—where δ'' is a fresh dirt variable—and proceed with the simplified constraint $\delta \leqslant \{\mathtt{Op}_1, \mathtt{Op}_2\} \cup \delta''$. Note that due to the set semantics of dirts, it is not valid to simplify the above constraint to $\delta \leqslant \{\mathtt{Op}_2\} \cup \delta''$. After all the substitution $[\delta \mapsto \{\mathtt{Op}_1\}, \delta'' \mapsto \emptyset]$ solves the former and the original constraint, but not the latter.

The second case, $\emptyset \leqslant \Delta'$, always holds and is discharged by instantiating ω to $\emptyset_{\Delta'}$. The third case, $\delta \leqslant \emptyset$, has only one solution: $\delta \mapsto \emptyset$ with coercion $\emptyset_\emptyset$. The fourth case, $\mathcal{O} \cup \delta \leqslant \mathcal{O}'$, has as many solutions as there are subsets of $\mathcal{O}'$, provided that $\mathcal{O} \subseteq \mathcal{O}'$. We then simplify the constraint to $\delta \leqslant \mathcal{O}'$, which we move to the set of processed constraints. The fifth case, $\mathcal{O} \leqslant \mathcal{O}'$, holds iff $\mathcal{O} \subseteq \mathcal{O}'$. The last case, $\mathcal{O} \leqslant \mathcal{O}' \cup \delta'$, is like the first, but without a dirt variable in the left-hand side. We can satisfy it in a similar fashion, by partially instantiating δ' with $(\mathcal{O} \setminus \mathcal{O}') \cup \delta''$—where δ'' is a fresh dirt variable. Now the constraint is satisfied and can be discarded.

4.5.4 Discussion

At first glance, the constraint generation algorithm of Section 4.5.2 might seem needlessly complex, due to eager constraint solving for let-generalisation. Yet, we want to generalise at local `let`-bound values over both type and skeleton variables. As it will become apparent in Section 4.6, if we only generalise at the top over skeleton variables, the erasure does not yield local polymorphism. This means that we must solve all equations between skeletons before generalising. In turn, since skeleton constraints are generated when solving subtyping constraints (Section 4.5.3), all skeleton annotations should be available during constraint solving. This can not be achieved unless the generated constraints are propagated statefully.

4.6 Erasure of Effect Information from ExEff

4.6.1 The SkelEff Language

The target of the erasure is SkelEff, which is essentially a copy of ExEff from which all effect information Δ, type information T and coercions γ have been removed. Instead, skeletons τ play the role of plain types. Thus, SkelEff is essentially System F extended with term-level (but not type-level) support for algebraic effects.

Terms

$$\text{value } v ::= x \mid () \mid h \mid \texttt{fun}\ (x : \tau) \mapsto c \mid \Lambda\varsigma.v \mid v\ \tau$$

$$\text{handler } h ::= \{\texttt{return}\ (x : \tau) \mapsto c_r,\ \texttt{Op}_1\ x\ k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\ x\ k \mapsto c_{\texttt{Op}_n}\}$$

$$\text{computation } c ::= v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c \mid \texttt{return}\ v \mid \texttt{Op}\ v\ (y : \tau.c) \mid \texttt{do}\ x \leftarrow c_1; c_2 \mid \texttt{handle}\ c\ \texttt{with}\ v$$

Types

$$\text{type } \tau ::= \varsigma \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \mid \texttt{Unit} \mid \forall\varsigma.\tau$$

Figure 4.21: SkelEff Syntax

Figure 4.21 defines the syntax of SkelEff. One thing to notice is that the skeleton variables become type variables for this language. Moreover, in the values, we have skeleton variables applications $(v\ \tau)$ and abstractions $(\Lambda\varsigma.v)$.

The main point of SkelEff is to show that we can erase the effects and subtyping from ExEff to obtain types that are compatible with a System F-like language. At the term-level, SkelEff also resembles a subset of Multicore OCaml [27], which provides native support for algebraic effects and handlers but features no explicit polymorphism. Moreover, SkelEff can also serve as a staging area for further elaboration into System F-like languages without support for algebraic effects and handlers (e.g., Haskell or regular OCaml). In those cases, computation terms can be compiled to one of the known encodings in the literature, such as a free monad representation [41, 74], with delimited control [47], or using continuation-passing style [51], while values can typically be carried over as they are.

4.6.2 Typing

Typing for SkelEff values and computations take the form $\Gamma \vdash_{\text{ev}} v : \tau$ and $\Gamma \vdash_{\text{ec}} c : \tau$. They are defined in Figure 4.22. The type-system follows that of ExEff As illustrated by the rules, SkelEff is essentially System F extended with term-level (but not type-level) support for algebraic effects.

4.6.3 Erasure

Figure 4.23 defines erasure functions $\epsilon^{\sigma}_{\text{v}}(v)$, $\epsilon^{\sigma}_{\text{c}}(c)$, $\epsilon^{\sigma}_{\text{V}}(T)$, $\epsilon^{\sigma}_{\text{C}}(\underline{C})$ and $\epsilon^{\sigma}_{\text{E}}(\Gamma)$ for values, computations, value types, computation types, and type environments

typing environment $\Gamma ::= \epsilon \mid \Gamma, \varsigma \mid \Gamma, x : \tau$

Values

$$\boxed{\Gamma \vdash_{\text{ev}} v : \tau}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{\text{ev}} x : \tau}\ \textsc{SkVar} \qquad \frac{}{\Gamma \vdash_{\text{ev}} () : \texttt{Unit}}\ \textsc{SkUnit}$$

$$\frac{\Gamma, x : \tau_1 \vdash_{\text{ec}} c : \tau_2 \qquad \Gamma \vdash_{\tau} \tau_1}{\Gamma \vdash_{\text{ev}} (\texttt{fun}\ x : \tau_1 \mapsto c) : \tau_1 \to \tau_2}\ \textsc{SkAbs} \qquad \frac{\Gamma \vdash_{\text{ev}} v : \forall \varsigma.\tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\text{ev}} v\ \tau_2 : \tau_1[\tau_2/\varsigma]}\ \textsc{SkTyApp}$$

$$\frac{\Gamma, \varsigma \vdash_{\text{ev}} v : \tau}{\Gamma \vdash_{\text{ev}} \Lambda\varsigma.v : \forall \varsigma.\tau}\ \textsc{SkTyAbs}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau_x \vdash_{\text{ec}} c_r : \tau \\ \left[(\texttt{Op} : \tau_1 \to \tau_2) \in \Sigma \qquad \Gamma, x : \tau_1, k : \tau_2 \to \tau \vdash_{\text{ec}} c_{\texttt{Op}} : \tau\right]_{\texttt{Op} \in \mathcal{O}}\end{array}}{\Gamma \vdash_{\text{ev}} \{\texttt{return}\ (x : \tau_x) \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} : \tau_x \Rrightarrow \tau}\ \textsc{SkHandler}$$

Computations

$$\boxed{\Gamma \vdash_{\text{ec}} c : \tau}$$

$$\frac{\Gamma \vdash_{\text{ev}} v_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{ev}} v_2 : \tau_1}{\Gamma \vdash_{\text{ec}} v_1\ v_2 : \tau_2}\ \textsc{SkApp}$$

$$\frac{\Gamma \vdash_{\text{ev}} v : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\text{ec}} c : \tau_2}{\Gamma \vdash_{\text{ec}} \texttt{let}\ x = v\ \texttt{in}\ c : \tau_2}\ \textsc{SkLet} \qquad \frac{\Gamma \vdash_{\text{ev}} v : \tau}{\Gamma \vdash_{\text{ec}} \texttt{return}\ v : \tau}\ \textsc{SkReturn}$$

$$\frac{(\texttt{Op} : \tau_1 \to \tau_2) \in \Sigma \qquad \Gamma \vdash_{\text{ev}} v : \tau_1 \qquad \Gamma, y : \tau_2 \vdash_{\text{ec}} c : \tau}{\Gamma \vdash_{\text{ec}} \texttt{Op}\ v\ (y : \tau_2.c) : \tau}\ \textsc{SkOp}$$

$$\frac{\Gamma \vdash_{\text{ec}} c_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\text{ec}} c_2 : \tau_2}{\Gamma \vdash_{\text{ec}} \texttt{do}\ x \leftarrow c_1; c_2 : \tau_2}\ \textsc{SkDo}$$

$$\frac{\Gamma \vdash_{\text{ev}} v : \tau_1 \Rrightarrow \tau_2 \qquad \Gamma \vdash_{\text{ec}} c : \tau_1}{\Gamma \vdash_{\text{ec}} \texttt{handle}\ c\ \texttt{with}\ v : \tau_2}\ \textsc{SkHandle}$$

Figure 4.22: SkelEff Typing

$$
\begin{aligned}
\epsilon^\sigma_{\mathrm{v}}(x) &= x & \epsilon^\sigma_{\mathrm{v}}(\Lambda\delta.v) &= \epsilon^\sigma_{\mathrm{v}}(v)\\
\epsilon^\sigma_{\mathrm{v}}(()) &= () & \epsilon^\sigma_{\mathrm{v}}(\Lambda(\omega:\pi).v) &= \epsilon^\sigma_{\mathrm{v}}(v)\\
\epsilon^\sigma_{\mathrm{v}}(v \rhd \gamma) &= \epsilon^\sigma_{\mathrm{v}}(v) & \epsilon^\sigma_{\mathrm{v}}(v\ \tau) &= \epsilon^\sigma_{\mathrm{v}}(v)\ \tau\\
\epsilon^\sigma_{\mathrm{v}}(\mathtt{fun}\ (x:T) \mapsto c) &= \mathtt{fun}\ (x:\epsilon^\sigma_{\mathrm{V}}(T)) \mapsto \epsilon^\sigma_{\mathrm{c}}(c) & \epsilon^\sigma_{\mathrm{v}}(v\ T) &= \epsilon^\sigma_{\mathrm{v}}(v)\\
\epsilon^\sigma_{\mathrm{v}}(\Lambda\varsigma.v) &= \Lambda\varsigma.\epsilon^\sigma_{\mathrm{v}}(v) & \epsilon^\sigma_{\mathrm{v}}(v\ \Delta) &= \epsilon^\sigma_{\mathrm{v}}(v)\\
\epsilon^\sigma_{\mathrm{v}}(\Lambda(\alpha:\tau).v) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{v}}(v) & \epsilon^\sigma_{\mathrm{v}}(v\ \gamma) &= \epsilon^\sigma_{\mathrm{v}}(v)
\end{aligned}
$$

$$
\begin{aligned}
&\epsilon^\sigma_{\mathrm{v}}(\{\mathtt{return}\ (x:T) \mapsto c_r, [\mathtt{Op}\,x\,k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\}) =\\
&\quad \{\mathtt{return}\ (x:\epsilon^\sigma_{\mathrm{V}}(T)) \mapsto \epsilon^\sigma_{\mathrm{c}}(c_r), [\mathtt{Op}\,x\,k \mapsto \epsilon^\sigma_{\mathrm{c}}(c_{\mathtt{Op}})]_{\mathtt{Op}\in\mathcal{O}}\}
\end{aligned}
$$

$$
\begin{aligned}
\epsilon^\sigma_{\mathrm{c}}(v_1\ v_2) &= \epsilon^\sigma_{\mathrm{v}}(v_1)\ \epsilon^\sigma_{\mathrm{v}}(v_2)\\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{let}\ x = v\ \mathtt{in}\ c) &= \mathtt{let}\ x = \epsilon^\sigma_{\mathrm{v}}(v)\ \mathtt{in}\ \epsilon^\sigma_{\mathrm{c}}(c)\\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{return}\ v) &= \mathtt{return}\ (\epsilon^\sigma_{\mathrm{v}}(v))\\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{Op}\ v\ (y:T.c)) &= \mathtt{Op}\ (\epsilon^\sigma_{\mathrm{v}}(v))\ (y:\epsilon^\sigma_{\mathrm{V}}(T).\epsilon^\sigma_{\mathrm{c}}(c))\\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{do}\ x \leftarrow c_1; c_2) &= \mathtt{do}\ x \leftarrow \epsilon^\sigma_{\mathrm{c}}(c_1); \epsilon^\sigma_{\mathrm{c}}(c_2)\\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{handle}\ c\ \mathtt{with}\ v) &= \mathtt{handle}\ \epsilon^\sigma_{\mathrm{c}}(c)\ \mathtt{with}\ \epsilon^\sigma_{\mathrm{v}}(v)\\
\epsilon^\sigma_{\mathrm{c}}(c \rhd \gamma) &= \epsilon^\sigma_{\mathrm{c}}(c)
\end{aligned}
$$

$$
\begin{aligned}
\epsilon^\sigma_{\mathrm{V}}(\alpha) &= \sigma(\alpha) & \epsilon^\sigma_{\mathrm{C}}(T\ !\ \Delta) &= \epsilon^\sigma_{\mathrm{V}}(T)\\
\epsilon^\sigma_{\mathrm{V}}(T \to \underline{C}) &= \epsilon^\sigma_{\mathrm{V}}(T) \to \epsilon^\sigma_{\mathrm{C}}(\underline{C})\\
\epsilon^\sigma_{\mathrm{V}}(\underline{C}_1 \Rrightarrow \underline{C}_2) &= \epsilon^\sigma_{\mathrm{C}}(\underline{C}_1) \Rrightarrow \epsilon^\sigma_{\mathrm{C}}(\underline{C}_2) & \epsilon^\sigma_{\mathrm{E}}(\epsilon) &= \epsilon\\
\epsilon^\sigma_{\mathrm{V}}(\mathtt{Unit}) &= \mathtt{Unit} & \epsilon^\sigma_{\mathrm{E}}(\Gamma, \varsigma) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma), \varsigma\\
\epsilon^\sigma_{\mathrm{V}}(\pi \Rightarrow T) &= \epsilon^\sigma_{\mathrm{V}}(T) & \epsilon^\sigma_{\mathrm{E}}(\Gamma, \alpha:\tau) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{E}}(\Gamma)\\
\epsilon^\sigma_{\mathrm{V}}(\forall\varsigma.T) &= \forall\varsigma.\epsilon^\sigma_{\mathrm{V}}(T) & \epsilon^\sigma_{\mathrm{E}}(\Gamma, \delta) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma)\\
\epsilon^\sigma_{\mathrm{V}}(\forall(\alpha:\tau).T) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{V}}(T) & \epsilon^\sigma_{\mathrm{E}}(\Gamma, x:T) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma), x:\epsilon^\sigma_{\mathrm{V}}(T)\\
\epsilon^\sigma_{\mathrm{V}}(\forall\delta.T) &= \epsilon^\sigma_{\mathrm{V}}(T) & \epsilon^\sigma_{\mathrm{E}}(\Gamma, \omega:\pi) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma)
\end{aligned}
$$

Figure 4.23: Definition of type erasure.

respectively. All five functions take a substitution σ from the free type variables α to their skeleton τ as an additional parameter.

Thanks to the skeleton-based design of ExEff, erasure is straightforward. All types are erased to their skeletons, dropping quantifiers for type variables and all occurrences of dirt sets. Moreover, coercions are dropped from values and computations. Finally, all binders and elimination forms for type variables, dirt set variables and coercions are dropped from values and type environments.

The expected following theorems hold. Types are preserved by erasure.

Theorem 9 (Type Preservation). *If* $\Gamma \vdash_{\mathrm{v}} v : T$ *then* $\epsilon^{\emptyset}_{\mathrm{E}}(\Gamma) \vdash_{\mathrm{ev}} \epsilon^{\Gamma}_{\mathrm{v}}(v) : \epsilon^{\Gamma}_{\mathrm{V}}(T)$. *If* $\Gamma \vdash_{\mathrm{c}} c : \underline{C}$ *then* $\epsilon^{\emptyset}_{\mathrm{E}}(\Gamma) \vdash_{\mathrm{ec}} \epsilon^{\Gamma}_{\mathrm{c}}(c) : \epsilon^{\Gamma}_{\mathrm{C}}(\underline{C})$.

Here we abuse the notation and use Γ as a substitution from type variables to skeletons used by the erasure functions.

4.6.4 Operational Semantics for SkelEff

Figure 4.24 presents the small-step, call-by-value operational semantics of SkelEff, and Figure 4.25 gives the congruence closure of the step relations

Finally, we have that erasure preserves the operational semantics.

Theorem 10 (Semantic Preservation). *If* $v \rightsquigarrow_{\mathrm{v}} v'$ *then* $\epsilon^{\sigma}_{\mathrm{v}}(v) \equiv^{\rightsquigarrow}_{\mathrm{v}} \epsilon^{\sigma}_{\mathrm{v}}(v')$. *If* $c \rightsquigarrow_{\mathrm{c}} c'$ *then* $\epsilon^{\sigma}_{\mathrm{c}}(c) \equiv^{\rightsquigarrow}_{\mathrm{c}} \epsilon^{\sigma}_{\mathrm{c}}(c')$.

In both cases, $\equiv^{\rightsquigarrow}$ denotes the congruence closure of the step relation in SkelEff.

4.6.5 Discussion

Binder dropping Typically, when type information is erased from call-by-value languages, type binders are erased by replacing them with other (dummy) binders. For instance, the expected definition of erasure would be:

$$\epsilon^{\sigma}_{\mathrm{v}}(\Lambda(\alpha : \tau).v) = \lambda(x : \mathtt{Unit}).\epsilon^{\sigma}_{\mathrm{v}}(v)$$

This replacement is motivated by a desire to preserve the behaviour of the typed terms. By dropping binders, values might be turned into computations that trigger their side-effects immediately, rather than at the later point where the original binder was eliminated. However, there is no call for this circumspect approach in our setting, as our grammatical partition of terms in values (without side-effects) and computations (with side-effects) guarantees that this problem cannot happen when we erase values to values and computations to computations.

