Žiga Lukšič

APPLICATIONS OF ALGEBRAIC EFFECT THEORIES

Doctoral thesis

Advisor: doc. dr. Matija Pretnar

Ljubljana, 2020
Žiga Lukšič

UPORABA TEORIJ ALGEBRAJSKIH UČINKOV

Doktorska disertacija

Mentor: doc. dr. Matija Pretnar

Ljubljana, 2020
Acknowledgments

Firstly, I would like to thank my advisor Matija Pretnar for all his persistent help and guidance. Thank you for your endless patience and understanding. You taught me so much about math, programming, teaching, and life in general. I enjoyed working on every project we shared, and I hope there are more in the future. Working with you has been an honour. Thank you for everything.

I also want to thank the members of my thesis committee, Jeremy Gibbons, Marko Petkovšek, and Alex Simpson, for their thorough reading of the thesis. Your suggestions were very helpful for improving this thesis.

I would like to thank all of my work colleagues—you were by far the best part of this journey. I always had the chance to consult Andrej or Alex whenever I encountered difficulties in my research. I was only able to get this far with your help. The support from Danel, Filip, and Ohad was never-ending and amazingly helpful. Thank you for all your advice, ideas, and stories. Work was a lot more fun when Brett, Niels, Philipp, and Gavin were around. Your jokes were often the highlight of my day. I am especially thankful to Anja for being an absolute joy to share an office with. I cannot imagine a better office-mate. I also want to thank Jure, Vesna, and Tatiana for the weekly lunch meetings. I always have an amazing time when I am in your company. I also wish to extend my thanks to all the personnel at FMF, especially Jerneja and Silva. Your hard work is incredibly appreciated.

I want to thank my family—in particular my parents and my brother, who were always there for me. You sparked my interest in science and continue to encourage me at every step. You made sure I was able to focus on the things I wanted to do. I am grateful to have you in my life.

And lastly, I wish to thank all of my friends from the bottom of my heart. You are far too many to name, but each and every one of you plays an important role in my life. Thank you for keeping me sane when I was overwhelmed. Thank you for urging me on when life got difficult. And thank you for all the good times we shared.

This thesis is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.
Abstract

Algebraic effects are an established method of implementing effectful behaviour in functional programming languages. Computational effects are represented by operations and implemented through effect handlers. An effect theory consists of a type signature and a set of equations describing the behaviour of effect invocations. All effect handlers are required to adhere to the prescribed effect theory, meaning that they do not differentiate between two programs considered equal in the given theory. The standard approach to algebraic effects assumes a global effect theory, so all handlers need to respect the same set of equations. This often becomes very restricting in terms of suitable handlers and therefore most contemporary work focuses on theories that contain no equations. Discarding equations allows for a wider variety of viable handlers but drastically reduces the capabilities to reason about properties of effectful code.

In the thesis we present the language \textit{EEFF} that relaxes the single theory limitation by using local effect theories, allowing the use of different theories in different parts of the program, even when pertaining to effects with the same signature. This alleviates the issues of global effect theories while providing all benefits of equations. The type system is upgraded to track theory information, allowing for safe use of handlers and ensuring their correctness at the relevant theory. Proofs of handler correctness are done in a logic that is coupled with the type system. The type system can be coupled with different logics, granting the option to select a logic suitable for the problems at hand. The soundness of a logic is established with respect to a denotational semantics based on partial equivalence relations.

The safety theorems of \textit{EEFF} are formalised in the proof assistant Coq, and the implementation of \textit{EEFF} is an extension of the language Eff. The formalisation also doubles as a reasoning tool for programs with algebraic effect theories and features two different logics to choose from, both of which are shown to be sound. Multiple examples throughout the thesis showcase the benefits of local algebraic theories.

\textbf{Math. Subj. Class. (2010):} 68N15, 68N18, 03B35, 03B70

\textbf{Keywords:} Algebraic effects, effect handlers, functional programming, theory of programming languages, denotational semantics
Izvleček

Algebrajski učinki so uveljavljena metoda za modeliranje računskih učinkov v funkcijskem programiranju. Učinke predstavimo z operacijami, pomen pa jim dodelimo s prestrezniki. Teorije algebrajskih učinkov so sestavljene iz signature, ki poda tipe operacij, in enačb, ki opisujejo njihovo obnašanje. Vsi prestrezniki morajo biti skladni s predpisano teorijo; prestreznik ne sme razlikovati med programi, ki jih v dani teoriji smatramo za enake. Teorije učinkov so običajno globalne, torej morajo vsi prestrezniki spoštovati isto množico enačb, kar omeji nabor možnih prestreznikov. Mnogo kasnejših pristopov zato enačbe odmisli, kar sicer olajša uporabo prestreznikov, vendar močno omeji možnosti dokazovanja lastnosti programov ob prisotnosti računskih učinkov, kjer enačbe igrajo ključno vlogo.


\textbf{Math. Subj. Class. (2010)}: 68N15, 68N18, 03B35, 03B70

\textbf{Ključne besede}: Algebrajski učinki, prestrezniki algebrajskih učinkov, funkcijsko programiranje, teorija programskih jezikov, denotacijska semantika
## Contents

1 Introduction
   1.1 Aim of the thesis ............................................. 2
   1.2 Structure of the thesis ..................................... 2

2 Algebraic effects and handlers .................................. 5
   2.1 The natural need for effects ................................. 5
   2.2 Implementation of computational effects .................... 7
   2.3 Overview of effects handlers ................................. 8
      2.3.1 Effect system ........................................... 10
      2.3.2 Exceptions .............................................. 11
      2.3.3 Printing ................................................ 11
      2.3.4 Mutable state .......................................... 12
   2.4 Reasoning with effects ...................................... 14
      2.4.1 The theory of nondeterminism .......................... 14
      2.4.2 Further applications of local theories ............... 15

3 Core Language .................................................. 17
   3.1 Term syntax ................................................ 17
   3.2 Type syntax ................................................ 19
      3.2.1 Examples of equations ................................. 21
   3.3 Operational semantics ...................................... 23
   3.4 Sugared syntax .............................................. 24

4 Type system ..................................................... 27
   4.1 Subtyping .................................................. 27
   4.2 Typing judgements ......................................... 29
      4.2.1 Well-formed types, contexts, and templates .......... 30
      4.2.2 Typing values, computations, and operation cases .... 31
   4.3 Properties of the type system ............................... 34
      4.3.1 Properties of subtyping ............................... 34
      4.3.2 Safety ................................................ 35
   4.4 Templates to computations .................................. 37

5 Logics .......................................................... 39
   5.1 Empty logic ................................................ 39
   5.2 Free logic .................................................. 39
   5.3 Full logic .................................................. 39
   5.4 Equational logic ........................................... 40
      5.4.1 Inheriting from theory ................................. 45
A Collected judgements
A.1 Operational semantics ........................................... 149
A.2 Typing judgements ............................................ 150
A.3 Equational logic ............................................... 152
A.4 Predicate logic .............................................. 156

B Collected denotational semantics 159
B.1 Type and term semantics ...................................... 159
B.2 Theory semantics ........................................... 160

C Povzetek v slovenščini 163
C.1 Uvod ......................................................... 163
C.2 Prestrezniki algebrajskih učinkov ......................... 164
C.3 Jezik EEFF .................................................. 165
C.4 Sistem tipov ................................................ 166
C.5 Logika ....................................................... 167
C.6 Denotacijska semantika jezika ............................. 168
C.7 Semantika teorij učinkov ................................. 169
C.8 Implementacija in formalizacija .......................... 169
C.9 Zaključek .................................................... 171

XIII
Chapter 1

Introduction

The theory of algebraic effects [33, 34] and effect handlers [36, 38, 37] provides a structured way of modelling computational effects in functional languages. Effects are represented by operations, a special language construct that is called in order to invoke the effect. An operation call captures the continuation of the program and propagates to the nearest encompassing effect handler. Handlers contain a set of instructions to be executed upon intercepting an operation call, with the handler having access to the continuation captured by the intercepted call. The handler may resume the continuation by providing a value, at which point the program proceeds from the origin of invocation, as if the value was produced by the operation call. Algebraic effects and handlers may be thought of as a generalisation of exception handlers, where throwing an exception does not necessitate program termination. The approach can successfully model a number of computational effects and is implemented in a variety of languages (Eff, Koka, Links) and libraries (Multicore OCaml, Pyro).