Sensible erasure In Section 4.5.4, we argued that we generalise at local let-bound values over both type and skeleton variables. We have also shown it by means of an example in Section 4.2.3. This is now more apparent as we have shown the erasure procedure and the inference algorithm that we substitute every free type-variable α with its skeleton τ so that the skeletons become the actual type of the SkelEff calculus. Therefore, skeleton generalisation is crucial for correct erasure. The following example explains the importance of skeleton generalisation.

$$\text{value result } v^R ::= () \mid h \mid \texttt{fun}\ (x : \tau) \mapsto c \mid \Lambda\varsigma.v$$
$$\text{computation result } c^R ::= \texttt{return}\ v^R \mid \texttt{Op}\ v^R\ (y.c)$$

Values

$$\boxed{v \rightsquigarrow_{\mathrm{v}} v'}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v\ \tau \rightsquigarrow_{\mathrm{v}} v'\ \tau} \qquad (\Lambda\varsigma.v)\ \tau \rightsquigarrow_{\mathrm{v}} v[\tau/\varsigma]$$

Computations

$$\boxed{c \rightsquigarrow_{\mathrm{c}} c'}$$

$$\frac{v_1 \rightsquigarrow_{\mathrm{v}} v_1'}{v_1\ v_2 \rightsquigarrow_{\mathrm{c}} v_1'\ v_2} \qquad \frac{v_2 \rightsquigarrow_{\mathrm{v}} v_2'}{v_1^R\ v_2 \rightsquigarrow_{\mathrm{c}} v_1^R\ v_2'} \qquad (\texttt{fun}\ (x : \tau) \mapsto c)\ v^R \rightsquigarrow_{\mathrm{c}} c[v^R/x]$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{let}\ x = v\ \texttt{in}\ c \rightsquigarrow_{\mathrm{c}} \texttt{let}\ x = v'\ \texttt{in}\ c} \qquad \texttt{let}\ x = v^R\ \texttt{in}\ c \rightsquigarrow_{\mathrm{c}} c[v^R/x]$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{return}\ v \rightsquigarrow_{\mathrm{c}} \texttt{return}\ v'} \qquad \frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{Op}\ v\ (y : \tau.c) \rightsquigarrow_{\mathrm{c}} \texttt{Op}\ v'\ (y : \tau.c)}$$

$$\frac{c_1 \rightsquigarrow_{\mathrm{c}} c_1'}{\texttt{do}\ x \leftarrow c_1; c_2 \rightsquigarrow_{\mathrm{c}} \texttt{do}\ x \leftarrow c_1'; c_2} \qquad \texttt{do}\ x \leftarrow \texttt{return}\ v^R; c_2 \rightsquigarrow_{\mathrm{c}} c_2[v^R/x]$$

$$\texttt{do}\ x \leftarrow \texttt{Op}\ v^R\ (y : \tau.c_1); c_2 \rightsquigarrow_{\mathrm{c}} \texttt{Op}\ v^R\ (y : \tau.\texttt{do}\ x \leftarrow c_1; c_2)$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{handle}\ c\ \texttt{with}\ v \rightsquigarrow_{\mathrm{c}} \texttt{handle}\ c\ \texttt{with}\ v'}$$

$$\frac{c \rightsquigarrow_{\mathrm{c}} c'}{\texttt{handle}\ c\ \texttt{with}\ v^R \rightsquigarrow_{\mathrm{c}} \texttt{handle}\ c'\ \texttt{with}\ v^R}$$

$$\texttt{handle}\ (\texttt{return}\ v^R)\ \texttt{with}\ h \rightsquigarrow_{\mathrm{c}} c_r[v^R/x]$$

$$\texttt{handle}\ (\texttt{Op}\ v^R\ (y : \tau.c))\ \texttt{with}\ h \rightsquigarrow_{\mathrm{c}} {}^{c_{\texttt{Op}}}[v^R/x, (\texttt{fun}\ (y : \tau) \mapsto \texttt{handle}\ c\ \texttt{with}\ h)/k]$$

$$\texttt{handle}\ (\texttt{Op}\ v^R\ (y : \tau.c))\ \texttt{with}\ h \rightsquigarrow_{\mathrm{c}} \texttt{Op}\ v^R\ (y : \tau.\texttt{handle}\ c\ \texttt{with}\ h)$$

Figure 4.24: SkelEff Operational Semantics

Terms with holes

$$
\begin{aligned}
\text{value-holed value } V^{\mathrm{v}} ::= {} & [\,] \mid x \mid () \mid H^{\mathrm{v}} \mid \mathtt{fun}\ (x : \tau) \mapsto C^{\mathrm{v}} \\
& \mid \Lambda\varsigma.V^{\mathrm{v}} \mid V^{\mathrm{v}}\ \tau \\
\text{value-holed handler } H^{\mathrm{v}} ::= {} & \{\mathtt{return}\ (x : \tau) \mapsto C^{\mathrm{v}}_r, \\
& \mid [\mathtt{Op}\, x\, k \mapsto C^{\mathrm{v}}_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} \\
\text{value-holed computation } C^{\mathrm{v}} ::= {} & V^{\mathrm{v}}_1\ V^{\mathrm{v}}_2 \mid \mathtt{let}\ x = V^{\mathrm{v}}\ \mathtt{in}\ C^{\mathrm{v}} \mid \mathtt{return}\ V^{\mathrm{v}} \\
& \mid \mathtt{Op}\ V^{\mathrm{v}}\ (y : \tau.C^{\mathrm{v}}) \mid \mathtt{do}\ x \leftarrow C^{\mathrm{v}}_1; C^{\mathrm{v}}_2 \\
& \mid \mathtt{handle}\ C^{\mathrm{v}}\ \mathtt{with}\ V^{\mathrm{v}} \\
\text{computation-holed value } V^{\mathrm{c}} ::= {} & x \mid () \mid H^{\mathrm{c}} \mid \mathtt{fun}\ (x : \tau) \mapsto C^{\mathrm{c}} \\
& \mid \Lambda\varsigma.V^{\mathrm{c}} \mid V^{\mathrm{c}}\ \tau \\
\text{computation-holed handler } H^{\mathrm{c}} ::= {} & \{\mathtt{return}\ (x : \tau) \mapsto C^{\mathrm{c}}_r, \\
& [\mathtt{Op}\, x\, k \mapsto C^{\mathrm{c}}_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} \\
\text{computation-holed computation } C^{\mathrm{c}} ::= {} & [\,] \mid V^{\mathrm{c}}_1\ V^{\mathrm{c}}_2 \mid \mathtt{let}\ x = V^{\mathrm{c}}\ \mathtt{in}\ C^{\mathrm{c}} \\
& \mid \mathtt{return}\ V^{\mathrm{c}} \\
& \mid \mathtt{Op}\ V^{\mathrm{c}}\ (y : \tau.C^{\mathrm{c}}) \\
& \mid \mathtt{do}\ x \leftarrow C^{\mathrm{c}}_1; C^{\mathrm{c}}_2 \\
& \mid \mathtt{handle}\ C^{\mathrm{c}}\ \mathtt{with}\ V^{\mathrm{c}}
\end{aligned}
$$

We define values $V^{\mathrm{v}}[v], V^{\mathrm{c}}[c]$, and computations $C^{\mathrm{v}}[v], C^{\mathrm{c}}[c]$ in a straightforward way.

Values

$$\boxed{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v'}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v'} \qquad v \equiv^{\rightsquigarrow}_{\mathrm{v}} v \qquad \frac{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v'}{v' \equiv^{\rightsquigarrow}_{\mathrm{v}} v} \qquad \frac{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v' \qquad v' \equiv^{\rightsquigarrow}_{\mathrm{v}} v''}{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v''}$$

$$\frac{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v'}{V^{\mathrm{v}}[v] \equiv^{\rightsquigarrow}_{\mathrm{v}} V^{\mathrm{v}}[v']} \qquad \frac{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c'}{V^{\mathrm{c}}[c] \equiv^{\rightsquigarrow}_{\mathrm{v}} V^{\mathrm{c}}[c']}$$

Computations

$$\boxed{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c'}$$

$$\frac{c \rightsquigarrow_{\mathrm{c}} c'}{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c'} \qquad c \equiv^{\rightsquigarrow}_{\mathrm{c}} c \qquad \frac{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c'}{c' \equiv^{\rightsquigarrow}_{\mathrm{c}} c} \qquad \frac{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c' \qquad c' \equiv^{\rightsquigarrow}_{\mathrm{c}} c''}{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c''}$$

$$\frac{v \equiv^{\rightsquigarrow}_{\mathrm{v}} v'}{C^{\mathrm{v}}[v] \equiv^{\rightsquigarrow}_{\mathrm{c}} C^{\mathrm{v}}[v']} \qquad \frac{c \equiv^{\rightsquigarrow}_{\mathrm{c}} c'}{C^{\mathrm{c}}[c] \equiv^{\rightsquigarrow}_{\mathrm{c}} C^{\mathrm{c}}[c']}$$

Figure 4.25: Congruence Closures of the Step Relations

The following let-computation we have used in Section 4.2.3 is written using ImpEff syntax:

$$\texttt{let}\ f = \texttt{fun}\ g \mapsto (g\ ())\ \texttt{in}\ c$$

The elaborated ExEff type of the variable f is

$$f : \forall \varsigma_1.\forall \varsigma_2.\forall \alpha_1 : \varsigma_1.\forall \alpha_2 : \varsigma_2.\forall \delta_1.\forall \delta_2.\alpha_1 \leqslant \alpha_2 \Rightarrow \delta_1 \leqslant \delta_2 \Rightarrow (\texttt{Unit} \to \alpha_1\ !\ \delta_1) \to \alpha_2\ !\ \delta_2$$

Note that f is a higher order function that takes a function g of type $(\texttt{Unit} \to \alpha_1\ !\ \delta_1)$ and returns the resulting types $\alpha_2\ !\ \delta_2$. Since f just returns the result of the application, then we know that $\alpha_1 ! \delta_1 = \alpha_2 ! \delta_2$. The elaboration of f into ExEff is as follows:

$$\begin{gathered}\Lambda\varsigma_1.\Lambda\varsigma_2.\Lambda\alpha_1 : \varsigma_1.\Lambda\alpha_2 : \varsigma_2.\Lambda\delta_1.\Lambda\delta_2.\Lambda(\omega_1 : \alpha_1 \leqslant \alpha_2).\Lambda(\omega_2 : \delta_1 \leqslant \delta_2).\\ \texttt{let}\ f = \texttt{fun}\ (g : (\texttt{Unit} \to \alpha_1\ !\ \delta_1) \mapsto (g\ ()) \rhd \omega_1\ !\ \omega_2\end{gathered}$$

In the `unify` algorithm for type variables resolving. We have the following clauses:

$$\mid \alpha^{\tau_1} \leqslant A \mapsto \texttt{let}\ \tau_2 = \mathit{skeleton}(A)\ \texttt{in}\ \texttt{solve}(\sigma;\ \mathcal{P}, \omega : \alpha^{\tau_1} \leqslant A;\ \tau_2 = \tau_1, \mathcal{Q})$$

$$\mid A \leqslant \alpha^{\tau_1} \mapsto \texttt{let}\ \tau_2 = \mathit{skeleton}(A)\ \texttt{in}\ \texttt{solve}(\sigma;\ \mathcal{P}, \omega : A \leqslant \alpha^{\tau_1};\ \tau_2 = \tau_1, \mathcal{Q})$$

Knowing that $\Lambda(\omega_1 : \alpha_1 \leqslant \alpha_2)$ and that the annotated skeletons for α_1 and α_2 are τ_1 and τ_2, respectively, then the unification algorithm unifies τ_1 and τ_2 and we can rewrite the type of f as follows:

$$\forall \varsigma.\forall \alpha_1 : \varsigma.\forall \alpha_2 : \varsigma.\forall \delta_1.\forall \delta_2.\alpha_1 \leqslant \alpha_2 \Rightarrow \delta_1 \leqslant \delta_2 \Rightarrow (\texttt{Unit} \to \alpha_1\ !\ \delta_1) \to \alpha_2\ !\ \delta_2$$

Applying the erasure function $\epsilon^{\sigma}_{(*)}$ on the type and f drops the α and δ quantifiers, the coercion abstractions, the dirt variables and the cast. Moreover, σ contains the substitutions $\alpha_1 \to \varsigma$ and $\alpha_2 \to \varsigma$. Meaning that this substitution will be applied in both the type and the f as follows:

$$f : \forall\varsigma.(\texttt{Unit} \to \varsigma) \to \varsigma$$

$$\texttt{let } f = \Lambda\varsigma.\texttt{fun } (g : (\texttt{Unit} \to \varsigma)) \mapsto (g\ ())$$

The skeleton now got lifted to be the new type-variables in SkelEff. The final type of f gives the expected result. However, without the skeleton generalisation, we do not reach this resulting type. If we use only type variables generalisation, meaning that we remove skeletons entirely and keep only type variables of ExEff, the resulting type of f becomes:

$$\forall\alpha_1.\forall\alpha_2.(\texttt{Unit} \to \alpha_1) \to \alpha_2$$

This type is semantically incorrect since α_1 and α_2 can be different. However, the correct semantics of the program states that they are equivalent.

4.7 Conclusion and Discussion

In this chapter, we presented an explicitly-typed polymorphic core calculus for algebraic effect handlers with a subtyping-based type-and-effect system. We use subtyping in explicit casts with coercions that witness the subtyping proof. Our typing-directed elaboration comes with a constraint-based inference algorithm that turns an implicitly-typed Eff-like language (ImpEff) into an explicitly-typed calculus (ExEff). Moreover, we showed that all coercions and effect information can be erased in a straightforward way (SkelEff), demonstrating that coercions have no computational content. Also, the erasure language can target famous target languages like OCaml.

The calculus introduced in this chapter with explicit typing coercions has the power to eliminate the issues we faced in optimising Eff. As we have discussed in Section 3.8, the current Eff implementation with implicit-subtyping makes code transformations very fragile. Therefore, using ExEff as a core language for the optimisations should eliminate this problem.

In the next section, we discuss the similarities and differences of the calculi we introduced in this chapter with other Eff calculi.

4.7.1 Eff related type systems

The most closely related work is that of Pretnar [71] on inferring algebraic effects for Eff, which is the basis for our implicitly-typed ImpEff calculus, its type system and the type inference algorithm. There are three major differences with Pretnar's inference algorithm.

First, our ImpEff calculus and the inference algorithm are based on the work of Pretnar [71]. However, our work introduces an explicitly-typed calculus. For this reason we have extended the constraint generation phase with the elaboration into ExEff and the constraint solving phase with the construction of coercions.

Secondly, we add skeletons to guarantee erasure. Skeletons also allow us to use standard occurs-check during unification. In contrast, unification in Pretnar's algorithm is inspired by Simonet [86] and performs the occurs-check up to the equivalence closure of the subtyping relation. In order to maintain invariants, all variables in an equivalence class (also called a skeleton) must be instantiated simultaneously, whereas we can process one constraint at a time. As these classes turn out to be surrogates for the underlying skeleton types, we have decided to keep the name.

Finally, Pretnar incorporates garbage collection of constraints [70]. The aim of this approach is to obtain unique and simple type schemes by eliminating redundant constraints. Garbage collection is not suitable for our use as type variables and coercions witnessing subtyping constraints cannot simply be dropped, but must be instantiated in a suitable manner, which cannot be done in general.

Consider, for instance, a situation with type variables α_1, α_2, α_3, α_4, and α_5 where $\alpha_1 \leqslant \alpha_3$, $\alpha_2 \leqslant \alpha_3$, $\alpha_3 \leqslant \alpha_4$, and $\alpha_3 \leqslant \alpha_5$. Suppose that α_3 does not appear in the type. Then garbage collection would eliminate it and replace the constraints by $\alpha_1 \leqslant \alpha_4$, $\alpha_2 \leqslant \alpha_4$, $\alpha_1 \leqslant \alpha_5$, and $\alpha_2 \leqslant \alpha_5$. While garbage collection guarantees that for any ground instantiation of the remaining type variables there exists a valid ground instantiation for α_3, ExEff would need to be extended with joins (or meets) to express a generically valid instantiation like $\alpha_1 \sqcup \alpha_2$. Moreover, we would need additional coercion formers to establish $\alpha_1 \leqslant (\alpha_1 \sqcup \alpha_2)$ or $(\alpha_1 \sqcup \alpha_2) \leqslant \alpha_4$.

As these additional constructs considerably complicate the calculus, we propose a simpler solution. We use ExEff as it is for internal purposes, but display types to programmers in their garbage-collected form.

Chapter Notes and Contributions This chapter is based on the paper *"Explicit Effect Subtyping"* by Saleh et al. [77]. My own contribution to this work was the following:

- Language design including syntax, typing, elaboration for ImpEff, ExEff and SkelEff.
- Constraint generation when elaborating from ImpEff to ExEff.
- Constraint solving algorithm design and implementation.
- Implementation of the language in Eff's compiler.

Chapter 5

Effect Handlers in Logic Programming

So far in this thesis, we have focused on algebraic effect handlers in the setting of Functional Programming. The main advantage of handlers is that they can change the program's control flow by capturing the continuation in order to resume it at a later point. In this chapter, we study effect handlers for the Logic Programming paradigm. Potentially, effect handlers can be used to manipulate Prolog's search strategy and add a layer of modularity to Prolog. While traditionally the language does not have many control flow constructs, the introduction of *delimited control* constructs for Prolog by Schrijvers et al. [80] has remedied this situation. Unfortunately, there are two prominent downsides to delimited control. Firstly, it is a rather primitive feature that has been likened to the imperative `goto`, which was labelled harmful for high-level programming by Dijkstra [22]. Secondly, the overhead of delimited control for encoding state-passing features is non-negligible. For example, the delimited control implementation of DCGs (Definite Clause Grammars) is ten times slower for a tight loop than the traditional Prolog implementation.

This chapter addresses the downsides of delimited control by bringing a form of algebraic effects and handlers to Prolog which provide a high-level structured interface to delimited control. Moreover, in exchange for the restricted expressiveness, we provide two benefits. Firstly, multiple handlers can be combined effortlessly to deal with distinct effects, to deal with one effect in terms of another or to customise the behaviour of an effect. Secondly, we provide an automated program transformation that eliminates much of the overhead of delimited control. We adopt different optimisation techniques from the optimisations we discussed

earlier in Chapter 3. The transformation is formulated using partial evaluation augmented with rewrite rules. These rewrite rules are driven by an effect analysis system that characterises which effects a Prolog goal may trigger.

Our design of effect handlers for Prolog is more restricted than that of the Eff calculus. However, compared to the free form of delimited control, the structured approach of effect handlers simplifies the identification of program patterns that can be optimised.

Chapter layout The rest of this chapter is structured as follows: In section 5.1, we give an introduction to delimited control constructs in Prolog. Then, we introduce effect handlers syntax and give an informal explanation of their semantics. Section 5.2 presents our optimisation techniques that aim to eliminate delimited control code from the input program as delimited control is very costly with respect to run-time. Afterwards, in Section 5.3, we discuss the results of evaluating the run-times of Prolog programs with effect handlers with optimisations and without. Finally, we discuss the work related to effect handlers in Logic Programming in Section 5.4.[1]

5.1 Delimited Control and Algebraic Effect Handlers

This section gives the necessary background on delimited control in Logic Programming and then introduces our algebraic effect handlers syntax and semantics for Prolog.

5.1.1 Delimited Control in Prolog

Delimited control in Prolog is a compelling means to dynamically manipulate the control-flow of programs that was first explored in the setting of functional programming [29, 19]. Schrijvers et al. [80] show its usefulness in Prolog to concisely define implicit state, DCGs and coroutines. More recently, Desouter et al. [21] have shown that delimited control also concisely captures the control-flow manipulation of tabling.

The work of Schrijvers et al. [80] introduces delimited control for Prolog. It provides Prolog with two predicates for delimited control:

[1] All examples of this chapter are available at http://github.com/ah-saleh/prologhandlers

- `reset(G,Cont,T)` executes goal `G`.
- `shift(T1)` suspends the execution of the current goal and captures the remainder up to the nearest surrounding `reset/3`. This remainder is called the *continuation*. It unifies the captured continuation with `Cont` and `T` with `T1`. The control is then returned to the call just after the `reset/3`.

The following example shows delimited control in action.

```
main :- reset(p,Cont,Term),          p :- write(a),
        write(b).                         write(c).
```

```
?- main.
a c b
```

Because `p` terminates without shifting, the variables `Cont` and `Term` are unified with `0`. The next example illustrates the interaction between `shift/1` and `reset/3`.

```
main :- reset(p,Cont,Term),      p :- write(a),
        write(Term),                  shift(hello),
        write(b),                     write(c).
        call(Cont).
```

```
?- main.
a hello b c
```

Executing `?-main.` calls `p` inside the reset, prints `a`, then suspends the execution due to `shift(hello)`, giving the control back to the `main` clause after the `reset/3` and unifying `Term` with `hello` and `Cont` with `(write(c))`.

The meta-interpreter in Figure 5.1 captures the semantics of delimited control. The main predicate that evaluates a goal is `eval/1`. It calls the helper predicate `eval/2` that attaches an extra argument `Signal` to every goal. `Signal` is unified with `ok` when the goal succeeds normally without shifting. When a goal's evaluation is terminated by a `shift(Term)`, `Signal` is unified with `shift(Term,Cont)` such that `Cont` is the remainder of the goal. The `reset(G,Cont,Term)` clause evaluates `G` and unifies `Cont` and `Term` with `0` when `G` terminates normally without shifting; otherwise, they get unified with the returned values of evaluating the goal inside the `reset/3`. The conjunction clause `(G1,G2)` starts by evaluating `G1`. If it succeeds normally, the conjunction continues with evaluating `G2`. If `G1` is stopped by `shift/1`, then the whole conjunction stops and G2 is added to the the main `Signal`. The *ITE*

(If-Then-Else) clause evaluates the condition which has to succeed or fail without shifting. Otherwise, an undefined action error stops the evaluation. Depending on the success or failure of `C`, `G1` or `G2` get executed. If the goal is a built-in Prolog predicate, then the goal gets called and the `signal` unifies with `ok`.

5.1.2 Syntax and Informal Semantics

We introduce two new syntactic constructs. The *effect operations* are Prolog predicate symbols op/n that are declared as such with the following syntax.

```
:- effect op/n.
```

For instance, we declare operation `c/1` to consume a token, `get/1` and `put/1` to respectively retrieve and overwrite an implicit state, and `out/1` to output a term.

The *handler* is a new Prolog goal form that specifies how to interpret effect operations. Its syntax is as follows (square brackets identify optional clauses):

$$
\begin{array}{l}
\texttt{handle } G_0 \texttt{ with} \\
op_1(\bar{X}) \to G_1; \\
\ldots \\
op_n(\bar{X}) \to G_m \\
[\texttt{finally}(G_f)] \\
[\texttt{for}(P_1 = T_1, \ldots, P_n = T_n)]
\end{array}
$$

The `handle` clause scopes G_0 and handles the effects that arise in it. The operation clauses $op_i(\bar{X}) \to G_i$ stipulate that an occurrence of operation $op_i(\bar{X})$ is to be handled by the goal G_i.

Before we explain the optional `finally` and `for` clauses, consider a few ways in which the `out/1` operation can be handled in `hw/0`.

```
hw :- out(hello), out(world).
```

In terms of the exception analogy, `hw/0` throws two `out/1` exceptions. Our first handler intercepts the first `out/1` and does nothing.

```
?- handle hw with (out(X) -> true).
true.
```

A more interesting handler prints the argument of `out/1`.

```
eval(G) :-
    eval(G,Signal),
    ( Signal = shift(Term,Cont) ->
        write('ERROR: Uncaught shift'), fail
    ; true).

eval(shift(Term),Signal) :- !,
                            Signal = shift(Term,true).

eval(reset(G,Cont,Term),Signal) :- !,
     eval(G,Signal1),
    ( Signal1 = ok -> Cont = 0,
                      Term = 0
    ; Signal1 = shift(Term,Cont)),
    Signal = ok.

eval((G1,G2),Signal) :- !,
      eval(G1,Signal1),
      ( Signal1 = ok -> eval(G2,Signal)
      ; Signal1 = shift(Term,Cont),
                  Signal = shift(Term,(Cont,G2))
      ).

eval((C->G1;G2),Signal) :- !,
      ( eval(C,Signal1) ->
            ( Signal1 = ok -> eval(G1,Signal)
            ;  write('ERROR: Undefined action'), fail
            )
      ; eval(G2,Signal)).

eval(Goal,Signal) :- built_in_predicate(Goal),
                     !,
                     call(Goal),
                     Signal = ok.

eval(Goal,Signal) :- clause(Goal,Body),
                     eval(Body,Signal).
```

Figure 5.1: Delimited Control Meta-Interpreter

```
?- handle hw with (out(X) -> writeln(X)).
hello
true.
```

Note that only the first `out/1` is handled; this aborts the remainder of `hw` and the second `out/1` is never reached. To handle all operations, effect handlers support a feature akin to *resumable* exceptions: in the lexical scope of G_i, we can call `continue` to resume the part of the computation after the effect operation (i.e., its continuation). For instance, the next handler resumes the computation after handling the first `out/1` operation and intercepts later `out/1` operations in the same way.

```
?- handle hw with (out(X) -> writeln(X), continue).
hello
world
true.
```

In contrast, the second handler suppresses all output.

```
?- handle hw with (out(X) -> continue).
true.
```

Interestingly, we can invoke the same continuation multiple times, for instance both before and after printing the term.

```
?- handle hw with (out(X) -> continue, writeln(X), continue).
world
hello
world
true.
```

The `finally` Clause The optional `finally` clause is performed when G_0 finishes; if omitted, G_f defaults to `true`.

```
?- handle hw with (out(X) -> writeln(X), continue)
             finally (writeln(done)).
hello
world
done
true.
```

Note that if the goal does not run to completion, the `finally` clause is not invoked.

```
?- handle hw with (out(X) -> writeln(X))
              finally (writeln(done)).
hello
true.
```

The `for` Clause All variables in the operation and `finally` clauses are local to that clause, except if they are declared in the `for` clause.[2] Every $Var = Term$ pair in the `for` clause relates a variable, which we call a *parameter*, that is in scope of all the operations and `finally` clauses with a term whose variables are in scope in the handler context. For instance, the following handler collects all outputs in a list.

```
?- handle hw with (out(X) ->
                      Lin = [X|Lmid],
                      continue(Lmid,Lout))
             finally (Lin=Lout)
             for (Lin = List, Lout=[]).
List = [hello,world].
```

Note that `continue` has one argument for each parameter to indicate which values the parameters take in the continuation. The next handler shows another usage for the finally, which is consuming the input list `Lin` and unifying it with the variables inside the `c/1` "consume" effect.

```
?- handle (c(A), c(B)) with
     (c(X) -> Lin = [X|Lmid], continue(Lmid,Lout))
  finally (Lin = Lout)
  for (Lin = [a,b], Lout = []).
A = a, B = b.
```

5.1.3 Nested Handlers and Forwarding

Nesting algebraic effect handlers is similar to nesting exception handlers. If an operation is not caught by the inner handler, it is *forwarded* to the outer handler. Moreover, if the inner handler catches an operation and, in the process of handling it, raises another operation, then this operation is handled by the outer handler. Let us illustrate both scenarios.

[2] The `for` clause plays a similar role as that in Schimpf's *logical loops* [79].

We can easily define a non-deterministic choice operator `or/2` in the style of TOR [84, 82] in terms of the primitive `choice/1` effect which returns either of the two boolean values `t` and `f`.

```
:- effect choice/1.

or(G1,G2) :- choice(B), (B == t -> G1 ; B == f -> G2).

chooseAny(G) :- handle G with
                  (choice(B) -> (B = t ; B = f), continue).
```

The `chooseAny` handler interprets `choice/1` in terms of Prolog's built-in disjunction `(;)/2`. The semantics of the handler is as follows: whenever an effect `choice(B)` is intercepted, `B` unifies with `t`, then the continuation is called. If the continuation fails, or we asked for an alternative solution, then Prolog backtracks and unify `B` with `f` and recall the continuation again.

```
?- chooseAny(or(X = 1, X = 2)).
X = 1;
X = 2.
```

To obtain more interesting behavior, we can nest this handler with:

```
flip(G) :- handle G with
             (choice(B) -> choice(B1),
                           not(B1,B),
                           continue).

not(t,f).
not(f,t).
```

to flip the branches in a goal without touching the goal's code.

```
?- chooseAny(flip(or(X = 1, X = 2))).
X = 2;
X = 1.
```

What happens is that the inner `flip` handler intercepts the `choice(B)` call of `or/2`. It produces a new `choice(B1)` call that reaches the outer `chooseAny` handler, and unifies `B` with the negation of `B1`, which affects the choice in the `continue`-ation of `or/2`.

Thanks to forwarding, we can also easily mix different effects. For instance, with:

```
writeOut(G) :- handle G with
                  (out(T) -> writeln(T), continue).
```

we can combine output and non-determinism.

```
?- chooseAny(writeOut(or(out(hello), out(world)))), fail.
hello
world
false.
```

Note that the inner `writeOut` handler does not know how to interpret the `choice/1` effect. As a consequence, it (implicitly) *forwards* this operation to the next surrounding handler, `chooseAny`, who does know what to do.

Handlers can be nested to deal with goals that use multiple kinds of effects. For instance, we can combine our basic `out/1` handler, with a handler for an implicit state.

```
:- effect get/1.
:- effect put/1.

stateH(G,Sin,Sout) :-
 handle G with
   (get(S) -> Sin1 = S, continue(Sin1,Sout1)
   ;put(S) -> continue(S,Sout1)
   )
 finally (Sin1 = Sout1)
 for (Sin1=Sin,Sout1=Sout).

modeH(G) :-
 handle G with
   (out(X) -> get(V), ( V == quiet -> true ; out(X)),
              continue).

writeH(G) :- handle G with (out(X) -> write(X),continue).
```

```
?- stateH(writeH(modeH((hw,set(quiet),hw))),verbose,Mode).
hello
world
Mode = quiet.
```

5.1.4 Elaboration Semantics

There is a straightforward elaboration of handlers into the `shift/1` and `reset/3` delimited control primitives for Prolog [80]. For instance, the handler inside the predicate `stateH/3` of the last example of Section 5.1.2 is elaborated into:

```
?- handler42(hw,List,[]).
List = [hello,world].
```

The first argument of the query predicate is the original handler's goal. The second and third arguments are the terms of the `for` clause in the original query. The declaration of the `out/1` operation is elaborated into:

```
out(X) :- shift(out(X)).
```

which shifts the term representation of the operation. The actual handler code is elaborated into a predicate (with a *fresh* name).

```
handler42(Goal,Lin,Lout) :-
  reset(Goal,Cont,Signal),
  ( Signal == 0 ->              %finally clause
      Lin = Lout
  ; Signal = out(X) ->          %handle out(x)
      Lin = [X|Lmid],
      handler42(Cont,Lmid,Lout)
  ; shift(Signal),              %effect forwarding
    handler42(Cont,Lin,Lout)
  ).
```

These predicate arguments are a goal variable and each variable that appear in the `for` clause of the original handler query. The predicate executes the goal in the delimited scope of a `reset/3`, which captures any `shift/1` call. If the goal terminates normally (i.e., `Signal=0`), then the `finally` code is run. If the goal suspends with a `shift/1`, the predicate checks whether the operation matches the handler's operation clause. If so, the clause's body is run. Note that `continue(Lmid,Lout)` has been expanded into a recursive invocation of the handler with the actual continuation goal `Cont`. If the operation does not match, the handler forwards it to the nearest surrounding handler with `shift/1` and continues with the continuation.

Generalisation of the elaboration The example above generalises straightforwardly. Any declaration of an effect operation is elaborated into a predicate definition.

```
:- effect op/n.  ↦   op(X1,...,Xn) :- shift(op(X1,...,Xn)).
```

Also, every handler goal is substituted with a predicate call.

```
handle G0 with
op1 → G1;
...                                  ↦  h(G0,T1,...,Tn).
opn → Gn
finally(Gf)
for(P1 = T1,...,Pn = Tn)
```

where $h/n+1$ is an auxiliary predicate defined as:

```
h(Goal,P1,..,Pn) :-
  reset(Goal,Cont,Signal),
  ( Signal == 0  -> Gf
  ; Signal = op1 -> G'1
  ; ...
  ; Signal = opn -> G'n
  ; shift(Signal), h(Cont,P1,...,Pn)
  ).
```

Here, each G'_i is derived from G_i by replacing all occurrences of `continue(`S_1`,...,`S_n`)` with recursive calls h`(Cont,`S_1`,...,`S_n`)`.

5.2 Optimisation

Section 5.1.4's elaboration of algebraic effects into the delimited control constructs is conveniently straightforward. Unfortunately, capturing the delimited continuation incurs a non-trivial runtime cost. In many simple cases, this cost is quite steep compared to more conventional program transformation approaches. For instance, the implementation of DCGs with delimited control is ten times slower in a tight loop than the traditional term expansion approach [60].

However, the runtime overhead is not inherent in the algebraic effects and handlers approach, and we can obtain competitive performance through optimised compilation. This section presents our optimisation approach, which aims to eliminate most uses of delimited control. The optimisation consists of two

collaborating transformation approaches: rewrite rules (Section 5.2.2) and partial evaluation (Section 5.2.3). We use term rewrite rules to simplify handler goals and possibly eliminate the handler constructs altogether. These rules depend on an effect system (Section 5.2.1) that infers which effects can or cannot be generated by a goal. Partial evaluation complements the rewrite rules by specialising handled predicate calls which enables, in particular, the specialisation of (mutually) recursive predicates.

5.2.1 Effect System

Driving our optimisation is an *effect system* that associates with each goal G an *effect set* E that denotes which effects the goal may call.

Effect Sets In order to cater for modular programs, effect sets E are not elements of the powerset lattice over the closed set OP of locally known effect operation symbols op/n. Instead, we use the powerset lattice over an open-ended set of effect operations augmented with the additional top element All. This allows us to express the effects of unknown goals and unknown effect operations in an abstract manner.

Hence, we denote effect sets in one of two forms: $\bigcup_i op_i/n_i$ or $All - \bigcup_i op_i/n_i$. The former is an explicit enumeration of effect operations, while the latter expresses the dual: all but the given effect operations.