Operations may come with an effect theory that consists of a type signature, and an equational theory describing effect behaviour. Since operations are interpreted by handlers, these handlers need to be correct with regard to the effect theory. A correct handler does not differentiate between computations considered equal in the effect theory, mapping them to equal results. Determining whether a handler is correct is, in general, undecidable [36] and therefore requires a logic for working with algebraic effects [38]. Proofs of correctness pose an additional burden, but the logic used for proofs of correctness also doubles as a tool for reasoning about other properties of effectful programs.

The restriction for handlers to be correct with respect to a global effect theory can invalidate certain useful handlers. For instance, in the theory of nondeterminism we usually assume commutativity of arguments, meaning that choosing between \( x \) and \( y \) is the same as choosing between \( y \) and \( x \). A handler for nondeterminism that gathers all possible results in a list is, in general, not correct with respect to commutativity of choice, as the resulting lists may have a different order of elements. Using such a handler requires removal of the commutativity equation from the effect theory, which in turn also prevents us from using the equation when reasoning about other parts of the program. The theories that handlers respect are often incompatible, so the global theory usually assumes no equations to allow for a wider variety of handlers, which is the approach in most language implementations and contemporary work [23, 7, 26, 8]. This simplifies the use of handlers but results in a weaker reasoning logic, since equations play a vital role for reasoning in presence of effects.

There are also approaches [2, 3, 18, 43] that focus on utilizing the benefits of algebraic theories. Equations enable the abstraction of properties of effectful computations away from (handler) implementations, a feature that can to some extent be done with
monads [20, 19, 1]. Reasoning about effectful behaviour is important in fields that rely heavily on effects such as probabilistic programming [42, 46], which is of interest due to the possibility of using of handlers [9]. A stronger reasoning logic also provides the basis for other advancements, such as effect-dependent optimisations [24].

1.1 Aim of the thesis

The aim of this thesis is to combine the reasoning utility of algebraic theories with the flexibility of handlers in languages that feature only trivial theories. This should be done in a way that avoids placing a large burden on the end user.

We approach the issue by generalising global effect theories to local effect theories in much the same way that global effect signatures have been adapted to local ones [23]. This results in a stronger logic in parts of the program where an effect theory is assumed, without placing additional restrictions on other parts. Local theories also enable a nested use of different theories pertaining to the same effects by using handlers that act as theory transformers.

The information about effect theories is moved to the type-and-effect system, upgrading it to an effect-theory system. An effect theory mainly affects the typing process for handlers, which are now required to be correct with regard to the local theory. To program under a certain effect theory, we only need to designate it in the types of programs, and the effect-theory system ensures that only handlers respecting the theory can be used. Effect behaviour is abstracted through equations, which decouples the implementation of handlers from reasoning. This in turn allows for better reusability of proofs and provides a healthier environment for creating safe libraries.

The final goal of the thesis is to show applicability of the approach by providing an implementation of a language with local effect theories. We aim to provide a type inference algorithm, with bidirectional type inference showing promise as a natural choice for working with local theories. To avoid circular definitions or other subtle mistakes, we provide a formalisation in the Coq proof assistant.

1.2 Structure of the thesis

In Chapter 2, we informally introduce the concepts of algebraic effects and handlers through examples. We present the treatment of some common computational effects, such as mutable state and nondeterminism. We conclude the chapter by presenting the benefits of equations and displaying the drawbacks of the original approach.

Chapter 3 provides the formal syntax of the language $EEFF$, a prototype language for working with local algebraic theories. The language is equipped with extensions, such as pairs, sums, lists, and recursion. We provide a small-step operational semantics of $EEFF$ and present the syntactic sugar used in examples throughout the thesis.

We introduce the effect-theory system in Chapter 4. Terms are assigned types, which carry equations that can in turn contain terms. This forces judgements for well-formedness and type assignment to be defined through mutually recursive induction. For a well-rounded treatment, we extend the type system with subtyping, as the extension is far from trivial.

In Chapter 5, we describe multiple logics that can be coupled with the effect-theory system. The first three logics are simple systems with an interesting impact on the language. We proceed by constructing two larger logic systems, complete with step-by-step
examples. Due to a strong link between the type system and logic, properties of the type system are shown for each logic separately.

The denotational semantics is split into two parts. Chapter 6 deals with denotations of terms whose interpretations are not affected by effect theories. The denotational semantics is shown to be sound and coherent. In Chapter 7 we account for theories by interpreting equations as partial equivalence relations, which connect elements that we consider equal in a given theory. We identify requirements for logic soundness, which results in desired behaviour of denotations with respect to relations. The chapter is concluded by an informal adaptation of adequacy to the partial equivalence relations.

Chapter 8 focuses on the implementation\(^1\) and formalisation\(^2\) of \textit{EEFF}. The implementation is built on the framework of Eff and uses a bidirectional type inference algorithm. We explain some of the deviations from the theoretical framework, which result in a better user experience. The chapter also provides an overview of the \textit{EEFF} formalisation in the Coq proof assistant. The formalisation formalises all definitions and results of Chapters 3–5 and all results concerning the skeletal language that appears in Chapter 6.

We conclude with Chapter 9 where we compare local effect theories with related work and discuss direction for future work.

\[^1\]https://github.com/zigaLuksic/eff/tree/EEFF
\[^2\]https://github.com/zigaLuksic/eeff-formalization
Chapter 2

Algebraic effects and handlers

This chapter serves as an informal introduction to algebraic effects and handlers as well as the effect-theory system. We assume the reader is familiar with the general concepts of programming languages and advise them to consult [32] for any unfamiliar language constructs. For further introductory examples of handlers, we recommend [39].

2.1 The natural need for effects

Assume we are given the simple task of designing a general-purpose programming language. When treating programming languages only as a means to calculate, very little is required to obtain a powerful tool. In fact, even a language as simple as the untyped $\lambda$-calculus [10, 11] is Turing complete [45]. But working with only the basic $\lambda$-calculus is difficult, so we improve the language by adding extensions, such as integers, strings, pairs, lists, records, pattern matching, etc.

Such an extended $\lambda$-calculus already looks a lot more like the programming languages that we are used to. We have no issues writing a program that sums all the elements of a list via recursion.

\begin{verbatim}
let rec sum_list l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum_list xs
\end{verbatim}

To make the language safer, we introduce a type system, because programmers tend to be error prone at the best of times. We now rest assured that functions of type $A \rightarrow B$ can only be applied to values of type $A$, and that lists of type $A$ list only contain elements of type $A$. We have clearly solved all problems of programming language design. To reap the rewards of such a feat, we attempt to use our language to contact the nearest computer scientist, at which point we are faced with the unwavering stubbornness of $\lambda$-calculus to stay within the boundaries of programs.