The $\in$ relation as well as the functions $\cup$ and $-$ are extended from the powerset lattice to our augmented version.

$$op/n \in E \equiv \begin{cases} \exists i : op/n = op_i/n_i & , E = \bigcup_i op_i/n_i \\ \forall i : op/n \neq op_i/n_i & , E = All - \bigcup_i op_i/n_i \end{cases}$$

$$E_1 - E_2 \equiv \begin{cases} \{op_i/n_i \mid op_i/n_i \notin E_2\} & , E_1 = \bigcup_i op_i/n_i \\ & , E_2 = \bigcup_j op_j/n_j \\ All - \bigcup_i op_i/n_i \cup \bigcup_j op_j/n_j & , E_1 = All - \bigcup_i op_i/n_i \\ & , E_2 = \bigcup_j op_j/n_j \\ \{op_j/n_j \mid op_j/n_j \notin \bigcup_i op_i/n_i\} & , E_1 = All - \bigcup_i op_i/n_i \\ & , E_2 = All - \bigcup_j op_j/n_j \\ (\bigcup_i op_i/n_i \cap \bigcup_j op_j/n_j) & , E_1 = \bigcup_i op_i/n_i \\ & , E_2 = All - \bigcup_j op_j/n_j \end{cases}$$

$$
\begin{array}{lcll}
E - \emptyset & = & E & \\
op_i - op_j & = & \emptyset & \textit{if } i = j \\
op_i - op_j & = & op_i & \textit{if } i \neq j \\
\bigcup_I op_i - op_j & = & \bigcup_J op_j & (J = I - \{j\}) \\
E - (E_1 \cup E_2) & = & (E - E_1) - E_2 & \\
E - All & = & \emptyset & \\
E \cup All & = & All & \\
E \cup \emptyset & = & E & \\
\bigcup_I op_i \cup op_1 & = & \bigcup_J op_j & (J = I \cup \{j\}) \\
op_k \cup (All - \bigcup_I op_i) & = & All - (\bigcup_J op_j) & (J = I - \{k\})
\end{array}
$$

Figure 5.2: Normalisation rules

$$
E_1 \cup E_2 \equiv \begin{cases}
\bigcup_i op_i/n_i \cup \bigcup_j op_j/n_j & , E_1 = \bigcup_i op_i/n_i \\
 & , E_2 = \bigcup_i op_j/n_j \\
All - (\bigcup_i op_i/n_i \cap \bigcup_j op_j/n_j) & , E_1 = All - \bigcup_i op_i/n_i \\
 & , E_2 = All - \bigcup_j op_j/n_j \\
All - (\bigcup_i op_i/n_i - \bigcup_j op_j/n_j) & , E_1 = All - \bigcup_i op_i/n_i \\
 & , E_2 = \bigcup_j op_j/n_j \\
E_2 \cup E_1 & , otherwise
\end{cases}
$$

$$
\begin{array}{lll}
E ::= \emptyset & & \text{No effects} \\
\quad \mid op/n & & \text{Operation } op/n \\
\quad \mid E \cup E & & \text{Union of effects} \\
\quad \mid E - E & & \text{Difference of effects} \\
\quad \mid All & & \text{All effects}
\end{array}
$$

Here $\emptyset$ means that the goal is *pure*. The effect expression op/n means that the goal may call op/n. The union and difference have their obvious meaning. Finally, the effect expression All abstracts over all possible effect operations.

Figure 5.2 lists the axioms of the equational theory of effect expressions. They are normalisation rules such that when they are used from left to right as rewrite rules, they turn the expression E into the forms $\emptyset$, op_i/n_i or $All - \bigcup_i op_i/n_i$.

Effect System We use these functions over effect sets in the definition of our effect system judgement $E_c \vdash G : E$. This judgement expresses that goal G calls only effect operations from the effect set E, provided that `continue` calls only

$$\boxed{E_c \vdash G : E}$$

$$E_c \vdash X : All \ \ (\text{E-VAR}) \qquad \frac{op/n \in OP}{E_c \vdash op(T_1, \ldots, T_n) : op/n} \ (\text{E-OP})$$

$$\frac{E_c \vdash G_1 : E_1 \qquad E_c \vdash G_2 : E_2}{E_c \vdash (G_1, G_2) : E_1 \cup E_2} \ (\text{E-CONJ}) \qquad \frac{\begin{array}{c} E_c \vdash G_1 : E_1 \\ E_c \vdash G_2 : E_2 \end{array}}{E_c \vdash (G_1; G_2) : E_1 \cup E_2} \ (\text{E-DISJ})$$

$$E_c \vdash \texttt{continue}(\bar{T}) : E_c \ \ (\text{E-CONT}) \qquad E_c \vdash \texttt{true} : \emptyset \ \ (\text{E-TRUE})$$

$$\frac{p(S_1, \ldots, S_n) \ \texttt{:-} \ G \quad E_c \vdash G : E}{E_c \vdash p(T_1, \ldots, T_n) : E} \ (\text{E-PRED})$$

$$\frac{\begin{array}{c} E^* = (E_0 - \bigcup_i op_i) \cup E_f \cup \bigcup_i E_i \\ E_c \vdash G_0 : E_0 \quad E_c \vdash G_f : E_f \quad E^* \vdash G_i : E_i \ (\forall i) \end{array}}{E_c \vdash \left(\begin{array}{c} \texttt{handle } G_0 \texttt{ with} \\ \overline{op_i(\bar{X}) \to G_i} \\ \texttt{finally}(G_f) \texttt{ for}(G_s) \end{array} \right) : E^*} \ (\text{E-HANDLE})$$

Figure 5.3: Effect Inference Rules

effect operations from the effect set E_c. Since `continue` is not defined for a top-level goal, we may assume any value for its E_c. Hence, for convenience, we always take $E_c = \emptyset$ for top-level goals G and just write $\vdash G : E$.

Figure 5.3 defines this judgement using inference rules. Rule (E-VAR) expresses that a variable (i.e., unknown) goal may call All effect operations. Rule (E-OP) states that a known effect operation calls itself. Rules (E-CONJ) and (E-DISJ) combine the effects of their subgoals. Rule (E-TRUE) expresses that the goal `true`, as an example for other built-ins, is op-free. Rule (E-CONT) captures the invariant that `continue` has the E_c effect. In Rule (E-PRED) the effect of a user-defined predicate is the effect of its body. Finally, most of the complexity of the inference system is concentrated in Rule (E-HANDLE) that deals with a handler goal. The rule expresses that the handler goal forwards all the effect operations E_0 of the goal G_0 it handles, except for the ones that the handler takes care of, $\bigcup_i op_i/n_i$.

Also, the handler may introduce additional calls to effect operations in its operation and `finally` clauses. Also, note that calls to `continue` in the operation clauses have the same effect as the handler goal itself; they are essentially recursive calls after all.

Here are a few examples:

$\vdash$ `hw` : `out/1`
$\vdash$ `handle hw with (out(X) -> writeln(X))` : $\emptyset$
$\vdash$ `handle Y with (out(X) -> writeln(X))` : $All -$ `out/1`

5.2.2 Rewrite Rules

We use the information of the effect system and the syntactic structure of goals to perform a number of handler-specific optimisations. We denote these optimisations in terms of semantics-preserving equivalences $G_1 \equiv G_2$ that we use as left-to-right rewrite rules. Figure 5.4 lists our rewrite rules in the form of inference rules where conditions on the inferred effects are written above the bar.

Rule (O-Disj) captures the fact that effect handling is orthogonal to disjunction to specialise the branches of a disjunction separately. The proof of soundness for this rule can be found in Appendix B.3. There are two rules for conjunction. Rule (O-Conj) pulls the first goal G_1 of a conjunct out of the handler if it does not call any of the handler's operations. This covers both the case where G_1 is an op-free goal and the case where the handler forwards all the operations in G_1. The second rule for conjunction, Rule (O-Op), statically evaluates the special case where the first goal is an operation dealt with by the handler. This consists of three parts: 1) the unification of the formal and actual parameters, 2) the unification of the formal and actual operation arguments, and 3) calling the operation clause's goal. Note that we substitute all calls to `continue`$(\bar{U})$ (for any $\bar{U}$) in this last goal with the second conjunct wrapped in the handler; note that the arguments $\bar{U}$ become the new actual parameters. In the process we are careful to *freshen* all the local logical variables that are used.

Rule (O-Drop) removes spurious operation clauses from the handler; it only retains those that correspond to operations that the goal may call. In the case that no operation clauses remain, Rule (O-Triv) dispenses with the handler altogether. This amounts to unifying the formal and actual parameters and calling the `finally` goal.

Finally, the most complex rule of all, Rule (O-Merge), merges two nested handlers into one single handler and thereby eliminates expensive forwarding of operations. At first, it might seem trivial to merge two handlers: We simply merge all the components of the two handlers pairwise. There is an obvious simplification to

$$\begin{pmatrix} \texttt{handle } (G_1\texttt{;}G_2) \\ \texttt{with } \overline{op \to G}\texttt{;} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \equiv \begin{pmatrix} \texttt{handle } G_1 \\ \texttt{with } \overline{op \to G}\texttt{;} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \texttt{;} \begin{pmatrix} \texttt{handle } G_2 \\ \texttt{with } \overline{op \to G}\texttt{;} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \quad \text{(O-Disj)}$$

$$\frac{E_c \vdash G_1 : E_1 \qquad E_1 \cap \bigcup_i op_i = \emptyset}{\begin{pmatrix} \texttt{handle } (G_1\texttt{,}G_2)\ \texttt{with} \\ \overline{op \to G}\texttt{;} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \equiv G_1\texttt{,} \begin{pmatrix} \texttt{handle } G_2\ \texttt{with} \\ \overline{op \to G}\texttt{;} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix}} \quad \text{(O-Conj)}$$

$$\frac{(op(\bar{S}) \to G_i) \in \overline{op \to G} \qquad \mathit{freshen}(\bar{P}_F, \bar{S}, G_i) = (\bar{P}'_F, \bar{S}', G'_i)}{\begin{pmatrix} \texttt{handle } (op(\bar{T}), G_c) \\ \texttt{with } \overline{op \to G} \\ \texttt{finally}(G_f) \\ \texttt{for}(\bar{P}_F\texttt{=}\bar{P}_A) \end{pmatrix} \equiv \bar{P}'_F\texttt{=}\bar{P}_A\texttt{,}\bar{T}\texttt{=}\bar{S}'\texttt{,}G'_i \begin{bmatrix} \texttt{continue}(\bar{U}) \mapsto & \texttt{handle } G_c\ \texttt{with} \\ & \texttt{with } \overline{op \to G} \\ & \texttt{finally}(G_f) \\ & \texttt{for}(\bar{P}_F\texttt{=}\bar{U}) \end{bmatrix}} \quad \text{(O-Op)}$$

$$\frac{E_c \vdash G : E \qquad \overline{op' \to G'} = (\overline{op \to G}) \cap E}{\begin{pmatrix} \texttt{handle } G \\ \texttt{with } \overline{op \to G} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \equiv \begin{pmatrix} \texttt{handle } G \\ \texttt{with } \overline{op' \to G'} \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix}} \quad \text{(O-Drop)}$$

$$\begin{pmatrix} \texttt{handle}(G)\ \texttt{with } \emptyset \\ \texttt{finally}(G_f)\ \texttt{for}(G_s) \end{pmatrix} \equiv G\texttt{,}G_s\texttt{,}G_f \quad \text{(O-Triv)}$$

$$\frac{\text{see text} \qquad \overline{op'_2 \to G'_2} = (\overline{op_2 \to G_2}) - \overline{op_1}}{\begin{pmatrix} \texttt{handle } \begin{pmatrix} \texttt{handle } G \\ \texttt{with } \overline{op_1 \to G_1} \\ \texttt{finally}(G_{1,f}) \\ \texttt{for}(G_{1,s}) \end{pmatrix} \\ \texttt{with } \overline{op_2 \to G_2} \\ \texttt{finally}(G_{2,f}) \\ \texttt{for}(G_{2,s}) \end{pmatrix} \equiv \begin{pmatrix} \texttt{handle } G\ \texttt{with} \\ \overline{op_1 \to G'_1} \\ \overline{op'_2 \to G'_2} \\ \texttt{finally } G'_{1,f} \\ \texttt{for}(G_{1,s}\texttt{,}G_{2,s}) \end{pmatrix}} \quad \text{(O-Merge)}$$

Figure 5.4: Optimisation Rules for effect handlers

perform in the process: we can drop all outer handler's operation clauses that overlap with any of the inner handler's clauses, as the inner handler takes precedence over the outer one.

There is a further subtle issue that has to be taken account in order to preserve the original semantics. The `finally` goal $G_{1,f}$ and the operation clause goals $\overline{op_1}$ may call operations that are originally intercepted by the outer handler. We have to make sure that this remains the case. For that reason, we adjust those goals to $G'_{1,f}$ and $\bar{G}'_1$ in the merged handler. Let us explain these adjustments for the different forms of operation clause goals $G_{1,i}$ that we consider.

1. The operation goal $G_{1,i}$ is of the form $G_{1,i,a}$`,continue(`$\bar{V}$`)` where $G_{1,i,a}$ does not contain any call to `continue`. We wrap the initial part of the goal in the outer handler and finally proceed with `continue`.

$$G'_{1,i} = \left(\begin{array}{c} \texttt{handle}\ G_{1,i,a}\ \texttt{with} \\ \overline{op_2 \to G_2} \\ \texttt{finally continue}(\bar{V}, \bar{P}_{2,F}) \\ \texttt{for}\ (\bar{P}_{2,F}, \bar{P}'_{2,F}) \end{array} \right)$$

2. The operation goal $G_{1,i}$ does not contain a call to `continue`. In this case, we wrap the entire goal in the outer handler and make sure to call the outer handler's final goal.

$$G'_{1,i} = \left(\begin{array}{c} \texttt{handle}\ G_{1,i}\ \texttt{with} \\ \overline{op_2 \to G_2} \\ \texttt{finally}\ G_{2,f} \\ \texttt{for}\ (\bar{P}_{2,F}, \bar{P}'_{2,F}) \end{array} \right)$$

Similarly, we adapt the final goal $G_{1,f}$ to

$$G'_{1,f} = \left(\begin{array}{c} \texttt{handle}\ G_{1,f}\ \texttt{with} \\ \overline{op_2 \to G_2} \\ \texttt{finally}\ G_{2,f} \\ \texttt{for}\ (\bar{P}_{2,F}, \bar{P}'_{2,F}) \end{array} \right)$$

5.2.3 Partial Evaluation

We use a custom partial evaluation approach to expose more optimisation opportunities for the rewrite rules and to deal with recursive predicates. Our partial evaluation is targeted at predicate calls that are handled. Consider the following simple DCG example that checks if a phrase is a succession of the terminals `ab`:

```
:- effect c/1.
ab.
ab :- c(a), c(b), ab.
query(Lin) :-
  handle ab with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=[]).
```

Here we abstract the goal `handle ab with ...` into a fresh predicate (say `ab0/2`), which makes abstraction of the actual handler parameters. This yields the new definition of `query/1`:

```
query(Lin) :- ab0(Lin,[]).
```

At the same time we unfold the definition of `ab/0` in the newly created predicate `ab0/2`. Because `ab/0` has two clauses, this means that `ab0/2` bifurcates similarly.

```
ab0(Lin,Lout) :-                  %first ab clause
 handle true with
 (c(X) -> Lin1=[X|Lmid],
          continue(Lmid,Lout1))
 finally (Lin1 = Lout1)
 for (Lin1=Lin,Lout1=Lout).


 ab0(Lin,Lout) :-                  %second ab clause
  handle (c(a), c(b), ab) with
  (c(X) -> Lin1=[X|Lmid],
           continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).
```

This unfolding exposes new rewriting opportunities. Using the Rules (O-DROP) and (O-TRIV), the first clause specialises to `Lin1=Lin, Lout1=Lout, Lin1=Lout1`. In the second clause, a double use of Rule (O-OP) deals with the `c/1` operations. This leaves a recursive invocation of `ab/0`, wrapped in the handler. Now the partial evaluation kicks in again, realises that this is a variant of the earlier specialization and ties the knot with a recursive call to `ab0/2`. After further clean-up of the unifications, we get:

```
ab0(L,L).
ab0([a,b|Lmid],Lout) :- ab0(Lmid,Lout).
```

There is no trace of delimited control left. Moreover, this is precisely the tight code that the traditional DCG yields. A step-by-step elaboration for this example is in Appendix B.1

5.3 Evaluation

We evaluate the usefulness of our optimisation approach experimentally on a set of benchmarks. All results were obtained on an Intel Core i7 with 8 GB RAM running hProlog 3.2.38 on Ubuntu 14.04.

The first experiment concerns the `ab` program of Section 5.2.3. Figure 5.5 lists the timings (in ms) for different input sizes obtained with three different versions of the program: the traditional DCG implementation (based on SICStus), the elaborated handler implementation and the optimised handler implementation. Clearly, the original use of delimited control slows the program down by more than an order of magnitude. Fortunately, our optimisation eliminates all uses of delimited control and matches the traditional implementation's performance.[3]

Input Size	Traditional	Elaboration	Optimised
1×10^3	0	2	0
1×10^4	1	4	1
1×10^5	8	37	5
1×10^6	32	321	29
2×10^6	67	635	58
5×10^6	150	1821	146
1×10^7	300	4757	297
1×10^8	2953	47632	2922

Figure 5.5: DCG benchmark results in ms

The second experiment considers three scenarios with nested handlers. Figure 5.6 lists the runtime results (in ms) for different input sizes of different versions: the plain elaborated program, the program optimised with only the rewrite rules and the program optimised with both rewrite rules and partial evaluation.

The first benchmark, `state_dcg`, extends the `ab` example with an implicit state that is incremented with every occurrence of `ab` in the input and because the rewrite rules merge the two handlers in this benchmark, they generate an almost two-fold speed-up.

With partial evaluation, the speed-up of around two orders of magnitude is much more dramatic. The main reason is that delimited control is again eliminated.

[3] Thanks to more aggressive inlining it is even slightly faster.

The details of `state_dcg` handler and an overview on how the handler merging optimisation, and the partial evaluation work in this example are in Appendix B.2.

The second benchmark adds an inner-most dummy handler for an unused `foo/0`. This benchmark aims to assess the cost of forwarding. In the plain elaborated version, we can see there is a significant overhead. Thanks to the rewriting, the three handlers are again merged and most of the overhead of the spurious handler disappears – the only remaining cost is the spurious `foo/0` operation clause. Finally, with partial evaluation, all trace of the `foo/0` is eliminated.