None of the aforementioned extensions help us breach the barrier between the program and the outside world. For that we need to extend our language with computational effects, which allow interaction with agents outside of our program. Examples of effectful behaviour are reading the contents of a webpage or instructing a robot on Mars to sing “Happy Birthday”. While non-effectful (pure) programming languages can be very expressive and useful, most general-purpose programming languages require some form of computational effects.
Effects are necessary to communicate with the outside world, but they also benefit the expressivity of the language. A clear-cut case is the widespread use of mutable state and exceptions, which are supported in most major programming languages. Sometimes the use of effects is not necessary, but it improves readability or even efficiency of code. As an example, we tackle the task of writing a function that sums all values in a binary tree.

```ocaml
type tree = Empty | Node of tree * int * tree

let rec sum_tree t =
  match t with
  | Empty -> 0
  | Node (lt, x, rt) -> sum_tree lt + x + sum_tree rt
```

We now make the task more difficult by requiring our function to fail should it encounter a negative value (as an example of a reaction to malformed input data). It is standard to use the `option` type to do so.

```ocaml
let rec sum_tree t =
  match t with
  | Empty -> Some 0
  | Node (lt, x, rt) ->
    if x < 0 then None
    else
      match (sum_tree lt, sum_tree rt) with
      | Some xl, Some xr -> Some (xl + x + xr)
      | _ -> None
```

Here we already make use of advanced pattern matching just to make the function readable. There are further ways to improve this solution, but in all cases, we end up with significantly modified code. We now try to solve the same problem by using exceptions.

```ocaml
exception Malformed

let rec sum_tree t =
  match t with
  | Empty -> 0
  | Node (lt, x, rt) ->
    if x < 0 then raise Malformed
    else sum_tree lt + x + sum_tree rt

let safe_sum_tree t =
  try Some (sum_tree t) with Malformed -> None
```

The modification to the original function is kept minimal, and we recover purity by using exception handlers. While the preferred style is left to the reader, it is obvious that there are problems which are more naturally modelled using effectful behaviour, even when the resulting program is pure.

Another example is working with functions that log their execution. We first write the example by using an effectful function `add_log_message` that prints a message to a dedicated channel. It is implied that it is also used in functions `f`, `g`, and `h`. 

```ocaml
```
Instead of updating the log by using effectful functions, we can simulate such behaviour by threading a special log state through our program. We assume a more advanced functionality, where functions may also read from the log, and so they need a way to access it.

```ml
let important_procedure x y =
    let _ = add_log_message "Starting procedure:" in
    let f_res = f x in
    let g_res = g y f_res in
    let (z1, z2) = h f_res g_res in
    let _ = add_log_message "Done!" in
    somehow_combine z1 z2
```

This approach has a few drawbacks. All functions now accept an additional argument and return a pair instead of just a value, requiring additional unpacking. Even more importantly, a log must include all function executions in the correct order. It is easy to make a mistake when passing it around, just like how in the above example, the log passed to function \( h \) is incorrect.

### 2.2 Implementation of computational effects

There are four common approaches to adding computational effects: adding effects as primitives, extending the language with monads, adding support for delimited control, and the use of algebraic effects and handlers. The last three approaches are expressible in terms of each other [17], so the choice of approach depends on other criteria.

- **Structural approach and extensibility**: Implementing multiple effects with the same structural approach reduces the amount of work needed for the implementation and analysis of the language. Ideally, it is also extensible by the user.
- **Ease of use**: Using computational effects and adapting existing code to use effects should be as painless as possible.
- **Effect information**: The programming language should offer additional information about parts of code that use effects. This is preferably done in the type system.
- **Reasoning**: The extension should preserve as many useful reasoning techniques for pure terms as possible, while also providing new ways to reason about the effectful parts of the code.

When *adding effects as primitives*, we are able to precisely specify the behaviour of each effect. On the other hand, we also *need* to precisely specify the behaviour of each
effect. The behaviour also needs to be hardcoded by the language designer and offers very little in the way of user-defined effects. It is rarely done in a structured manner, bloating the language and requiring careful analysis for each newly added effect. The extensions tend to be efficient and easy-to-use but are often not tracked by the type system. There are methods for reasoning about certain primitive effects, such as the Hoare triples [22], but such tools tend to be restricted to a set collection of effects.

Using monads [31] allows for a more structured approach with strong mathematical foundations. Monads allow user-defined effects and are trackable by the type system. However, using monads requires the switch to a different coding style and tends to become difficult in presence of multiple computational effects. Some of these challenges can be eased by adapting systems such as Haskell’s typeclass system, which, as shown by Haskell’s popularity, makes monads a viable choice. Due to the strong theoretic background there are techniques [19, 20] and tools [29] for reasoning about effectful code.

Delimited continuations [16] can be used for implementing effects [13] by providing delimited control operators that can be utilized to create user-defined effects. A type system and effect system can track use of control operators and, in turn, computational effects. The extension is also backed up by operational and denotational semantics [30], but is lacking when it comes to providing more powerful tools for reasoning.

The approach that is the subject of this thesis is algebraic effects and handlers [33, 34, 36]. Effects are invoked through operation calls, which are used like regular functions and can also be defined by the user. Similar to monads, the language requires a handful of primitive operations such as Print or RandomInt (as the effect needs to be performed by the computer). Such primitive operations can be intercepted by user-defined handlers but result in a computational effect if left unhandled. Algebraic effects can also be equipped with effect systems that support polymorphism [21, 25] and subtyping [41]. The theoretical foundations of algebraic effects lie in free models of equational theories and their homomorphisms, which is helpful in showing program equivalence. Due to practical reasons (i.e. to avoid the restrictions for handlers), many implementations of algebraic effects ignore equations, forfeiting their use in reasoning.

2.3 Overview of effects handlers

In this section we plunge headfirst into easy-to-grasp examples and take a look behind the scenes afterwards. We adopt the syntax of E EFF, which is strongly based on the syntax of Eff 5.0 and Multicore OCaml. We assume that operations have no inherent primitive effects bound to them, but lightly touch on primitive effects in Chapter 8.

To implement algebraic effects, we extend our language with operations and handlers. Operations serve for invoking computational effects by transferring control to the appropriate handler. Every operation is equipped with a type that specifies the type of arguments and a result type of the operation, for instance Print : string -> unit. To invoke the effect, we need to call the operation with an appropriate argument, for which we use the syntax !Print "Calling Print." (this differs from Eff and Multicore OCaml, which use perform (Print "Calling Print."))

We can use operations in the same manner as functions.

```plaintext
let rec print_list l =
  match l with
  | [] -> ()
  | x :: xs -> !Print x; print_list xs
```
We use \( c_1; c_2 \) as the usual sequencing operator, which is equal to \( \text{let } () = c_1 \text{ in } c_2 \). Operation calls may also return a value that can be used in further computations, for instance \( \text{RandomInt : unit} \to \text{int} \).

```ocaml
let rec make_random_pair () =
  let x = !RandomInt () in
  let y = !RandomInt () in
  (x, y)
```

When an operation is called, the call propagates outwards until it is intercepted by a suitable effect handler. If we run \( \text{print_list ['What'; 'will'; 'happen?']} \), the computation would produce an unhandled call of \( \text{Print} \) with the argument "What", with the rest of the program waiting for a response. Calling an operation halts all further evaluation until the effect is resolved, which can only be done by a handler, so unhandled calls effectively terminate the program.

Effect handlers are a set of instructions on how to proceed when an operation is called. Similar to using exception handlers, we wrap effect handlers around computations to await operation calls. An important distinction between operations and exceptions is that the evaluation of the program is not necessarily terminated when an operation is called. The effect handler has access to a \( \text{continuation} \), which captures the remainder of the program at the point of the operation call and can be resumed by the handler to proceed with the evaluation. As an introductory example, we write a simple \( \text{ignore} \) handler.

```ocaml
let ignore = handler
  | effect Print x k -> k ()
```

Here \( \text{effect Print x k -> ...} \) is an \textit{operation case}. When a call of \( \text{Print} \) is intercepted, we bind its argument (the string) to \( x \), the continuation of the program is bound to \( k \), and we proceed with the evaluation of the case instructions. In the above example, we apply the continuation \( k \) to the unit value \( () \), which resumes with the program evaluation. The handler \( \text{ignore} \), as the name suggests, simply ignores operation calls and proceeds as if nothing has happened.

We apply the handler by using the \textit{with ... handle ...} construct to wrap it around a computation.

```ocaml
with ignore handle
  (!Print "Ignore"); (!Print "this."); 12)
```

When we invoke \( !\text{Print} "Ignore" \), the handler intercepts the operation call and the variable \( x \) is set to the value "Ignore". The continuation \( k \) is a function waiting for an argument of the \textit{unit} type, as that is the return type of the operation \( \text{Print} \). At first glance, we assume that \( k \) is equal to

```ocaml
fun y -> (y; !Print "this."); 12)
```

Our approach uses \textit{deep handlers} [23]. This means that every continuation is implicitly handled by the intercepting handler, so we actually have access to the continuation

```ocaml
fun y -> with ignore handle (y; !Print "this."); 12)
```

The \( \text{ignore} \) handler proceeds to simply resume the continuation (by applying it to \( () \)). The next call of \( \text{Print} \) is then also intercepted (and immediately continued) by the \( \text{ignore} \) handler. After dealing with the second print, the computation returns 12.