The third benchmark re-implements the calculator example of Dragan et al. [38] with two handlers, one to manage an implicit stack and one to one for an implicit register. The behaviour is similar to the other two benchmarks: merging the handlers roughly halves the runtime and partially evaluating them speeds up the code by two orders of magnitude.

Program Name	Input Size	Elaborated	Rewriting	Rewriting + PE
state_dcg	1×10^3	3	2	0
state_dcg	1×10^4	20	11	0
state_dcg	1×10^5	151	63	3
state_dcg	1×10^6	1879	604	37
state_dcg	2×10^6	2814	1208	75
state_dcg	5×10^6	7919	4348	186
state_dcg	1×10^7	29695	18094	375
state_dcg_foo	1×10^3	4	3	0
state_dcg_foo	1×10^4	23	11	0
state_dcg_foo	1×10^5	358	61	3
state_dcg_foo	1×10^6	4666	670	37
state_dcg_foo	2×10^6	8777	1350	75
state_dcg_foo	5×10^6	30026	4551	186
calculator	1×10^3	4	3	1
calculator	1×10^4	30	16	1
calculator	1×10^5	307	78	10
calculator	1×10^6	1195	761	57
calculator	2×10^6	3015	1525	110
calculator	5×10^6	12326	6114	247

Figure 5.6: Runtimes of nested-handler benchmarks in ms

5.4 Related Work

Language Extensions Various Prolog language extensions have been proposed concerning program transformations. Van Roy has proposed *Extended DCGs* [75] to thread multiple named accumulators. Similarly, Ciao Prolog's structured state

threading [38] enables different implicit states. Algebraic effects and handlers can easily provide similar functionality.

Schimpf's *logical loops* [79] approach has been very influential on our handler design, in particular regarding the elaboration into recursive predicates and the notions of locally fresh variables and parameters. Of course, both features originate in distinct paradigms: logical loops are inspired by imperative loops, while handlers originate in the functional programming paradigm.

Control Primitives Various works have considered extensions of Prolog that enable control-flow manipulation. Before the work of Schrijvers et al. [80], Tarau and Dahl [90] already allowed the users of BinProlog to access and manipulate the program's continuation.

Various coroutine-like features have been proposed in the context of Prolog for implementing alternative execution mechanisms, such as constraint logic programming and delay. Nowadays most of these are based on a single primitive concept: attributed variables [37, 48, 58, 20]. Like delimited control, attributed variables are a very low-level feature that is meant to be used directly, but is often used by library writers as the target for much higher-level declarative features.

Algebraic Effects and Handlers in Prolog Schrijvers et al. [84] have previously appealed to a functional model of algebraic effects and handlers to derive a Prolog implementation of search heuristics [82]. The proposed concepts in this chapter enable a direct Prolog implementation that avoids this detour into a different paradigm.

5.5 Conclusion and Discussion

This chapter defined algebraic effects and handlers for Prolog as a high-level alternative to delimited control for implementing custom control-flow and dataflow effects.

In order to avoid undue runtime overhead of capturing delimited continuations, we provide an optimised compilation approach based on partial evaluation and rewrite rules. In order to apply these re-writing rules, we developed an effect system that guides the rules showing which effects are triggered by which terms. Our experimental evaluation shows that this approach greatly reduces the runtime overhead.

In this section, we discuss the similarities and the differences we have observed between the two languages in focus. We first discuss the conceptual differences of introducing effects and handlers into the two languages and we also note the advantages that we ported from one language to the other.

5.5.1 Eff vs Prolog: Concepts

Eff, on the one hand, is a functional language that is based on OCaml. Therefore, when effects and handlers are introduced, they can be seen as functions. The captured continuations and the results of handling computations are also functions. This facilitates parameters passing and accumulation of results which allows contextualising a behaviour of a specific part of the Eff program.

Prolog, on the other hand, does not support functions since it uses unification and backtracking to resolve a query. A Prolog variable can be unified only once, and in order to un-unify that variable, we need backtracking. It does not support function passing or re-assigning values to a variable. That is why we use local variables inside a handler definition and unify it with external global variables, using the keyword `for`, to connect them to the top-level Prolog code.

Prolog's calculus can be modelled in Functional Programming. The work of Spivey and Seres [87] showed techniques of embedding Prolog's semantics in Haskell. Later, the work of Schrijvers et al. [84] modelled Prolog's calculus using monads and effect handlers. The concept of their work is representing unification and backtracking as side effects and then using handlers to give these effects semantics according to Prolog's semantics. They have extended Prolog's search techniques by adding different handlers that interpret these effects differently. Encoding such different search techniques in Prolog itself without the aid of delimited control constructs can be a tedious job since these techniques would have to be implemented using Prolog's unification and backtracking only.

Techniques adapted from Eff to Prolog The overall high-level syntax of effect handlers introduced to Prolog is largely inspired by Eff's syntax. The `handle...with...` keywords for Prolog are very similar to those `with...handle...` for Eff. Listing the different effects in the handler is also inspired by Eff. The `finally` clause in Prolog resembles the value clause in Eff.

The notion of type systems in Prolog is absent. Some work in the literature, nevertheless, introduces type systems into Prolog as an add-on, such as the work of Mycroft and O'Keefe [57]. Though native Prolog variables can be unified with any term. When we introduced effect handlers to Prolog, we needed to provide a system that infers which effects can be thrown by which terms, in order to facilitate

Eff Optimisations	Prolog Optimisations
With-Handled-Op	(O-Op)
With-Pure	(O-Drop) + (O-Triv)

Figure 5.7: Similarities of Optimisations

the optimisations that we wanted to implement. Therefore, we adapted the Eff's effect system to Prolog after taking into considerations the differences between the two calculi.

Eff and Prolog continuations In Eff's calculus, continuations are functions. They are identified by a variable that stores the continuation function. This variable can be accessed, passed around and captured and in general, used as a first class value which increases the expressiveness of the language. In Prolog, on the contrary, the effect handlers calculus we presented does not allow accessing the continuation itself as we can only use the keyword `continue` to invoke it. This limits the freedom of using the continuation in the language. If we looked at the compilation process of effect handlers in Prolog, we see that the `continue` key-word gets translated into a recursive call for the predicate representing the handler. The continuation itself is a variable within this predicate that does not escape. In theory, the delimited control construct `reset/3` can give the programmer the access to the continuation variable. However, restricting that access allows for a more structured way of writing the programs and also enables more optimisations.

5.5.2 Eff vs Prolog: Optimisations

Similarities between Eff and Prolog Optimisations Both Chapter 3 and this chapter introduce sets of rewriting rules that have a similar goal, namely to eliminate handler code or open more possibilities to eliminate handler code, so that in return the output code is more efficient. In this part, we discuss the similarities between the optimisation rules of Figure 3.2 in Chapter 3 and the rules of this chapter listed in Figure 5.4. We show the similarities between the two optimisation techniques in Figure 5.7.

For example, in Figure 3.2 we presented Rule With-Handled-Op that partially evaluates a handler handling an effect. The similarity is evident in Figure 5.4 where Rule O-Op partially evaluates a handler with an effect in the beginning of its conjunctive goal.

In the rule With-Handled-Op, c_{op} gets executed after substituting the operation variable with the input of `Op`v and the continuation with a function that executes

the return clause of the handler. The Prolog rule is rather similar. If the rest of the conjunctive goal `true`, then the newly generated handler reduces to the `finally` clause of the handler.

Another similarity is the rule WITH-PURE which checks if a computation is pure concerning the handler surrounding it. This is achieved by checking the intersection between the handler's effect and the computation's effects. If the intersection is empty, then only the return clause of the handler is relevant, and it is added to the end of the computation. In the Logic Programming setting, The previous rule is a combination of the two rules (O-DROP) and (O-TRIV). The first rule checks the intersection of effects between the goal and the handler; then it drops the effects in the handler that are do not appear in the goal. The second checks if there are no remaining effects handled by the handler. If this is the case, then it drops the handler and applies the unification between the local and global variables and also the `finally` clause.

Differences between rewrite rules of Eff and Prolog We also discovered differences between the two paradigms regarding the introduction and optimisation of effect handlers. Prolog's calculus for effect handlers is more restricted. This enables for more aggressive optimisations. For example, the handler merging rule (O-MERGE) removes a layer of delimited control `reset/shift` code when elaborating to Prolog. The continuations do not have types in Prolog, which allows easy merging the handlers. However, in EFF, the types of handlers, the continuation variables and the effects themselves make merging more complicated.

Merging handlers in Functional Programming has been discussed in the literature. The work of Wu and Schrijvers [95] introduced a technique that fuses (or merges) nested algebraic effect handlers. The paper uses algebras for free monads to represent handlers and uses folds and fold/build fusion techniques [31] to merge two or more nested handlers. However, it does not give a proper compilation scheme.

The handlers merging techniques used in that work can be adapted into our EFF compiler. However, the calculus introduced should be modified in order to fit EFF's calculus and provide a compilation scheme that fits the other optimisation techniques that occur in EFF. We have not included handlers merging into the optimisation techniques so far because we focus on eliminating one handler at a time. If a program with nested handlers is compiled using our current optimisations, the compiler will try to eliminate the inner handler first then optimise the outer levels afterwards. However, the optimisations stop whenever the code that needs to be handled is a variable or none of the re-writing rules is applicable. Thus, our compilation can be made more powerful in the future by employing handlers merging, improving the overall efficiency of the compiled program.

Contributions This chapter is based on my paper with Schrijvers [78]. My contributions are the following:

- Design of the effect handlers syntax in Prolog.
- Elaboration from effect handlers syntax to delimited control.
- The optimisation techniques discussed in this chapter.
- Implementation of the system in Prolog as a source to source compiler. The implementation is available on `http://github.com/ah-saleh/prologhandlers`.
- Testing and benchmarking.
- Proof of the soundness of the O-DISJ rule in Appendix B.

Chapter 6

Conclusion and Future Work

In this chapter, we briefly recall the aim of this thesis, and summarise our research contributions in the context of both Eff and Prolog. Additionally, we discuss the ongoing work that builds on the content of this thesis, as well as potential ideas for future work.

6.1 Summary of Contributions

Looking back to the research question we have presented in the introductory section, we see that the primary goal of this thesis is to optimise the compilation of algebraic effect and handlers, while maintaining the modularity that it provides. Our work is divided into two parts. The first focuses on the Functional Programming Programming paradigm and the second focuses on the Logic Programming paradigm. For the former we use Eff, while for the latter we use Prolog.

6.1.1 Effect handlers in Functional Programming

Optimising Eff compiler The compiler of Eff uses a free monad representation aas a structure for the compiled code. This proved to be costly during run-time. The main aim of Chapter 3 is to find different optimisation techniques that decrease the run-time of Eff programs.

As a result, the run-time performance of programs written in Eff can now compete with other different functional systems' run-times, while having the extra advantage

of abstraction and modularity of plugging in different handlers to give effects different behaviours. We developed three main techniques that work together to reduce (and occasionally eliminate) the impact of the free monad representation:

- Term-rewriting rules: we use these rules to partially evaluate and eliminate as much handler code as possible.
- Function specialisation: this technique allows further term-rewriting optimisation of handlers that are nested within functions. Without this technique, such handlers are not accessible and therefore can not be optimised.
- Purity aware compilation: we use the effect system of Eff to compile non-effectful Eff code directly into native OCaml code, instead of using the free monad representation.

We implement all the optimisations in Eff. Then, we benchmark the optimised compiler versus hand-written OCaml code and other systems:

- The combination of all three optimisation techniques generates an optimised OCaml code that competes with hand-written code (1x-1.5x). A great factor for this speed-up is the successful removal of all traces of handlers through the optimisation process, resulting in non-monadic OCaml code.
- Eff was consistently 25-30% faster than its closest competitor, Multicore OCaml.

However, due to the implicit sub-typing of the original Eff calculus, many bugs in the system were introduced during optimisation. These bugs occur because the compiler cannot know the precise type of a term during optimisation. Moreover, in the function specialisation optimisation, pinpointing the type of the new specialised function was not possible due to the same issue. This problem showed up when we tried to optimise larger Eff programs such as parser combinators using effect handlers.

These problems point to the need for an explicitly-typed language, where we can ensure that an implmentation of our optimisations is type-preserving. Such a language can assist in eliminating a large class of bugs and make the implementation of our optimisations significantly more reliable.

Eff calculus with explicit sub-typing Chapter 4 provides an entirely new calculus for Eff which uses explicit typing at its core calculus. The proposed calculus aims to facilitate the optimisation of the compilation process which, in return, produces more efficient code.

We propose three different calculi that interact with each other to provide a robust system our optimisations can be implemented in.

- ImpEff: an implicitly-typed calculus. A programmer writes the program in ImpEff syntax; then we elaborate the program into ExEff.
- ExEff: an explicitly-typed language in which subtyping is explicitly stated in terms using casts with a subtyping proof. The main aim of ExEff is to facilitate the optimisation process. The optimisations can access the exact type of every term during compilation. Hence, generating new optimised terms with types built on existing terms should be straightforward. After doing all source transformations in ExEff, we elaborate the transformed code into SkelEff.
- SkelEff: this language omits all additional typing information that is included in ExEff. This makes SkelEff programs significantly smaller and easy to translate to many different target languages, such as plain OCaml.

Along with these languages, we have also provided a type-and-effect inference system with a constraint solving algorithm to resolve the types and the effects of the terms during the elaboration from ImpEff to ExEff. We have implemented a new Eff compiler that incorporates the three new calculi we provided for Eff.

6.1.2 Effect handlers in Logic Programming

In Prolog, we can introduce effect handlers to manipulate the control flow of a strictly controlled language where only cuts can be used to exploit program flow. The delimited control constructs presented by Schrijvers et al. [81] opened the gates for a a better understanding of how we can manipulate the control flow of Prolog programs. The hindrance of the delimited control constructs is that they are inefficient due to copying from the heap whenever a `shift` is triggered. Therefore, effect handlers in Prolog provide a more systematic way of using control flow constructs, thus eliminating many of the drawbacks of the more primitive delimited control constructs.

Moreover, Effect handlers provide an extra layer of modularity when used.

Effect handlers introduction In the first part of Chapter 5, We addressed the introduction of effect handlers in Prolog by:

- Introducing syntax for algebraic effect handlers in Prolog.

- Showing an elaboration of handlers syntax into delimited control by means of a meta-interpreter.

The direct elaboration of effect handlers into delimited control constructs results in inefficient compiled code. We addressed this problem in the second part of Chapter 5 by devising two different optimisation techniques:

- The first technique consists of term rewriting whose main aim is to eliminate handler code during compilation. These rules are accompanied by an effect system we developed to realise which terms can throw effects. This information is used to guide the application of rewriting rules.
- The second method uses a partial evaluation that works together with the rewriting rules, exposing more handlers and allowing more possibilities for optimisations.

Applying the optimisations to a program containing effect handlers eliminates the inefficiency of delimited control in the output code while preserving the semantics of the input program. Our benchmarks show that, after applying the optimisation techniques, we reach the same level of efficiency as a hand-written Prolog implementation. We have also implemented this system as a source-to-source transformation compiler on top of Prolog.

6.2 Ongoing and Future work

We have introduced in this thesis several optimisation techniques in two different programming paradigms. Additionally, we introduced a new calculus for Eff. We see much potential for further development of both paradigms of this thesis. Most of the ideas we present in this section are already under active development.

6.2.1 Import optimisations to new Eff calculus

One of the immediate paths to follow is to port and implement the optimisation techniques presented in Section 3.3 in the new calculus of Eff. The probability to get clean and fast code with this approach is high.

Before this approach can be applied, The term rewriting rules should get adapted to work with the syntax of ExEff. For example, the following rule from Figure 3.2

$$\frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \rightsquigarrow c_r[v/x]}\ \textsc{With-Ret}$$

can be modified, to fit the ExEff calculus, as

$$\frac{h = \{\texttt{return}\ (x : T) \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \rightsquigarrow c_r[v/x]}\ \textsc{With-Ret-ExEff}$$

However, this example is rather simple. The main challenge is to devise a technique to maintain the consistency of the type coercions associated with every term. The rewriting rule has to make sure that the coercions of the re-written term should be consistent with the coercions of the original one.

6.2.2 Handler Merging in Eff

In Section 5.5, we discussed the difficulty of merging (or flattening) nested handlers in Eff. However, one can adapt in Eff the techniques introduced by Wu and Schrijvers [95] for merging nested algebraic effect handlers. Then, providing a terminating compilation scheme on top of them would be needed.

Handlers merging can be used to speed up programs that have nested handlers when the compiler does not know how to optimise the computation the handler is handling. For example, if the handler is handling a function application where the function is a variable, `handle f x with ....`, then we cannot proceed with any optimisation discussed in the previous chapters, since the compiler does not have the information of what this variable will be bound to during compilation time. Therefore, the only thing an optimiser can do is to flatten the handlers since a single handler instead of nested ones means only one layer of continuation passing which is more efficient.

6.2.3 Explicit subtyping for polymorphic effects

Our main focus in this thesis is targeting monomorphic algebraic effects and handlers. To declare an effect, we provide a concrete input type and output type. However, lately, a new wave of research started experimenting with abstraction over effects

and introducing polymorphic effects [85, 7]. Instead of defining an effect with a concrete type, we can define it with a type variable as follows:

$$\texttt{effect}\ e : \alpha \to \alpha$$

This introduction introduces new challenges such as which handler to pick during runtime to handle a polymorphic effect, or how can we proof the type safety property for languages that support polymorphic effects and handlers.