Handlers can also feature a \textit{value case}, which is applied to the value result of a handled computation. When not specified, we assume that the value case is the identity. By using
a value case we can make `ignore` even more “ignoring” by replacing the result with the unit value.

```ocaml
let completely_ignore = handler
  | effect Print x k -> k ()
  | val x -> ()
```

Value cases use the `val` keyword for better distinction.

### 2.3.1 Effect system

As discussed in Section 2.2, a good type system also tracks effectful behaviour. We embellish types of computations with the information about which operations `may` be called during evaluation. We start by separating terms into values and computations, where values represent information (integers, strings, functions) and computations are instructions (function application, operation calls, sequencing, returning a value). Computations can invoke effects during evaluation, so we upgrade computation types accordingly. The computation type `A!Σ` informs us that the computation produces a value of type `A` while possibly calling operations from the `effect signature` `Σ` during evaluation. A trivial example are computations that call no operations.

\[(1 + 2 + 3): \text{int!}\{\}\]

We use local signatures that also include the type of the operations, which we usually omit to improve readability. An example of a computation featuring an operation call is multiplying a random number.

\[(2 \ast {!\text{RandomInt}}()): \text{int!}\{\text{RandomInt: unit \to int}\}\]

Handler types are of form `A!Σ \Rightarrow B!Σ'` to denote that a handler can handle computations of type `A!Σ`, meaning that its cases cover all effects from `Σ`. While handling, the handler may call operations from `Σ'`, and the final result is a value of type `B`. The handler

```ocaml
let ping_to_pong = handler
  | effect Ping () k -> {!Pong (); k ()}
```

can be assigned the type

\[\text{ping_to_pong}: A!\{\text{Ping: unit \to unit}\} \Rightarrow A!\{\text{Pong: unit \to unit}\}.\]

The operation type can change when handled.

```ocaml
let change_ping_type = handler
  | effect Ping () k -> {!Ping true; k ()}
```

The example is absurd, but the type of `Ping` now differs in the two signatures of the handler type.

\[\text{change_ping_type}: A!\{\text{Ping: unit \to unit}\} \Rightarrow A!\{\text{Ping: bool \to unit}\}\]

An example of a handler that also changes the value type of the resulting computation is the previously mentioned `completely_ignore`.

\[\text{completely_ignore}: A!\{\text{Print}\} \Rightarrow \text{unit!}\{\}\]
2.3.2 Exceptions

With effect handlers being generalisations of exception handlers, it is easy to recover exceptions. To ensure that raising an exception implies termination of evaluation, we make use of the type `empty`. We use the operation `Failwith : string -> empty` as an exception with a message on why the failure occurred. When writing a handler for `Failwith`, the captured continuation expects a value of type `empty`, which (as the name implies) cannot be provided during evaluation. Computations with a type that indicates values of the `empty` type are either nonterminating or raise further exceptions. This ensures that the continuation cannot be resumed, so no matter what handler we use, invoking `Failwith` will force a termination of the current computation. With no option of resumption, using handlers for `Failwith` is equivalent to using exception handlers.

The standard way of modelling computations with possible failure is to use the `option` type, so we write a handler that translates a program using `Failwith` to a program using `Some` and `None`.

```ocaml
let exception_to_option = handler
| effect Failwith msg _ -> None
| val x -> Some x

exception_to_option : A!{Failwith} ⇒ (option A)!{}
```

We need the value case because all cases of a handler need to result in the same type. The handler almost exactly matches the exception handler from the example in Section 2.1.

2.3.3 Printing

A simple way to emulate printing is to change functions of type `A → B` to functions of type `A → B × string`. The new functions additionally return a string that represents the output of the function. However, this approach breaks any kind of effect abstraction, and it is easy to make mistakes when manually passing strings. We can automate this by using handlers.

```ocaml
let collect_prints = handler
| effect Print s k ->
  let (x, out) = k () in
  (x, s ^ out)

collect_prints : A!{Print} ⇒ (A × string)!{}
```

The value case states that if a computation returns a value, it prints nothing. In the case of a `Print` call, we first run the continuation. Because the continuation is implicitly handled, it returns the value of the computation and also the output of its evaluation. The call of `Print` we are currently resolving happens before the continuation, so we concatenate the message to the beginning of the output. Running the program

```ocaml
let rec print_list l =
  match l with
  | [] -> 42
  | x :: xs -> !Print x; print_list xs

let test =
  with collect_prints handle
  print_list ["This "; "works "; "as "; "intended!"]
```
would result in (42, “This works as intended!”).

Handlers can also be nested, so we can use them to “transform” operation calls.

```ocaml
let indent_by_2 = handler
| effect Print s k -> !Print (" " ^ s); k ()

let echo_print echo = handler
| effect Print s k -> !Print s; !Print echo; k ()
```

Using `indent_by_2` will add two spaces and propagate the call outwards, while `echo_print` adds a second call for a predetermined string. This allows us to do some “pretty printing”.

```ocaml
let print_head () = !Print "HEAD"
let print_body () = !Print "BODY"
let print_tail () = !Print "TAIL"

let test =
  with collect_prints handle
  with echo_print "\n" handle
    print_head ();
    ( with indent_by_2 handle
        print_body (); print_body ();
        print_tail ()
    )
```

The above example results in ((), “HEAD
BODY
BODY
TAIL
”). We can display the output as actual text,

```
HEAD
BODY
BODY
TAIL
```

which seems almost useful.

### 2.3.4 Mutable state

Handlers can also be used to implement mutable state. To keep examples short we only emulate a single location, holding values of a type named `state`. We need two operations

- `Get : unit → state`
- `Set : state → unit`

that look up the current value of the state and update it to a new value, respectively. In a similar way as with printing, we use handlers to transform computations into a new form. This time, we change computations into functions that accept a value of `state` and use it to compute a result. Sometimes the final value of the state is important, so we return it along with the result of the computation.

We will use the types as guidelines, so we determine the type of our handler to be

```
state_handler : A!(Get,Set) ⇒ (state → (A × state)!){}!{}
```

In the type on the right side of the arrow ⇒ we see two empty signatures. The outer tells us that our handler will invoke no effects while computing and will return a function of type `(state → (A × state)!){}!).` The empty signature in the function type tells us that running the function will not call any operations.

We start with the easiest part, the value case. Here, we need to transform the value to a function that accepts a state and returns a pair of the value and state.
This corresponds to handling computations that do not use the state at all. The case for \texttt{Set} is slightly trickier.

We first wrap the handler output in a function that waits for the value of state and in the function body we describe how to proceed when we receive the state. The type of \texttt{Set} is \texttt{state} $\to$ \texttt{unit}, so we know that the continuation \texttt{k} is of type $k : \texttt{unit} \to (\texttt{state} \to (A \times \texttt{state})!{})!{}$.

We apply the continuation to transform the remainder of our program to the type \texttt{state} $\to (A \times \texttt{state})!{}}$, which now expects the current value of the state. The call to \texttt{Set} is expected to update the state to \texttt{new_s}, which is the value we pass to the program.

The case for \texttt{Get} is similar; however, we must respond to the operation call by providing the state value.

Let us take a look at a short example to see the behaviour of \texttt{state_handler} (shortened to \texttt{SH}) in action.

Here we use “...” for the remainder of the program, which is not relevant for the example. The value 0 is provided as the initial value of the state. To (partially) evaluate the term \texttt{(with SH handle (!Set 1; ...)) 0}, we handle the call of operation \texttt{Set} according to \texttt{state_handler}.

We first evaluate the outer application, where 0 is discarded.

Once we clean up the left term by resolving the function application and reduce the sequencing, we clearly see how the state changed from 0 to 1.

Let us take a look at a short example to see the behaviour of \texttt{state_handler} (shortened to \texttt{SH}) in action.

Here we use “...” for the remainder of the program, which is not relevant for the example. The value 0 is provided as the initial value of the state. To (partially) evaluate the term \texttt{(with SH handle (!Set 1; ...)) 0}, we handle the call of operation \texttt{Set} according to \texttt{state_handler}.

We first evaluate the outer application, where 0 is discarded.

Once we clean up the left term by resolving the function application and reduce the sequencing, we clearly see how the state changed from 0 to 1.
2.4 Reasoning with effects

As programmers we constantly use program equivalence, even if we do not realize it. For instance, when doing numeric calculation, we rarely stop to think whether we should write 1+x or x+1, despite the terms being syntactically different. We rely on the fact that mathematical primitives of languages are implemented correctly, which allows us to use either of the “mathematically equal” terms. The innate reasoning of programmers stretches even beyond simple arithmetic equivalencies. It is clear that \( \text{fun} \ x \rightarrow () \) 1 is equal to () because we know the evaluation rules of \( \lambda \)-calculus.

But what to do in presence of effectful behaviour? Even if we know that calling \text{print} results in the unit value, we can no longer claim that \text{print} "!" and () are equal. We need to take into account all the ways we interact with the world. At first glance, using algebraic effects makes the problem even worse, because we can’t even rely on our knowledge of effect implementations. In most programming languages, the program \text{print} "A"; \text{print} "B" behaves in the same way as \text{print} "AB", but it is easy to write a handler that invalidates the equivalence; for instance, the previously defined echo \text{print} handler. We can try to show that the equality holds when using collect \text{prints}, but that removes modularity. For instance, in ongoing work on using effect handlers for probabilistic programming, a statistical model is written once and then handled by multiple different handlers (simulating the model or inferring its distribution), so modular reasoning is a necessity.

The problem at hand consists of two parts:

- Reasoning about code using effects, without knowing their precise implementation.
- Making sure that handlers agree with properties of effects.

Both of these can be remedied by using equational theories. The full definition of algebraic effects consists of their signature and equations between them. So far we only concerned ourselves with the signature, but now we make use of the equations as a form of “specification” for the behaviour of effects.

2.4.1 The theory of nondeterminism

The signature of the theory of nondeterminism consists of a single effect representing a binary choice.

\[ \text{Choose} : \text{unit} → \text{bool} \]

To obtain a notation that chooses between two elements we use the function

\[
\text{let} \ \text{choose} \ x \ y = \text{if} \ \text{!Choose} () \ \text{then} \ x \ \text{else} \ y
\]

When working with nondeterminism we tend to rely on certain properties; for instance, a choice between \( x \) and \( x \) is no choice at all—we obtain \( x \) either way. Such properties can easily be stated with equations.

\[
\begin{align*}
\text{choose} \ x \ x & \sim x \\
\text{choose} \ x \ y & \sim \text{choose} \ y \ x \\
\text{choose} (\text{choose} \ x \ y) \ z & \sim \text{choose} \ x (\text{choose} \ y \ z)
\end{align*}
\]

If we now assume that \text{Choice} follows the above equations, we can prove that the following computations are equal.

\[
\text{choose} \ 0 (\text{choose} \ x \ 0) \sim \text{choose} \ x \ 0
\]

A theory provides specifications for effect behaviour, which in turn must be respected by the handler. An instance of a “well-behaved” handler is \text{find}_\text{max}, which returns the largest possible value in the nondeterministic model.
let find_max = handler
    | effect Choose () k -> max (k true) (k false)

Because we treat continuations as regular functions, we may evaluate both branches and then return the larger value. To show that find_max respects the theory, we check that handling both sides of an equation results in equal computations. We take a quick look at how to show that find_max respects the equation

choose x (choose y z) ~ choose (choose x y) z

We first apply the handler to both sides (and simplify the obtained computations).

max x (max y z) ~ max (max x y) z

We know that the above is true for any x, y, and z, thanks to the mathematical properties of the maximum function. The precise process of verifying handler correctness is a bit more intricate, but the idea is largely the same.

Using algebraic theories comes at a price. A very useful handler to use with nondeterministic programs collects all possible results into a list.

let collect_to_list = handler
    | effect Choose () k -> (k true) @ (k false)
    | val x -> [x]

However, collect_to_list does not respect the full theory. The equation for idempotency states that (choose 1 1) and 1 are equal, but handling them with collect_to_list results in [1; 1] and [1] respectively, which are obviously not equal. We usually lose little sleep over discarding handlers that misbehave, but collect_to_list does have its uses.

We solve the problem of restrictive theories by moving equations to the types as well. Because the type system now tracks information about algebraic theories as opposed to just effects, we rename it from a type-and-effect system to an effect-theory system. We skip the details of ensuring that the theories are well formed and plunge straight into the benefits. If we denote $\Sigma := \{\text{Choose}: \text{unit} \to \text{bool}\}$ and the equations of nondeterminism as $\mathcal{E}$, we can assign more accurate types to find_max and collect_to_list.

$$\text{find_max}: \text{int!}^\Sigma/\mathcal{E} \Rightarrow \text{int!}\{\}/\{\}$$

$$\text{collect_to_list}: A!^\Sigma/\{(\text{choose x y) z} \sim \text{choose x (choose y z)}\} \Rightarrow \text{int!}\{\}/\{\}$$

With this, we are not allowed to use collect_to_list on computations that assume all equations $\mathcal{E}$; but we can use both on computations that assume only associativity of choice. The latter calls for the inclusion of subtyping on equations, so that we only need to type a handler once.

2.4.2 Further applications of local theories

In the example of handlers for printing, we showcased handlers that did not “handle away” effects. Certain handlers only modify effect behaviour, but stay within the same signature. In the setting of global theories, all nested handlers had to be correct with respect to the same theory, but that is no longer the case with local theories. In fact, there are certain properties that can only be expressed when viewing the handler as a theory transformer. Recall the definition of echo_print, whose argument we now fix to the line-break character "\n" and rename into add_linebreaks for simplicity.
```ml
let add_linebreaks = handler
| effect Print s k -> !Print s; !Print "\n"; k ()
```

There is very little we can tell about the behaviour of `add_linebreaks` per se. The problem lies in the unknown behaviour of `Print` used in the operation case, but that can be corrected through effect theories. A reasonable theory to map into is

\[
E_{out} := \{ !\text{Print } x; !\text{Print } y \sim !\text{Print } (x \sim y) \},
\]

which is also a theory respected by the `collect_prints` handler. If we assume this equation, then we have enough information to conclude that `add_linebreaks` respects

\[
E_{in} := \{ !\text{Print } x; !\text{Print } y \sim !\text{Print } (x \sim \text{"\n"} y) \}.
\]

The type we wish to assign to the handler is

\[
\text{add_linebreaks : } A!\{\text{Print}\}/E_{in} \Rightarrow A!\{\text{Print}\}/E_{out}.
\]

This means that `add_linebreaks` implements `Print` in a way that is compliant with `E_{in}`, specified in the `A!\{Print\}/E_{in}` part of the type. The handler in turn again uses `Print`, but operation calls invoked by the handler follow the theory `E_{out}` instead, which can be seen in the outgoing type `A!\{Print\}/E_{out}`.

In order to type the handler, we must show that the handler respects `E_{in}`; but when doing so, we can also use equations from `E_{out}`. We start the proof by “handling” both sides of the equation `E_{in}`.

\[
!\text{Print } x; !\text{Print } "\n"; !\text{Print } y; !\text{Print } "\n" \sim !\text{Print } (x \sim \text{"\n"} y); !\text{Print } "\n"
\]

Thanks to `E_{out}`, we know how to combine consecutive prints and end up with two equivalent computations.

\[
!\text{Print } (x \sim \text{"\n"} y \sim \text{"\n"}); !\text{Print } (x \sim \text{"\n"} y \sim \text{"\n"})
\]

This goes to show that local theories play a crucial role in both components of a handler type, and that such theory-transforming handlers stand no chance of being typed in a global theory approach.
Chapter 3

Core Language

Terms and operational semantics of EEFF closely mirror those of Eff [6, 39] with some minor differences, such as the use of closed handlers. EEFF includes common extensions such as pairs, type sums, lists, and recursion. This is a direct upgrade of our earlier work [28] and allows for a thorough treatment of the intricate type system introduced in Chapter 4. Multiple extensions enable more complex examples and improve the usefulness of formalisation as a reasoning tool.