Extending Eff with polymorphic effects increases the expressiveness of the language. However, adapting the type-and-effect inference algorithm to handle that added expressiveness is a new challenge. In ExEff, for example, extensions for coercions to support polymorphic effects are needed in order to keep all the type and effect information explicit to the term. The constraint generation, inference and unification algorithms need to be updated. The new calculus will also need to prove type safety and preservation properties.

If a robust type-and-effect system is implemented for polymorphic effects and handlers, the modified optimisation techniques can be ported to the new calculus in a natural way.

6.2.4 Non tail-recursive continuations in Prolog

The Prolog examples discussed in Chapter 5 are tail-recursive examples. For instance, the `ab_inc` predicate is a tail recursive predicate, for which, after optimisation, the generated code is free from delimited control constructs. That happens due to the possibility of tying the recursive knot. Non-tail recursive predicates are more challenging to optimise fully.

In Chapter 3, we introduced the function specialisation optimisation technique in Eff. This technique can solve such non tail-recursive continuations issue in Prolog. With this technique, we can generate a new predicate that captures this non tail-recursion. Then, we can use it to tie the recursive knot.

6.2.5 WAM implementation of effect handlers in Prolog

Warren Abstract Machines (WAM) [93, 2] are the de facto standard target for Prolog compilers. Many Prolog systems such as HProlog, BProlog and SWI-Prolog target WAM to compile Prolog code.

Chapter 5 introduced an implementation of effects and handlers in Prolog using a source to source compilation into delimited control code. It has also provided

optimisation techniques to guide the compilation. One has to compile this effect handlers-style written code using our compiler then run it in standard Prolog to get the result which might be inconvenient for many users. Recently, SWI-Prolog added a WAM implementation of delimited control constructs into its library. It would be beneficial for users to also include an implementation for effect handlers in the WAM because of its convenience and efficiency.

Appendix A

Proofs of Eff optimisations

A.1 Soundness of Eff Rewriting rules

Proof of Theorem 1. We prove the theorem for each rewrite rule:

- App-Fun: This follows directly from Equation 3.1.
- Do-Ret: This follows directly from Equation 3.4.
- Do-Op: This follows directly from Equation 3.6.
- With-LetRec:

$$
\begin{aligned}
& \texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ v \\
\equiv\ & (\text{Eq. } 3.3) \\
& \texttt{handle}\ (c_2[(\texttt{fun}\ x \mapsto \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_1)/f])\ \texttt{with}\ v \\
\equiv\ & (f \notin v) \\
& (\texttt{handle}\ c_2\ \texttt{with}\ v)[(\texttt{fun}\ x \mapsto \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_1)/f] \\
\equiv\ & (\text{Eq. } 3.3) \\
& \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ v)
\end{aligned}
$$

- With-Ret: This follows directly from Equation 3.8.

- WITH-HANDLED-OP:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{Op}\, v)\ \texttt{with}\ h \\ \equiv & \text{(Eq. 3.5)} \\ & \texttt{handle}\ (\texttt{do}\ x \leftarrow \texttt{Op}\, v\ ;\ \texttt{return}\ x)\ \texttt{with}\ h \\ \equiv & \text{(Eq. 3.9)} \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto\ \texttt{handle}\ \texttt{return}\ x\ \texttt{with}\ h)/k] \\ \equiv & \text{(Eq. 3.8)} \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto\ c_r)/k] \end{array}$$

- WITH-PURE: We prove this property by induction on c.
  - Case $c = \texttt{return}\ v$:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \\ \equiv & \text{(Eq. 3.8)} \\ & c_r[v/x] \\ \equiv & \text{(Eq. 3.4)} \\ & \texttt{do}\ x \leftarrow \texttt{return}\ v\ ;\ c_r \end{array}$$

  - Case $c = \texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ c'$ with $\texttt{Op} \in \mathcal{O}$:
    This cannot happen since $\Delta \cap \mathcal{O} = \emptyset$.
  - Case $c = \texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ c'$ with $\texttt{Op} \notin \mathcal{O}$:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ c')\ \texttt{with}\ h \\ \equiv & \text{(Eq. 3.10)} \\ & \texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ \texttt{handle}\ c'\ \texttt{with}\ h \\ \equiv & \text{(Induction hypothesis)} \\ & \texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ (\texttt{do}\ x \leftarrow c'\ ;\ c_r) \\ \equiv & \text{(Eq. 3.6)} \\ & \texttt{do}\ x \leftarrow (\texttt{do}\ y \leftarrow \texttt{Op}\, v\ ;\ c')\ ;\ c_r \end{array}$$

- WITH-DO: We prove this property by induction on c_1.
  - Case $c_1 = \texttt{return}\ v$:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{return}\ v\ ;\, c_2)\ \texttt{with}\ h \\ \equiv & \text{(Eq. 3.4)} \\ & \texttt{handle}\ (c_2[v/y])\ \texttt{with}\ h \\ \equiv & (y \notin h) \\ & (\texttt{handle}\ c_2\ \texttt{with}\ h)[v/y] \\ \equiv & \text{(Eq. 3.8)} \\ & \texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h' \end{array}$$

– Case $c_1 = \texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1'$ with $\texttt{Op} \in \mathcal{O}$:

$$
\begin{array}{ll}
 & \texttt{handle}\ (\texttt{do}\ y \leftarrow (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1')\ ;\ c_2)\ \texttt{with}\ h \\
\equiv & \text{(Eq. 3.6)} \\
 & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2))\ \texttt{with}\ h \\
\equiv & \text{(Eq. 3.9)} \\
 & c_{\texttt{Op}}[v/x, (\texttt{fun}\ z \mapsto \texttt{handle}\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2)\ \texttt{with}\ h)/k] \\
\equiv & \text{(Induction hypothesis)} \\
 & c_{\texttt{Op}}[v/x, (\texttt{fun}\ z \mapsto \texttt{handle}\ c_1'\ \texttt{with}\ h')/k] \\
\equiv & \text{(Eq. 3.9)} \\
 & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1')\ \texttt{with}\ h'
\end{array}
$$

– Case $c_1 = \texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1'$ with $\texttt{Op} \notin \mathcal{O}$:

$$
\begin{array}{ll}
 & \texttt{handle}\ (\texttt{do}\ y \leftarrow (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1')\ ;\ c_2)\ \texttt{with}\ h \\
\equiv & \text{(Eq. 3.6)} \\
 & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2))\ \texttt{with}\ h \\
\equiv & \text{(Eq. 3.10)} \\
 & \texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ \texttt{handle}\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2)\ \texttt{with}\ h \\
\equiv & \text{(Induction hypothesis)} \\
 & \texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ \texttt{handle}\ c_1'\ \texttt{with}\ h' \\
\equiv & \text{(Eq. 3.10)} \\
 & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\, v\ ;\ c_1')\ \texttt{with}\ h'
\end{array}
$$

□

A.2 Type Preservation of Basic Compilation

To support the proof, we give the type system for the targeted subset of OCaml in Figure A.1. To simplify the proof, and omit unnecessary details, the type system contains a number of "derived rules" for the OCaml functions used in the elaboration. This way we can also avoid the additional complexity of object-level polymorphism.

Before we prove the main lemma, we prove a lemma about subtyping.

Lemma 11. *For all pure types A and B, and for all computation types $\underline{C}$ and $\underline{D}$ we have that:*

$$A \leqslant B \Rightarrow [\![A]\!] = [\![B]\!]$$

and

$$\underline{C} \leqslant \underline{D} \Rightarrow [\![\underline{C}]\!] = [\![\underline{D}]\!]$$

$$\textsc{O-Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{O-Const}\ \frac{(\mathtt{k} : A) \in \Sigma}{\Gamma \vdash \mathtt{k} : [\![A]\!]} \qquad \textsc{O-Fun}\ \frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash \mathtt{fun}\ x \mapsto E : T_1 \to T_2}$$

$$\textsc{O-App}\ \frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ E_2 : T_2} \qquad \textsc{O-Let}\ \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma, x : T_1 \vdash E_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 : T_2}$$

$$\textsc{O-LetRec}\ \frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash E_1 : T_2 \qquad \Gamma, f : T_1 \to T_2 \vdash E_2 : T_3}{\Gamma \vdash \mathtt{let\ rec}\ f\ x = E_1\ \mathtt{in}\ E_2 : T_3}$$

$$\textsc{IfThenElse}\ \frac{\Gamma \vdash v : \mathtt{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \underline{C}}$$

$$\textsc{O-HandlerCases}\ \frac{\Gamma \vdash E : T_1 \to T_2 \qquad \Big[\Gamma \vdash E_i : [\![A_{\mathtt{Op}}]\!] \to ([\![B_{\mathtt{Op}}]\!] \to T_2) \to T_2\Big]_{(\mathtt{Op}_i : A_i \to B_i) \in \Sigma}}{\Gamma \vdash \{\mathtt{return} = E; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} : (T_1, T_2)\ \mathtt{handler\_cases}}$$

$$\textsc{O-Ret}\ \frac{}{\Gamma \vdash \mathtt{return} : T \to T\ \mathtt{computation}} \qquad \textsc{O-Operation}\ \frac{(\mathtt{Op} : A \to B) \in \Sigma}{\Gamma \vdash \mathtt{op} : [\![A]\!] \to [\![B]\!]\ \mathtt{computation}}$$

$$\textsc{O-Handler}\ \frac{}{\Gamma \vdash \mathtt{handler} : (T_1, T_2)\ \mathtt{handler\_cases} \to (T_1\ \mathtt{computation} \to T_2)}$$

$$\textsc{O-Bind}\ \frac{}{\Gamma \vdash (>>=) : T_1\ \mathtt{computation} \to (T_1 \to T_2\ \mathtt{computation}) \to T_2\ \mathtt{computation}}$$

$$\textsc{O-FMap}\ \frac{}{\Gamma \vdash \mathtt{fmap} : (T_1 \to T_2) \to (T_1\ \mathtt{computation} \to T_2\ \mathtt{computation})}$$

Figure A.1: Typing of (a subset of) OCaml

Proof. The proof proceeds by mutual induction on the derivation of subtyping for pure and dirty types.

Sub-`bool`: $\texttt{bool} \leqslant \texttt{bool}$
In this case the lemma holds trivially.

Sub-`int`: $\texttt{int} \leqslant \texttt{int}$
In this case the lemma holds trivially.

Sub-$\to$: $A \to \underline{C} \leqslant A' \to \underline{C}'$.
From the rule's first hypothesis we have that $A' \leqslant A$. Thus by the induction hypothesis we have that $[\![A']\!] = [\![A]\!]$. From the rule's second hypothesis we have that $\underline{C} \leqslant \underline{C}'$. Thus by the induction hypothesis we have that $[\![\underline{C}]\!] = [\![\underline{C}']\!]$. Hence, we have that $[\![A]\!] \to [\![\underline{C}]\!] = [\![A']\!] \to [\![\underline{C}']\!]$. As $[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!]$ and $[\![A' \to \underline{C}']\!] = [\![A']\!] \to [\![\underline{C}']\!]$, we thus conclude that $[\![A \to \underline{C}]\!] = [\![A' \to \underline{C}']\!]$.

Sub-$\Rightarrow$: $\underline{C} \Rrightarrow \underline{D} \leqslant \underline{C}' \Rrightarrow \underline{D}'$.
From the rule's first hypothesis we have that $\underline{C}' \leqslant \underline{C}$. Thus by the induction hypothesis we have that $[\![\underline{C}']\!] = [\![\underline{C}]\!]$. From the rule's second hypothesis we have that $\underline{D} \leqslant \underline{D}'$. Thus by the induction hypothesis we have that $[\![\underline{D}]\!] = [\![\underline{D}']\!]$. Hence, we have that $[\![\underline{C}]\!] \to [\![\underline{D}]\!] = [\![\underline{C}']\!] \to [\![\underline{D}']\!]$. As $[\![\underline{C} \Rrightarrow \underline{D}]\!] = [\![\underline{C}]\!] \to [\![\underline{D}]\!]$ and $[\![\underline{C}' \Rrightarrow \underline{D}']\!] = [\![\underline{C}']\!] \to [\![\underline{D}']\!]$, we thus conclude that $[\![\underline{C} \Rrightarrow \underline{D}]\!] = [\![\underline{C}' \Rrightarrow \underline{D}']\!]$.

Sub-!: $A \,!\, \Delta \leqslant A' \,!\, \Delta'$.
The rule's first hypothesis is $A \leqslant A'$. Thus by the induction hypothesis we have $[\![A]\!] = [\![A']\!]$. Moreover, we have that $[\![A \,!\, \Delta]\!] = [\![A]\!]$ `computation` and $[\![A' \,!\, \Delta']\!] = [\![A']\!]$ `computation`. Hence, we conclude $[\![A \,!\, \Delta]\!] = [\![A' \,!\, \Delta']\!]$.

□

Now follows the proof of the main theorem.

Proof of Theorem 2. We prove the preservation of typing for the basic elaboration by mutual induction on the typing derivations for values and computations.

(SubVal): $\Gamma \vdash v : A'$
The rule's first assumption is that $\Gamma \vdash v : A$. By the induction hypothesis we thus have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. The rule's second assumption is that $A \leqslant A'$. From Lemma 11 we then have that $[\![A]\!] = [\![A']\!]$. Thus we conclude $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A']\!]$.

(Var): $\Gamma \vdash x : A$

From the hypothesis of the rule, we have that $(x : A) \in \Gamma$. It follows that $(x : [\![A]\!]) \in [\![\Gamma]\!]$. Hence, by rule Var we have $[\![\Gamma]\!] \vdash x : [\![A]\!]$. Because $[\![x]\!] = x$, we conclude $[\![\Gamma]\!] \vdash [\![x]\!] : [\![A]\!]$.

(Const): $\Gamma \vdash \mathtt{k} : A$

We have that $(k : A) \in \Sigma$. Hence, by rule O-Const we have $[\![\Gamma]\!] \vdash k : [\![A]\!]$. Because $[\![k]\!] = k$ we conclude $[\![\Gamma]\!] \vdash [\![k]\!] : [\![A]\!]$.

(Fun): $\Gamma \vdash \mathtt{fun}\ x \mapsto\ c : A \to \underline{C}$

From the rule it follows that $\Gamma, x : A \vdash c : \underline{C}$. By the induction hypothesis, we thus have $[\![\Gamma, x : A]\!] \vdash [\![c]\!] : [\![\underline{C}]\!]$. Because $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$ we thus have from rule O-Fun that $[\![\Gamma]\!] \vdash \mathtt{fun}\ x \mapsto\ [\![c]\!] : [\![A]\!] \to [\![\underline{C}]\!]$. As $[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!]$ and $[\![\mathtt{fun}\ x \mapsto c]\!] = \mathtt{fun}\ x \mapsto\ [\![c]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![\mathtt{fun}\ x \mapsto c]\!] : [\![A \to \underline{C}]\!]$.

(Hand): $\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A\ !\ \Delta \cup \mathcal{O} \Rrightarrow B\ !\ \Delta$

From the first hypothesis of the rule and the induction hypothesis we have that $[\![\Gamma, x : A]\!] \vdash [\![c_r]\!] : [\![B\ !\ \Delta]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash [\![c_r]\!] : [\![B]\!]\ \mathtt{computation}$.

From the second hypothesis and the induction hypothesis we have that $[\![\Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B\ !\ \Delta]\!] \vdash [\![c_{\mathtt{Op}}]\!] : [\![B\ !\ \Delta]\!]$ for each $\mathtt{Op} \in \mathcal{O}$. This simplifies to $[\![\Gamma]\!], x : [\![A_{\mathtt{Op}}]\!], k : [\![B_{\mathtt{Op}}]\!] \to [\![B]\!]\ \mathtt{computation} \vdash [\![c_{\mathtt{Op}}]\!] : [\![B]\!]\ \mathtt{computation}$.

For any $\mathtt{Op} \notin \mathcal{O}$, we have that $[\![\Gamma]\!], x : [\![A_{\mathtt{Op}}]\!], k : [\![B_{\mathtt{Op}}]\!] \to [\![B]\!]\ \mathtt{computation} \vdash (\mathtt{op} x \gg\!= k) : [\![B]\!]\ \mathtt{computation}$ by O-Operation, O-Bind and O-Apply.

Hence, by rule O-Fun and O-HandlerCases, we may conclude that $[\![\Gamma]\!] \vdash \{\mathtt{return} = \mathtt{fun}\ x \mapsto [\![c_r]\!]; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} : ([\![A]\!], [\![B]\!]\ \mathtt{computation})\ \mathtt{handler\_cases}$. Finally, by O-Handler and O-Apply we have that $[\![\Gamma]\!] \vdash \mathtt{handler}\ \{\mathtt{return} = \mathtt{fun}\ x \mapsto [\![c_r]\!]; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} : [\![A]\!]\ \mathtt{computation} \to [\![B]\!]\ \mathtt{computation}$, implying $[\![\Gamma]\!] \vdash [\![\{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\}]\!] : [\![A\ !\ \Delta \cup \mathcal{O} \Rrightarrow B\ !\ \Delta]\!]$.

(SubComp): $\Gamma \vdash c : \underline{C}'$

The proof proceeds analogously to one for rule SubVal.

(App): $\Gamma \vdash e_1\, e_2 : \underline{C}$

From the rule's two hypotheses and the induction hypotheses, we have that $[\![\Gamma]\!] \vdash [\![e_1]\!] : [\![A \to \underline{C}]\!]$ and $[\![\Gamma]\!] \vdash [\![e_2]\!] : [\![A]\!]$. Because $[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!]$ we have by rule O-App that $[\![\Gamma]\!] \vdash [\![e_1]\!]\, [\![e_2]\!] : [\![\underline{C}]\!]$. Because $[\![e_1\, e_2]\!] = [\![e_1]\!]\, [\![e_2]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![e_1\, e_2]\!] : [\![\underline{C}]\!]$.