To avoid ambiguity in typing some terms require type annotations. The inclusion of equations in types, on the other hand, requires the syntax of types to depend on that of terms. We must thus define the term and type syntax simultaneously; in fact, we require a mutually recursive definition for:

- values \( v \)
- computations \( c \)
- operation cases \( h \)
- value types \( A, B \)
- computation types \( C, D \)
- effect signatures \( \Sigma \)
- contexts \( \Gamma \)
- template contexts \( Z \)
- templates \( T \)
- equations \( E \)

We separate the definitions into more digestible chunks, but the strong links between the constructs should be taken into account when proving properties of the system. In Section 3.1 we introduce the syntax of terms, and in Section 3.2 we present the syntax of types.

While type annotations on terms are important for the type system, we sometimes omit them for clarity. We introduce other aesthetic corrections (sugared syntax) in Section 3.4.

3.1 Term syntax

We use a fine-grained call-by-value style [27] that differentiates between values \( v \) and computations \( c \). In most contemporary work [39, 25, 8], operation cases are part of the handler definition, but separating operation cases \( h \) as an additional sort allows for a more natural treatment.

Figure 3.1 presents the syntax for value terms. Handlers are constructed from a value case \( \text{ret} (x : A) \mapsto c_r \) and operation cases \( h \). Value cases of handlers and functions are
values \( v := \)
\[
| x \quad \text{variable} \\
| () \quad \text{unit} \\
| n \quad \text{integer} \\
| \text{fun} \ (x : A) \mapsto c \quad \text{function} \\
| \text{handler} \ (\text{ret} \ (x : A) \mapsto c; h) \quad \text{handler} \\
| (v_1, v_2) \quad \text{pair} \\
| \text{Left}_{A+B} \ v \ | \ \text{Right}_{A+B} \ v \quad \text{sum constructors} \\
| [] A \quad \text{empty list} \\
| v_1 :: v_2 \quad \text{list constructor}
\]

Figure 3.1: Syntax of values.

always annotated with the type of the argument. Sum constructors \text{Left} and \text{Right} also require type annotations and so does the empty list \([]\).

computations \( c := \)
\[
| \text{ret} \ v \quad \text{returned value} \\
| \text{do} \ x \leftarrow c_1 \ \text{in} \ c_2 \quad \text{sequencing} \\
| v_1 \ v_2 \quad \text{function application} \\
| \text{let rec} \ f \ x : A \to C = c_1 \ \text{in} \ c_2 \quad \text{recursive function} \\
| op_{A\to B}(v; y).c \quad \text{operation call} \\
| \text{with} \ v \ \text{handle} \ c \quad \text{handler application} \\
| \text{absurd}_C \ v \quad \text{empty value elimination} \\
| \text{match} \ v \ \text{with} \ (x, y) \mapsto c \quad \text{product elimination} \\
| \text{match} \ v \ \text{with} \ \text{Left} \ x \mapsto c_1 \ | \ \text{Right} \ y \mapsto c_2 \quad \text{sum elimination} \\
| \text{match} \ v \ \text{with} \ [] \mapsto c_1 \ | \ x :: xs \mapsto c_2 \quad \text{list elimination}
\]

Figure 3.2: Syntax of computations.

The syntax of computations is introduced in Figure 3.2. We use \text{ret} as a way of lifting values to computations, and we use \text{do} \ x \leftarrow c_1 \ \text{in} \ c_2 for sequencing computations, where the result of \( c_1 \) is bound to the variable \( x \) and can be used in \( c_2 \). We prefer the notation of \text{do} as opposed to \text{let}, as it carries a stronger connotation of effectful behaviour. Recursion is available through \text{let rec} \ f \ x : A \to C = c_1 \ \text{in} \ c_2, where \( c_1 \) is the function definition and \( c_2 \) the computation that is evaluated next and may use \( f \). Operation call \( op_{A\to B}(v; y).c \), with \( op_{A\to B} \) being the type-annotated name of the operation, contains a value argument \( v \) and a continuation \( y.c \). Continuations are kept syntactically different from functions, but can be viewed as \text{fun} \ (y : B) \mapsto c. The term \text{with} \ v \ \text{handle} \ c wraps the handler \( v \) around the computation \( c \) to intercept all operation calls that occur during evaluation of \( c \). The language includes the \text{empty} type, so we add \text{absurd} as a way of eliminating values of such a type. The use of \text{absurd} is important for operations that model exceptions. The other data constructors are eliminated with \text{match} statements that provide instructions based on the shape of the value.

Operation cases are represented as a set (shown in Figure 3.3) to avoid issues with the order of cases. Every case states what operation it handles, where variables \( x \) and \( k \)
operation cases $h ::= $

| $\emptyset_D$ | empty cases
| $h \cup \{op_{A\to B}(x; k) \mapsto c_{op}\}$ | operation case

Figure 3.3: Syntax of handler operation cases.

bind the argument and continuation of the call, while the computation $c_{op}$ represents the instructions to be carried out when a call is intercepted. Empty cases are annotated with the result type to avoid ambiguity in typing.

3.2 Type syntax

Types of EEFF work with local signatures [23]. The syntax for value types is presented in Figure 3.4 and remains similar to earlier work [6, 7, 41].

value type $A, B ::=$

| unit | unit type
| empty | empty type
| int | integer type
| $A \to C$ | function type
| $C \Rightarrow D$ | handler type
| $A \times B$ | product type
| $A + B$ | sum type
| $A$ list | list type

Figure 3.4: Value type syntax.

We use the mathematical notation for type products and sums, but adopt the OCaml syntax for the type of lists.

computation type $C, D ::= A!\Sigma/E$

signature $\Sigma ::= \{\} | \Sigma \cup \{op : A \to B\}$

Figure 3.5: Computation type syntax.

Computation types $A!\Sigma/E$ are built from three parts: the type of returned values $A$, the local operation signature $\Sigma$, and the equations of the local theory $E$. The signature contains the names and types of all operations that may be called within a computation of type $A!\Sigma/E$, while the equations $E$ tell us which computations are considered equal at that type.

Operation cases are treated as a separate entity in the term syntax (and receive separate judgements in the type system), but we do not construct a separate type for them. However, for easier discussion and improved readability, we use the following notation for cases that
cover operations of $\Sigma$ and handle them with computations of type $D$.

$$\Sigma \Rightarrow D$$
type of operation cases

There exists a relation between the handler type $A!\Sigma/E \Rightarrow D$ and the cases type $\Sigma \Rightarrow D$, which becomes clearer in Chapter 4 where we define typing judgements.

template $T ::=$
- $z v$ applied template variable
- $\text{op}_{A \rightarrow B}(v; y. T)$ operation call
- $\text{do pure } x \leftarrow c \text{ in } T$ effect-free sequencing
- $\text{absurd } v$ empty value elimination
- $\text{match } v \text{ with } (x,y) \mapsto T$ product elimination
- $\text{match } v \text{ with } \text{Left } x \mapsto T_1 | \text{Right } y \mapsto T_2$ sum elimination
- $\text{match } v \text{ with } [] \mapsto T_1 | x :: xs \mapsto T_2$ list elimination

effect theory $E ::= \{ \} \mid E \cup \{ \Gamma; Z \vdash T_1 \sim T_2 \}$

Figure 3.6: Equation syntax.

Equations are constructed by relating a pair of templates that represent computations of a certain shape. Templates use a restricted set of building blocks from the language syntax, combined with an additional construct called a template variable, which represents an “arbitrary computation”. To represent computations that await a value, template variables are applied in a similar way to functions. Another modification is the do pure sequencing, which is used to evaluate computations that do not call any operations; however, they can be nonterminating (which conflicts with some notions of purity). This allows one to use functions such as $+$ in templates.

context $\Gamma ::= \cdot \mid \Gamma, x : A$

template context $Z ::= \cdot \mid \Gamma, z : A \rightarrow *$

Figure 3.7: Syntax of contexts.

Templates use regular variables as well as template variables, so equations also include a context $\Gamma$ and template context $Z$. A context is a collection of variable names $x$ and their assigned types $A$, while a template context has template variables $z$ with types of form $A \rightarrow *$. Here the symbol $*$ is a wildcard type that is instantiated to a computation type when needed. Empty contexts are denoted with the $\cdot$ symbol.

In templates we only describe the shape of programs. For example,

$$\text{op}_{A \rightarrow B}(v; y. z ( ))$$

represents any computation that starts by calling $op$ and then ignores its output, since the template variable $z$ is applied to $()$ and not $y$. All effects in templates must occur through explicit operation calls, so do pure only accepts computations that invoke no effects (enforced in type system). It is nonetheless a useful inclusion, as it allows the use of terms such as $x + y$ in templates.
Templates also include \texttt{match} and \texttt{absurd} statements to provide a branching mechanism. The building blocks ensure that every evaluation branch results in a template variable (or is impossible), which allows a single template to represent computations of multiple types. If you imagine replacing template variables of type $A \rightarrow \ast$ with functions of type $A \rightarrow C$, then the template represents a computation of type $C$. The process of instantiating templates to computations is explained in further detail in Section 4.4, but follows the same idea.

Ideally, the template language should mirror the term language as much as possible, but restrictions provide useful benefits. Ensuring that all evaluation branches end with template variables is necessary for reusability and also simplifies the type system. The restriction that effects only occur through explicit operation calls, provides a natural way of dealing with handler correctness in the logic. Even a restricted template language is powerful enough for a wide variety of effect theories, and we consider the restrictions reasonable, given their advantages.

### 3.2.1 Examples of equations

To familiarise the reader with templates, we provide some examples of effect theories.

**Theory of nondeterminism**

We start with the theory of nondeterminism that we informally wrote down in Chapter 2.

\begin{align*}
\text{Choose} & : \text{unit} \rightarrow \text{bool} \\
\text{choose } x \ y & = \text{if } !\text{Choose ()} \text{ then } x \text{ else } y \\
\text{choose } x \ x & \sim x \\
\text{choose } x \ y & \sim \text{choose } y \ x \\
\text{choose} (\text{choose } x \ y) \ z & \sim \text{choose } x (\text{choose } y \ z)
\end{align*}

The above notation uses Booleans and conditionals, which we can emulate with \texttt{unit+unit} and \texttt{match}. We first construct a suitable signature.

\begin{align*}
\text{Choose} & : \text{unit} \rightarrow \text{unit + unit} \\
\text{Choose} (()) \text{; } \text{match } y \text{ with Left } _- \mapsto z (()) \text{ | Right } _- \mapsto z () \sim z ()
\end{align*}

Here $z ()$ is meant to be read as “continue with arbitrary program”. It is important to note that every use of $z$ represents the same arbitrary program. So for the commutativity equation, we use two template variables.

\begin{align*}
\text{Choose} (()) \text{; } y \text{.match } y \text{ with Left } _- \mapsto z_1 () \text{ | Right } _- \mapsto z_2 () \\
\sim \\
\text{Choose} (()) \text{; } y \text{.match } y \text{ with Left } _- \mapsto z_2 () \text{ | Right } _- \mapsto z_1 ()
\end{align*}
The above are actually not full equations, as they are missing both contexts. For the commutativity equation we use no value variables, so the value context is empty, and to account for the two templates, the equation has the contexts
\[ \cdot; z_1 : \text{unit} \to *, z_2 : \text{unit} \to \ast \rightarrow \ldots \]

While contexts are important, we tend to omit them, as we trust the reader can infer necessary types. For nondeterminism, we also adopt a shorter notation for branching and applying template variables.

\[ T_1 \oplus T_2 := \text{Choose}(); y.\text{match} y \text{ with Left } \_ \mapsto T_1 \mid \text{Right } \_ \mapsto T_2 \]
\[ \vec{z} := z() \]

We now collect all equations of nondeterminism (with omitted contexts).

\[ \vec{z} \oplus \vec{z} \sim \vec{z}, \quad (\text{IDEM}) \]
\[ \vec{z}_1 \oplus \vec{z}_2 \sim \vec{z}_2 \oplus \vec{z}_1, \quad (\text{COMM}) \]
\[ \vec{z}_1 \oplus (\vec{z}_2 \oplus \vec{z}_3) \sim (\vec{z}_1 \oplus \vec{z}_2) \oplus \vec{z}_3 \quad (\text{ASSOC}) \]

**Theory of mutable state**

We already encountered the signature of mutable state in Chapter 2. We use `state` as an abstract type that represents the contents of state.

\[ \text{Get} : \text{unit} \to \text{state} \quad \text{Set} : \text{state} \to \text{unit} \]

The equations of state pertain to the interaction between consecutive operation calls. Template variables are either of type `unit \to \ast`, `state \to \ast`, or `state \times state \to \ast`.

\[ \begin{align*}
\text{Get}(); s.\text{Get}(); s'.z(s,s') & \sim \text{Get}(); s.z(s,s) & (\text{GETGET}) \\
\text{Get}(); s.\text{Set}(s; \_z()) & \sim z() & (\text{GETSET}) \\
\text{Set}(s; \_z(); s'.z(s')) & \sim \text{Set}(s; \_z s) & (\text{SETGET}) \\
\text{Set}(s; \_z(); s'.z()) & \sim \text{Set}(s'; \_z()) & (\text{SETSET})
\end{align*} \]

The equations tell us the following:

- \text{GETGET}: Looking up a value does not change its value.
- \text{GETSET}: Updating the state with its current value has no effect.
- \text{SETGET}: Updating a state sets its value to the argument of the update.
- \text{SETSET}: Consecutively changing the value is the same as simply changing it to the final value.

**Non-traditional theory of print**

An example of a non-traditional effect theory is used in a theory-transforming handler for `Print` in Chapter 2. The equation we used is also relevant, because it is an example that could not be written in our previous work [28], as it lacked a way to apply pure functions.

\[ \begin{align*}
\text{Print}(s_1; \_\text{Print}(s_2; \_z())) & \sim \text{do pure } s \leftarrow s_1 \cdot s_2 \text{ in Print}(s; \_z())
\end{align*} \]

Here we use \( \cdot \) as the built-in operator for string concatenation. To avoid troublesome debates over built-in functions in templates, we solve a different—yet similar—problem. Assume we use lists instead of strings; since `EEFF` features recursion, we can define the list
concatenation operator @ within our language. We can now use the do pure template to first define @ inside the template and then a further do pure to apply it.

Another option for using functions inside templates would be to use special pure functions [37]. This requires a special set of primitive pure functions or a significant extension of the language to allow user-defined pure functions.

### 3.3 Operational semantics

We present a small-step evaluation relation for EEFF that closely follows [6, 7], since effect theories bear no effect on program evaluation. We use the notation c[x ↦ v] to mean “in c, replace all unbound occurrences of x with v”. We assume the reader is familiar enough with capture avoiding substitution that they can generalise it to any new constructs.

Substitution is an important part of the system but, not the focus of this thesis. We provide more details on how substitution was formalised in Chapter 8. The rules of operational semantics are presented and described throughout the section, but we provide a compact collection in Appendix A.1.

In the fine-grained call-by-value setting, only computations are evaluated, so values receive no evaluation rules. Both \( \text{ret} v \) and \( \text{op}_{A \rightarrow B}(v; y.c) \) are treated as results and in turn also receive no evaluation rules. The computation \( c \), captured in the continuation of \( \text{op}_{A \rightarrow B}(v; y.c) \), is only evaluated further if the continuation is called during the handling of the operation call.