(LetRec): $\Gamma \vdash \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 : \underline{D}$

From the rule's first hypothesis and the induction hypothesis we have

$[\![\Gamma, f : A \to \underline{C}, x : A]\!] \vdash [\![c_1]\!] : [\![\underline{C}]\!]$. We can simplify this to $[\![\Gamma]\!], f : [\![A]\!] \to [\![\underline{C}]\!], x : [\![A]\!] \vdash [\![c_1]\!] : [\![\underline{C}]\!]$. From the rule's second hypothesis and the induction hypothesis we have $[\![\Gamma, f : A \to \underline{C}]\!] \vdash [\![c_2]\!] : [\![\underline{D}]\!]$. We can simplify this to $[\![\Gamma]\!], f : [\![A]\!] \to [\![\underline{C}]\!] \vdash [\![c_2]\!] : [\![\underline{D}]\!]$. By rule O-LetRec we then have $[\![\Gamma]\!] \vdash \texttt{let rec}\ f\, x = [\![c_1]\!]\ \texttt{in}\ [\![c_2]\!] : [\![\underline{D}]\!]$. Since $[\![\texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_2]\!] = \texttt{let rec}\ f\, x = [\![c_1]\!]\ \texttt{in}\ [\![c_2]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![\texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_2]\!] : [\![\underline{D}]\!]$.

(Ret): $\Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset$

From the rule's assumption and the induction hypothesis we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. Hence, by means of rules O-Ret and O-App we have $[\![\Gamma]\!] \vdash \texttt{return}\ [\![v]\!] : [\![A]\!]\ \texttt{computation}$. As $[\![\texttt{return}\ v]\!] = \texttt{return}\ [\![v]\!]$ and $[\![A\ !\ \emptyset]\!] = [\![A]\!]\ \texttt{computation}$, we conclude $[\![\Gamma]\!] \vdash [\![\texttt{return}\ v]\!] : [\![A\ !\ \emptyset]\!]$.

(Op): $\Gamma \vdash \texttt{Op}\, v : B\ !\ \{\texttt{Op}\}$

From the first hypothesis of the rule, we have $(\texttt{Op} : A \to B) \in \Sigma$. From the second hypothesis of the rule and the induction hypothesis, we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. By rules O-Operation and O-App we then have $[\![\Gamma]\!] \vdash \texttt{op}\, [\![v]\!] : [\![B]\!]\ \texttt{computation}$. As $[\![\texttt{Op}\, v]\!] = \texttt{op}\, [\![v]\!]$ and $[\![B\ !\ \{\texttt{Op}\}]\!] = [\![B]\!]\ \texttt{computation}$, we conclude $[\![\Gamma]\!] \vdash [\![\texttt{op}\, v]\!] : [\![B\ !\ \{\texttt{Op}\}]\!]$.

(Do): $\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ;\ c_2 : B\ !\ \Delta$

From the rule's first hypothesis and the induction hypothesis we have $[\![\Gamma]\!] \vdash [\![c_1]\!] : [\![A\ !\ \Delta]\!]$. We can simplify this to $[\![\Gamma]\!] \vdash [\![c_1]\!] : [\![A]\!]\ \texttt{computation}$. From the rule's second hypothesis and the induction hypothesis we have $[\![\Gamma, x : A]\!] \vdash [\![c_2]\!] : [\![B\ !\ \Delta]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash [\![c_2]\!] : [\![B]\!]\ \texttt{computation}$. Using rule O-Fun we have $[\![\Gamma]\!] \vdash \texttt{fun}\ x \mapsto [\![c_2]\!] : [\![A]\!] \to [\![B]\!]\ \texttt{computation}$. Using rules O-Bind and O-App we then have $[\![\Gamma]\!] \vdash [\![c_1]\!] \texttt{>>=} (\texttt{fun}\ x \mapsto [\![c_2]\!]) : [\![B]\!]\ \texttt{computation}$. As $[\![\texttt{do}\ x \leftarrow c_1\ ;\ c_2]\!] = [\![c_1]\!] \texttt{>>=} (\texttt{fun}\ x \mapsto [\![c_2]\!])$, we conclude $[\![\Gamma]\!] \vdash [\![\texttt{do}\ x \leftarrow c_1\ ;\ c_2]\!] : [\![B]\!]\ \texttt{computation}$.

(With): $\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D}$

From the rule's first assumption and the induction hyptothesis we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![\underline{C} \Rrightarrow \underline{D}]\!]$. Because $[\![\underline{C} \Rrightarrow \underline{D}]\!] = [\![\underline{C}]\!] \to [\![\underline{D}]\!]$ we thus have $[\![\Gamma]\!] \vdash [\![v]\!] : [\![\underline{C}]\!] \to [\![\underline{D}]\!]$. From the rule's second assumption and the induction hypothesis we also have $[\![\Gamma]\!] \vdash [\![c]\!] : [\![\underline{C}]\!]$. By rule O-App we then have $[\![\Gamma]\!] \vdash [\![v]\!]\, [\![c]\!] : [\![\underline{D}]\!]$. Because $[\![\texttt{handle}\ c\ \texttt{with}\ v]\!] = [\![v]\!]\, [\![c]\!]$ we thus conclude $[\![\Gamma]\!] \vdash [\![\texttt{handle}\ c\ \texttt{with}\ v]\!] : [\![\underline{D}]\!]$.

□

Appendix B

Proofs and Detailed examples for Prolog

B.1 Detailed parital evaluation example

This appendix elaborates the optimisation example of Section 5.2.3 in more depth.

We start from the following program:

```
:- effect c/1.
ab.
ab :- c(a), c(b), ab.
query(Lin) :-
  handle ab with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=[]).
```

Step 1 We abstract the goal `handle ab with ...` into a new predicate `ab0/2`. This new predicate takes two arguments: one for every parameter in the handler's `for` clause. The original call is replaced by a call to the new predicate, supplying the actual parameters of the handler as actual arguments.

```
query(Lin) :- ab0(Lin,[]).
```

The predicate ab0/2 is a copy of ab/0's definition, with the handler wrapped around each clause's body.

```
ab0(Lin,Lout) :-
  handle true with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).

ab0(Lin,Lout) :-
  handle (c(a), c(b), ab) with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).
```

Step 2 The optimiser now applies rewrite rules to the two clauses. In the first clause, Rule (O-DROP) can be applied because the effect system provides the information that the goal `true` has no effects. Hence, we drop the operation clause:

```
ab0(Lin,Lout) :-
  handle true with
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).
```

Step 3 The handler currently handles no operations[1]. The optimiser proceeds with applying (O-TRIV):

```
ab0(Lin,Lout) :-
  true,
  Lin1 = Lout1 ,
  Lin1 = Lin,
  Lout1 = Lout.
```

Step 4 By partially evaluating `true` and the remaining unifications, the first clause is simplified to:

```
ab0(L,L).
```

[1] This syntax is only allowed during the compilation process.

Step 5 In the second clause the handler's goal starts with the c/1 operation. The optimiser applies (O-OP) to the handler, producing the following code:

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
  Lin1 = Lin,
  Lout1 = Lout,
  handle (c(b), ab) with
    (c(X1) -> Lin11=[X1|Lmid1], continue(Lmid1,Lout11))
  finally (Lin11 = Lout11)
  for (Lin11=Lmid,Lout11=Lout1).
```

All the variables in the new handler goal are fresh variables. Observe that the actual arguments in the newly generated `for` clause are taken from the `continue` call of the previous handler. This is to ensure the correct state threading of the handler, and to keep the correct semantics of the program.

Step 6 The optimiser re-applies (O-OP) for c(b), generating the following code:

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
  Lin1 = Lin,
  Lout1 = Lout,
  Lin11 = [b|Lmid1],
  Lin11 = Lmid,
  Lout11 = Lout1,
  handle (ab) with
    (c(X2) -> Lin12=[X1|Lmid2], continue(Lmid2,Lout12))
  finally (Lin12 = Lout12)
  for (Lin12=Lmid1,Lout12=Lout11).
```

Step 7 The remaining handler goal is now a variant of the original one, which was already abstracted into ab0/2. Therefore, we can replace it with ab0/2.

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
  Lin1 = Lin,
  Lout1 = Lout,
  Lin11 = [b|Lmid1],
  Lin11 = Lmid,
```

```
    Lout11 = Lout1,
    ab0(Lmid1,Lout11).
```

Step 8 The clause now consists of several unifications followed by a tail-recursive call. Partially evaluating the unifications leads to the final optimised code:

```
ab0([a,b|Lmid1],Lout) :-
  ab0(Lmid1,Lout).
```

B.2 State-DCG handler example in focus

This appendix shows the result of optimising a program that consists of two handlers. We first show the elaboration into delimited control. Then, we show how the original program can be optimised by means of the rewrite rules and partial evaluation.

We use the following program, which was used to generate the results of the first benchmarks in Figure 5.6. As described in Section 5.3, there are two handlers: one handles the implicit state operations and the other handles the DCG operations.

```
abinc.
abinc :- c(a), c(b), get_state(St), St1 is St+1,
         put_state(St1), abinc.

state_phrase_handler(Sin,Sout,Lin,Lout) :-
  handle
      (handle abinc
        with
          ( get_state(Q)  -> Q = Sin1, continue(Sin1,Sout1)
          ; put_state(NS) -> continue(NS,Sout1)
          )
        finally
          Sout1 = Sin1
        for
          (Sin1 = Sin, Sout1 = Sout)
      )
    with
      (c(X) -> Lin1 = [X|Lmid], continue(Lmid,Lout1))
    finally
      Lin1 = Lout1
```

```
    for
      (Lin1=Lin, Lout1=Lout).
```

The inner handler's goal is `abinc`, which consumes two elements, `a` and `b`, by using the operation `c/1` and then increments the state using the operations `get_state/1` and `put_state/1`.

```
?- state_phase_handler(0,Sout,[a,b,a,b,a,b],Lout).
Sout = 0
Lout = [a,b,a,b,a,b];
Sout = 1
Lout = [a,b,a,b];
Sout = 2
Lout = [a,b];
Sout = 3
Lout = [].
```

The immediate elaboration into delimited control yields:

```
state_phrase_handler(A, B, C, D) :-
  handler_0(handler_1(abinc,A,B), C, D).
handler_1(A, B, C) :-
  reset(A, D, E),
  ( D == 0 ->
      C = B
  ; E = get_state(F) ->
      F = B,
      handler_1(D, B, C)
  ; E = put_state(G) ->
      handler_1(D, G, C)
  ;   shift(E),
      handler_1(D, B, C)
  ).
handler_0(A, B, C) :-
  reset(A, D, E),
  ( D == 0 ->
      B = C
  ; E = c(F) ->
      B = [F|G],
      handler_0(D, G, C)
  ;   shift(E),
```

```
      handler_0(D, B, C)
  ).
```

The predicates `handler_0/3` and `handler_1/3` correspond to the elaborated DCG and state handlers respectively. They follow the semantics described in Section 5.1.4.

Using the rewrite rules first, yields the following elaborated program instead:

```
state_phrase_handler(A, B, C, D) :-
  handler_2(abinc, A, B, C, D).
handler_2(A, B, C, D, E) :-
  reset(A, F, G),
  ( F == 0 ->
      C = B,
      D = E
  ; G = get_state(H) ->
      H = B,
      handler_2(F, B, C, D, E)
  ; G = put_state(I) ->
      handler_2(F, I, C, D, E)
  ; G = c(J) ->
      D = [J|K],
      handler_2(F, B, C, K, E)
  ;   shift(G),
      handler_2(F, B, C, D, E)
  ).
```

The two handlers have been merged into one, with the corresponding performance improvement.

When partial evaluation is enabled as well, the optimisation goes one step further and yields the following final program:

```
state_phrase_handler(A, B, C, D) :-
  abinc0(A, B, C, D).
abinc0(A, A, B, B).
abinc0(A, B, [a,b|C], D) :-
  E is A+1,
  abinc0(E, B, C, D).
```

Partial evluation has pushed the handlers into the definition of `abcinc/0` where the rewrite rules have been able to replace the operations by the corresponding

```
eval(G) :- eval(G,Signal),
           ( Signal = shift(Term,Cont) ->
             fail
           ; true).

eval(shift(Term),Signal) :- !,Signal = shift(Term,true).

eval(reset(G,Cont,Term),Signal) :- !, eval(G,Signal1),
                                      ( Signal1 = ok -> Cont = 0, Term = 0
                                      ; Signal1 = shift(Term,Cont)),
                                      Signal = ok.

eval((G1,G2),Signal) :- !, eval(G1,Signal1),
                        ( Signal1 = ok -> eval(G2,Signal)
                        ; Signal1 = shift(Term,Cont),
                          Signal = shift(Term,(Cont,G2))).

eval((G1;G2),Signal) :- !, ( eval(G1,Signal)
                           ; eval(G2,Signal)).

eval((C->G1;G2),Signal) :- !, ( eval(C,Signal1) ->
                                ( Signal1 = ok -> eval(G1,Signal)
                                ; fail
                                )
                              ; eval(G2,Signal)).

eval(Goal,Signal) :- built_in_predicate(Goal), !, call(Goal), Signal = ok.

eval(Goal,Signal) :- clause(Goal,Body), eval(Body,Signal).
```

Figure B.1: Delimited Control Meta-Interpreter

handler clauses. As a consequence, the handlers are eliminated and no delimited control primitives are generated.

B.3 Soundness of Rule (O-Disj)

This appendix proves the soundness of the (O-DISJ) rewrite rule. Our proof relies on the elaboration of the handler syntax into delimited control and the corresponding semantics for delimited control given by Schrijvers et al. [80]. This semantics is expressed in terms of a Prolog meta-interpreter that we show in Figure B.1.

We start from the left-hand side of the rewrite rule and turn it into the right-hand side by means of a number of equivalence preserving transformations.

$$\begin{array}{c}\texttt{handle (G1;G2) with}\\ \overline{op \rightarrow G;}\\ \texttt{finally } G_f\\ \texttt{for } G_s.\end{array} \tag{B.1}$$

The elaboration of this handler goal into delimited control yields the following auxiliary predicate:

```
h(Goal,P1,..,Pn) :-
  reset(Goal,Cont,Term),
  ( Term == 0  -> Gf
  ; Term =  op  ->  G
  ; shift(Signal), h(Cont,P1,...,Pn)
  ).
```

Here the variables P_i are the formal parameters of G_s. The goal itself is then by definition equivalent to

$$\texttt{h((G1;G2),A}_1\texttt{,...,A}_n\texttt{)} \tag{B.2}$$

where the $\texttt{A}_i$ are the actual parameters of G_s.

This is equivalent to evaluation the goal in the meta-interpreter:

$$\texttt{eval(h((G1;G2),A}_1\texttt{,...,A}_n\texttt{))} \tag{B.3}$$

We can now unfold the `eval/1` call and subsequently unfold the resulting call to the auxiliary predicate `eval/2` which selects the last clause. After also evaluating the call to `clause/2` to unfold $\texttt{h}/n+1$ we get:

```
eval( ( reset((G1;G2),Cont,Term),
        ( Term == 0  -> Gf
        ; Term =  op  ->  G
        ; shift(Signal), h(Cont,P1,...,Pn)
        )
      )
    , Signal
    ),
(Signal = shift(Term,Cont) -> fail ; true)
```
(B.4)

For the sake of space, we refer to the if-then-else block after the `reset/3` call as `<Switches>`. We can thus abbreviate the above as:

```
eval( (reset((G1;G2),Cont,Term), <Switches>)
    , Signal
    ),                                                    (B.5)
(Signal = shift(Term,Cont) -> fail ; true)
```

Unfolding `eval/2` using the appropriate clause for conjunction, yields:

```
eval(reset((G1;G2),Cont,Term), Signal1),
( Signal1 = ok -> eval(<Switches>, Signal)
; Signal1 = shift(Term,Cont) ->                           (B.6)
            Signal = shift(Term,(Cont,<Switches>))
),
(Signal = shift(Term,Cont) -> fail ; true)
```

Now we unfold the first call to `eval/2` using the clause for `reset/3`:

```
eval((G1;G2), Signal2),
( Signal2 = ok -> Cont = 0, Term = 0
; Signal2 = shift(Term,Cont)
),
Signal1 = ok,
( Signal1 = ok -> eval(<Switches>, Signal)                (B.7)
; Signal1 = shift(Term,Cont) ->
            Signal = shift(Term,(Cont,<Switches>))
),
(Signal = shift(Term,Cont) -> fail ; true)
```

Again, we unfold the first call to `eval/2` using the clause for disjunction:

```
( eval(G1, Signal2) ; eval(G2, Signal2) ),
( Signal2 = ok -> Cont = 0, Term = 0
; Signal2 = shift(Term,Cont)
),
Signal1 = ok,
( Signal1 = ok -> eval(<Switches>, Signal)                (B.8)
; Signal1 = shift(Term,Cont) ->
            Signal = shift(Term,(Cont,<Switches>))
),
(Signal = shift(Term,Cont) -> fail ; true)
```

We now distribute what comes after the first disjunction into both branches.

```
(
  eval(G1, Signal2),
  ( Signal2 = ok -> Cont = 0, Term = 0
  ; Signal2 = shift(Term,Cont)
  ),
  Signal1 = ok,
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) ->
              Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
;                                                          (B.9)
  eval(G2, Signal2),
  ( Signal2 = ok -> Cont = 0, Term = 0
  ; Signal2 = shift(Term,Cont)
  ),
  Signal1 = ok,
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) ->
              Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

At this point we change gear and start folding again. First we fold the `reset/2` clause of `eval/2` twice, once in each branch.

```
(
  eval(reset(G1,Term,Cont),Signal1),
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) ->
              Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
;
  eval(reset(G2,Term,Cont),Signal1),
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) ->
              Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

(B.10)

Then we fold the conjunction clause of `eval/2` in each branch.

```
(
  eval((reset(G1,Term,Cont),<Switches>),Signal),
  (Signal = shift(Term,Cont) -> fail ; true)
;
  eval((reset(G2,Term,Cont),<Switches>),Signal),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

(B.11)

Subsequently, we fold `eval/1` twice.

```
(
  eval((reset(G1,Term,Cont),<Switches>))
;
  eval((reset(G2,Term,Cont),<Switches>))
)
```

(B.12)

Now we can drop the meta-interpretation layer again.

```
(
  (reset(G1,Term,Cont),<Switches>)
;
  (reset(G2,Term,Cont),<Switches>)
)
```

(B.13)

Then we fold $\texttt{h}/n+1$ twice.

$$\texttt{(h(G1,A}_1\texttt{,...,A}_n\texttt{); h(G2,A}_1\texttt{,...,A}_n\texttt{))} \tag{B.14}$$

Finally, we invert the elaboration to obtain the right-hand side of the rewrite rule.

$$\begin{array}{c}\texttt{handle } G_1 \texttt{ with}\\ \overline{op \to G;}\\ \texttt{finally}(G_f)\\ \texttt{for}(G_s)\end{array} \quad ; \quad \begin{array}{c}\texttt{handle } G_2 \texttt{ with}\\ \overline{op \to G;}\\ \texttt{finally}(G_f)\\ \texttt{for}(G_s)\end{array} \tag{B.15}$$

List of Symbols

The next list describes several symbols that are used in the thesis.

Chapter 2

$\underline{C}$	Dirty type
Γ	Typing environment
Δ	Dirt
!	Dirty type constructor
$\Downarrow$	Big-step semantics evaluate
$\Rightarrow$	Handler type
$\in$	Belongs to a set
λ	Lambda abstraction
$\mathcal{T}$	Derivation tree
$\mathcal{O}$	Set of operations
$\rightarrow$	Function type
$A \leqslant A'$	Subtyping relation
A	Simple type
c	Eff computation
v	Eff value
x : *int*	Variable x has type int
*	Tuple constructor in OCaml
()	Unit value
<>	Inequality check in OCaml
#	Effect call
&&	And operator in OCaml
k	Continuation variable

Chapter 3

$[\![C]\!]$	Compilation of C from Effy to OCaml
$\equiv$	Basic equivalence
$\forall.x$	For all values given to the variable x
$\rightsquigarrow$	Term-rewriting rule
$\notin$	Does not belong to a set
fmap	Higher order function in OCaml
T	Type of OCaml
[]	Substitution operator
>>=	Sequencing operator in OCaml
CPS	Continuation Passing Style

Chapter 4

$\bullet$	Empty substitution
$\triangleright$	ExEff cast
C^{c}	Computation-holed computation
H^{c}	Computation-holed handler
γ	Coercions of ExEff
$\mathcal{Q}$	Constraints set
$\gamma_1 \gg \gamma_2$	Transitivity of coercions in ExEff
ω	Coercion variable of ExEff
$\mathcal{P}$	Constraints queue
V^{c}	Computation-holed value
δ	Dirt variable
ϵ	Erasure function symbol
$\equiv_{\mathrm{c}}^{\rightsquigarrow}$	Congruence step for SkelEff computations
$\equiv_{\mathrm{v}}^{\rightsquigarrow}$	Congruence step for SkelEff values
τ	Skeleton
ς	Skeleton variable
$\rightsquigarrow_{\mathrm{c}}$	small-step semantics for ExEff and SkelEff computations
$\rightsquigarrow_{\mathrm{v}}$	small-step semantics for ExEff and SkelEff values
S	Polytype
K	Qualified type
$\langle T \rangle$	Reflexivity of coercions in ExEff
σ	Set of substitutions
ρ	Constraint
π	Simple constraint
α	Type variable
C^{v}	Value-holed computation
$\vdash_{co}$	Constraint Entailment for ImpEff
$\vdash_{c}$	Computation typing for ImpEff
$\vdash_{v}$	Value typing for ImpEff
$\vdash_{\Delta}$	Well-formedness for dirts
$\vdash_{\tau}$	Well-formedness for skeletons
$\vdash_{\mathtt{co}}$	Coercion typing for ExEff
$\vdash_{\mathtt{cty}}$	Well-formedness for computation types of ImpEff
$\vdash_{\mathtt{ct}}$	Well-formedness for constraints
$\vdash_{\mathtt{ec}}$	SkelEff computation typing
$\vdash_{\mathtt{ev}}$	SkelEff value typing
$\vdash_{\mathtt{vty}}$	Well-formedness for value types of ImpEff
H^{v}	Value-holed handler
T	Value type for ExEff
V^{v}	Value-holed value

Chapter 5

$op_n \to G_n$	operation clause for op_n is G_n
;	Prolog disjunctive operator
G	Prolog goal
op /n	Operation op has n arguments

Bibliography

[1] Agesen, O. Constraint-based type inference and parametric polymorphism. In *International Static Analysis Symposium* (1994), Springer, pp. 78–100.

[2] Ait-Kaci, H. *Warren's abstract machine: a tutorial reconstruction*. MIT press, 1991.

[3] Awodey, S. Category theory, volume 49 of oxford logic guides, 2006.

[4] Barendregt, H. *The Lambda Calculus: its Syntax and Semantics, volume 103 of Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.

[5] Bauer, A., and Pretnar, M. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science 10*, 4 (2014).

[6] Bauer, A., and Pretnar, M. Programming with algebraic effects and handlers. *Journal of Logic and Algebraic Programming 84*, 1 (2015), 108–123.

[7] Biernacki, D., Piróg, M., Polesiuk, P., and Sieczkowski, F. Abstracting algebraic effects.

[8] Brachthäuser, J. I., and Schuster, P. Effekt: extensible algebraic effects in scala (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017* (2017), H. Miller, P. Haller, and O. Lhoták, Eds., ACM, pp. 67–72.

[9] Brachthäuser, J. I., Schuster, P., and Ostermann, K. Effect handlers for the masses. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018), 111.

[10] Brady, E. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (2013), ACM, pp. 133–144.

[11] Breazu-Tannen, V., Coquand, T., Gunter, C. A., and Scedrov, A. Inheritance as implicit coercion. *Information and Computation vol 93* (1991), 172–221.

[12] Bruggeman, C., Waddell, O., and Dybvig, R. K. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 99–107.

[13] Cardelli, L. *An implementation of F*. Digital. Systems Research Center, 1993.

[14] Clocksin, W. F., and Mellish, C. S. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012.

[15] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (2006), vol. 4709 of *Lecture Notes in Computer Science*, Springer, pp. 266–296.

[16] Crary, K. Typed compilation of inclusive subtyping. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (NY, USA, 2000), ICFP '00, ACM, pp. 68–81.

[17] Damas, L., and Milner, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1982), ACM, pp. 207–212.

[18] Damas, L., and Milner, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (NY, USA, 1982), POPL '82, ACM, pp. 207–212.

[19] Danvy, O., and Filinski, A. Abstracting control. In *LISP and Functional Programming* (1990), pp. 151–160.

[20] Demoen, B. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Comp. Sc., KU Leuven, Belgium, 2002.

[21] Desouter, B., van Dooren, M., and Schrijvers, T. Tabling as a library with delimited control. *TPLP 15*, 4-5 (2015), 419–433.

[22] Dijkstra, E. W. Letters to the editor: Go to statement considered harmful. *Commun. ACM 11*, 3 (Mar. 1968), 147–148.

[23] Dolan, S. *Algebraic Subtyping*. PhD thesis, PhD thesis, University of Cambridge, 2016, 2016.

[24] Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K., and White, L. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming* (2017), Springer, pp. 98–117.

[25] Dolan, S., and Mycroft, A. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (2017), ACM, pp. 60–72.

[26] Dolan, S., White, L., and Madhavapeddy, A. Multicore ocaml. In *OCaml Workshop* (2014), vol. 2.

[27] Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., and Madhavapeddy, A. Effective concurrency through algebraic effects. In *OCaml Users and Developers Workshop* (2015).

[28] Faes, A. Algebraic subtyping for algebraic effects and handlers. Master's thesis, KULeuven, 2018.

[29] Felleisen, M. The theory and practice of first-class prompts. POPL '88, pp. 180–190.

[30] Gaster, B. R., and Jones, M. P. A polymorphic type system for extensible records and variants.

[31] Gill, A., Launchbury, J., and Peyton Jones, S. L. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture* (1993), ACM, pp. 223–232.

[32] Girard, J.-Y., Taylor, P., and Lafont, Y. *Proofs and Types*. Cambridge University Press, 1989.

[33] Hillerström, D., and Lindley, S. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (2016), ACM, pp. 15–27.

[34] Hillerström, D., and Lindley, S. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems* (2018), Springer, pp. 415–435.

[35] Hillerström, D., Lindley, S., and Sivaramakrishnan, K. Compiling links effect handlers to the ocaml backend. In *OCaml Workshop* (2016).

[36] Hindley, R. The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society 146* (1969), 29–60.

[37] Holzbaur, C. Metastructures versus attributed variables in the context of extensible unification. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings* (1992), M. Bruynooghe and M. Wirsing, Eds., vol. 631 of *Lecture Notes in Computer Science*, Springer, pp. 260–268.

[38] Ivanovic, D., Morales Caballero, J. F., Carro, M., and Hermenegildo, M. Towards structured state threading in Prolog. In *CICLOPS 2009* (2009).

[39] Jones, M. P. A theory of qualified types. In *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings* (1992), B. Krieg-Brückner, Ed., vol. 582 of *Lecture Notes in Computer Science*, Springer, pp. 287–306.

[40] Kahn, G. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science* (1987), Springer, pp. 22–39.

[41] Kammar, O., Lindley, S., and Oury, N. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming* (2013), ICFP '14, ACM, pp. 145–158.

[42] Kammar, O., and Plotkin, G. D. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012), ACM, pp. 349–360.

[43] Kammar, O., and Pretnar, M. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming 27* (2017).

[44] Kiselyov, O. Delimited control in ocaml, abstractly and concretely: System description. In *International Symposium on Functional and Logic Programming* (2010), Springer, pp. 304–320.

[45] Kiselyov, O., and Ishii, H. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015* (2015), B. Lippmeier, Ed., ACM, pp. 94–105.

[46] Kiselyov, O., Sabry, A., and Swords, C. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013* (2013), C. Shan, Ed., pp. 59–70.

[47] Kiselyov, O., and Sivaramakrishnan, K. Eff directly in ocaml. In *OCaml Workshop* (2016).

[48] Le Houitouze, S. A New Data Structure for Implementing Extensions to Prolog. vol. 456 of *LNCS*, pp. 136–150.

[49] Leijen, D. Koka: Programming with row polymorphic effect types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.* (2014), P. Levy and N. Krishnaswami, Eds., vol. 153 of *EPTCS*, pp. 100–126.

[50] Leijen, D. Implementing algebraic effects in C - "monads for free in c". In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings* (2017), B. E. Chang, Ed., vol. 10695 of *Lecture Notes in Computer Science*, Springer, pp. 339–363.

[51] Leijen, D. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 486–499.

[52] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique 54* (2014).

[53] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), ACM, pp. 333–343.

[54] Lindley, S. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming* (2014), ACM, pp. 47–58.

[55] Lindley, S., McBride, C., and McLaughlin, C. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 500–514.

[56] Mitchell, J. C. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1984), POPL '84, ACM, pp. 175–185.

[57] Mycroft, A., and O'Keefe, R. A. A polymorphic type system for prolog. *Artificial intelligence 23*, 3 (1984), 295–307.

[58] Neumerkel, U. Extensible unification by metastructures. In *META'90* (Apr. 1990), pp. 352–364.

[59] Nielson, H. R., and Nielson, F. *Semantics with applications*, vol. 104. Springer, 1992.

[60] Pereira, F. C., and Warren, D. H. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence 13*, 3 (1980), 231–278.

[61] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. Simple unification-based type inference for gadts. In *ICFP '06* (2006).

[62] Pierce, B. C. *Types and programming languages*. MIT press, 2002.

[63] Pierce, B. C. *Advanced topics in types and programming languages*. MIT press, 2005.

[64] Pierce, B. C., and Turner, D. N. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS) 22*, 1 (2000), 1–44.

[65] Plaisted, D. A. Equational reasoning and term rewriting systems. *Handbook of logic in artificial intelligence and logic programming 1* (1993), 273–364.

[66] Plotkin, G., and Pretnar, M. Handlers of algebraic effects. In *European Symposium on Programming* (2009), Springer, pp. 80–94.

[67] Plotkin, G. D. A structural approach to operational semantics. *J. Log. Algebr. Program. 60-61* (2004), 17–139.

[68] Plotkin, G. D., and Pretnar, M. A logic for algebraic effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA* (2008), IEEE Computer Society, pp. 118–129.

[69] Pottier, F. A versatile constraint-based type inference system. *Nordic Journal of Computing 7*, 4 (2000), 312–347.

[70] Pottier, F. Simplifying subtyping constraints: A theory. *Information and Computation 170*, 2 (2001), 153–183.

[71] Pretnar, M. Inferring algebraic effects. *Logical Methods in Computer Science 10*, 3 (2014).

[72] Pretnar, M. An introduction to algebraic effects and handlers, invited tutorial. *Electronic Notes in Theoretical Computer Science 319* (2015), 19–35.

[73] Pretnar, M., and Plotkin, G. D. Handling algebraic effects. *Logical Methods in Computer Science 9* (2013).

[74] Pretnar, M., Saleh, A. H., Faes, A., and Schrijvers, T. Efficient compilation of algebraic effects and handlers. Tech. Rep. CW 708, KU Leuven Department of Computer Science, 2017.

[75] Roy, P. V. A useful extension to prolog's definite clause grammar notation. 132–134.

[76] Saleh, A. H. Transforming delimited control: Achieving faster effect handlers. *ICLP (Technical Communications) 1433* (2015).

[77] Saleh, A. H., Karachalias, G., Pretnar, M., and Schrijvers, T. Explicit effect subtyping. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP* (2018), Springer, pp. 327–354.

[78] Saleh, A. H., and Schrijvers, T. Efficient algebraic effect handlers for prolog. *Theory and Practice of Logic Programming 16*, 5-6 (2016), 884–898.

[79] Schimpf, J. Logical loops. In *International Conference on Logic Programming* (2002), Springer, pp. 224–238.

[80] Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. Delimited continuations for Prolog. *TPLP 13*, 4-5 (2013), 533–546.

[81] Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. Delimited continuations for Prolog. *Theory and Practice of Logic Programming 13*, 4-5 (2013), 533–546.

[82] Schrijvers, T., Demoen, B., Triska, M., and Desouter, B. Tor: Modular search with hookable disjunction. *Sci. Comput. Program. 84* (2014), 101–120.

[83] Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. Type checking with open type functions. In *ICFP '08* (2008), ACM, pp. 51–62.

[84] Schrijvers, T., Wu, N., Desouter, B., and Demoen, B. Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* (2014), ACM, pp. 259–270.

[85] Sekiyama, T., and Igarashi, A. Handling polymorphic algebraic effects. *arXiv preprint arXiv:1811.07332* (2018).

[86] Simonet, V. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings* (2003), A. Ohori, Ed., Springer, pp. 283–302.

[87] Spivey, J. M., and Seres, S. Embedding prolog in haskell. In *Proceedings of Haskell* (1999), vol. 99, pp. 1999–28.

[88] Sulzmann, M., Chakravarty, M. M. T., Peyton Jones, S., and Donnelly, K. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New York, NY, USA, 2007), TLDI '07, ACM, pp. 53–66.

[89] Swierstra, W. Data types à la carte. *Journal of functional programming 18*, 4 (2008), 423–436.

[90] Tarau, P., Dahl, V., and Fall, A. Backtrackable state with linear assumptions, continuations and hidden accumulator grammars. In *ILPS* (1995), Citeseer, p. 642.

[91] van der Ploeg, A., and Kiselyov, O. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014* (2014), W. Swierstra, Ed., ACM, pp. 133–144.

[92] Wang, T., and Smith, S. F. Precise constraint-based type inference for java. In *European Conference on Object-Oriented Programming* (2001), Springer, pp. 99–117.

[93] Warren, D. H. An abstract prolog instruction set. *Technical note 309* (1983).

[94] Wile, D. S. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering* (1997), ACM, pp. 472–480.

[95] Wu, N., and Schrijvers, T. Fusion for free - efficient algebraic effect handlers. In *Mathematics of Program Construction* (2015), vol. 9129 of *LNCS*, Springer, pp. 302–322.

[96] Wu, N., Schrijvers, T., and Hinze, R. Effect handlers in scope. *ACM SIGPLAN Notices 49*, 12 (2015), 1–12.

List of publications

Workshops papers

- Saleh, Amr Hany. "Transforming Delimited Control: Achieving Faster Effect Handlers." In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*

Technical reports

- Pretnar, Matija, Amr Hany Saleh, Axel Faes, and Tom Schrijvers. "Efficient compilation of algebraic effects and handlers." *Technical Report CW 708, KU Leuven Department of Computer Science*, 2017.

Papers at international conferences

- Saleh, Amr Hany, and Tom Schrijvers. "Efficient algebraic effect handlers for Prolog." *Theory and Practice of Logic Programming* 16.5-6 (2016): 884-898.
- Saleh, Amr Hany, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. "Explicit Effect Subtyping." In *European Symposium on Programming*, pp. 327-354. Springer, Cham, 2018.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
ah.saleh@cs.kuleuven.be
https://dtai.cs.kuleuven.be