The rules for sequencing evaluate the first component until it produces a result. If the computation produces a value, it is extracted from the \( \text{ret} \) construct and substituted for \( x \) in \( c_2 \). In the case of an operation call, we propagate the call outwards while expanding the continuation to ensure that operations reach handlers with the correct continuation.

\[
\begin{align*}
\frac{c_1 \leadsto c'_1}{\text{DoStep}} & \quad \text{do } x \leftarrow c_1 \text{ in } c_2 \leadsto \text{do } x \leftarrow c'_1 \text{ in } c_2 \\
\frac{c \leadsto c[x \mapsto v]}{\text{DoRet}} & \quad \text{do } x \leftarrow \text{ret } v \text{ in } c \leadsto c[x \mapsto v] \\
\frac{c_2 \leadsto \text{op}_{A \rightarrow B}(v; y.c_1)}{\text{DoOp}} & \quad \text{do } x \leftarrow \text{op}_{A \rightarrow B}(v; y.c_1) \text{ in } c_2 \leadsto \text{op}_{A \rightarrow B}(v; y.\text{do } x \leftarrow c_1 \text{ in } c_2)
\end{align*}
\]

It is important to note that the “bubble up” effect of operation calls does not change the meaning of the program, because \( y \) is only present in \( c_1 \). Both computations expect the result of the operation call to be bound to \( y \), and then proceed with evaluation of \( c_1 \).

We encounter functions either through function application or the let rec construct.

\[
\frac{(\text{fun } (x : A) \mapsto c) \mapsto c[x \mapsto v]}{\text{APPFUN}}
\]

\[
\frac{(\text{let rec } f : A \rightarrow C = c_1 \text{ in } c_2 \mapsto c_2[f \mapsto (\text{fun } (y : A) \mapsto \text{let rec } f : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])])}{\text{LETRecStep}}
\]

Function application is the same as in \( \lambda \)-calculus and replaces all occurrences of the bound variable with the argument value. The operational semantics for recursive functions is a bit more involved. The \( c_1 \) part of \( \text{let rec } f : A \rightarrow C = c_1 \text{ in } c_2 \) only serves as the function definition, with \( c_2 \) representing the actual computation to be evaluated. The rule substitutes all occurrences of \( f \) with

\[
(\text{fun } (y : A) \mapsto \text{let rec } f : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])
\]
and in a way “installs” the function in \( c_2 \). Making the `let rec` construct part of the function ensures that if the function is called, it will first replace \( y \) with the argument in \( c_1[x \mapsto y] \) and then “install” the recursive definition of \( f \). The cycle repeats only when we encounter a recursive call to \( f \).

We handle a computation by evaluating it until either it provides a value, or we intercept an operation call. Depending on the situation, we either use the value case or find the appropriate operation case. It should be noted that the type annotations of the intercepted call might differ from the annotations used in handler cases. Evaluation rules are only affected by the operation name, and we rely on the type system to ensure that the annotations are compatible.

\[
\begin{align*}
& c \rightsquigarrow c' \\
\text{with \( v \) handle} & \quad c \rightsquigarrow c' & \text{HANDLESTEP} \\
\text{with (handler \( (\text{ret \( (x : A) \mapsto c_r; h) \) handle \( (\text{ret \( v) \mapsto c \_x \mapsto v} \])} & \quad c_2[x \mapsto v] \quad \text{HANDLERet} \\
H = \text{handler \( (\text{ret \( (x : A) \mapsto c_r; h) \text{ (} o \_p A;_r \text{ op } B;_r \text{ op}(x; k) \mapsto c \_op) \) \in h} & \quad \text{with \( H \) handle} \quad (o \_p A;_r \text{ op } B;_r \text{ op}(v; y; c)) \mapsto c \_op[x \mapsto v, k \mapsto (\text{fun} (y : B;_r \text{ op}) \mapsto \text{with \( H \) handle \( c)})] & \quad \text{HANDLEOp}
\end{align*}
\]

When handling an operation, we transform the continuation into a function and wrap it with the same handler that intercepted the call. This implements deep handlers, while the other approach, shallow handlers, would skip the implicit handling. We also provide no rule for the scenario where \( op \) has no matching case in \( h \), because handlers in EEFF are closed. For open handlers, we would add a rule that allows unknown operations to propagate outwards. Open handlers tend to be more modular, but they complicate effect-theory systems, as discussed in Chapter 9.

The match statements all follow the same approach. We extract the data from values and continue down the appropriate branch.

\[
\begin{align*}
\text{match} \quad (v_1, v_2) \text{ with} & \quad (x, y) \mapsto c \rightsquigarrow c[x \mapsto v_1, y \mapsto v_2] & \text{MATCHPAIR} \\
\text{match} \quad \text{Left}_{A;_r \text{ op } B} \quad v \text{ with} & \quad \text{Left} \quad x \mapsto c_1 \quad \text{Right} \quad y \mapsto c_2 \rightsquigarrow c_1[x \mapsto v] & \text{MATCHLEFT} \\
\text{match} \quad \text{Right}_{A;_r \text{ op } B} \quad v \text{ with} & \quad \text{Left} \quad x \mapsto c_1 \quad \text{Right} \quad y \mapsto c_2 \rightsquigarrow c_2[y \mapsto v] & \text{MATCHRIGHT} \\
\text{match} \quad [ ] \text{ with} & \quad [ ] \mapsto c_1 \quad \text{with} \quad [ ] \mapsto c_1 \quad x :: xs \mapsto c_2 \rightsquigarrow c_1 & \text{MATCHNIL} \\
\text{match} \quad (v :: vs) \text{ with} & \quad [ ] \mapsto c_1 \quad x :: xs \mapsto c_2 \rightsquigarrow c_2[x \mapsto v, xs \mapsto vs] & \text{MATCHCONS}
\end{align*}
\]

Seeing as there is no way to produce data of the empty type, we do not need to provide a rule for absurd.

### 3.4 Sugared syntax

The core language has a large amount of functionality; however, when writing examples, we will default to a sugared syntax, with the desugaring briefly covered in this section.
only sketch the process of desugaring types and terms, which we denote with \( \Rightarrow \). Sugared terms in examples do not feature type annotations unless it is relevant to the example at hand.

When using type sums and products with multiple components, we assume that operators associate to the left. We also adopt the OCaml syntax for writing lists.

\[
\begin{align*}
A_1 + A_2 + A_3 & \Rightarrow (A_1 + A_2) + A_3 \\
A_1 \times A_2 \times A_3 & \Rightarrow (A_1 \times A_2) \times A_3 \\
(x_1, x_2, x_3) & \Rightarrow ((x_1, x_2), x_3) \\
\left[x_1; x_2; x_3\right] & \Rightarrow x_1 :: (x_2 :: (x_3 :: \[]))
\end{align*}
\]

We construct Booleans and conditionals through the use of type sums.

\[
\begin{align*}
\text{bool} & \Rightarrow \text{unit} + \text{unit} \\
\text{true} & \Rightarrow \text{Left}_\text{unit+unit} () \\
\text{false} & \Rightarrow \text{Right}_\text{unit+unit} ()
\end{align*}
\]

\[
\text{if } v \text{ then } c_1 \text{ else } c_2 \Rightarrow \text{match } v \text{ with Left } _{} \mapsto c_1 | \text{Right } _{} \mapsto c_2
\]

The encoding of the frequently used \texttt{option} type is done with type sums as well.

\[
\begin{align*}
A \text{ option} & \Rightarrow A + \text{unit} \\
\text{Some } x & \Rightarrow \text{Left}_{A + \text{unit}} x \\
\text{None} & \Rightarrow \text{Right}_{A + \text{unit}} ()
\end{align*}
\]

\[
\text{match } v \text{ with Some } x \mapsto c_1 | \text{None} \mapsto c_2 \Rightarrow \text{match } v \text{ with Left } x \mapsto c_1 | \text{Right } _{} \mapsto c_2
\]

Continuations in operation calls are not meant to be explicitly provided by the programmer. We therefore provide syntactic sugar for \textit{generic operations}.

\[
!op \ v \Rightarrow op_{A \rightarrow B}(v; y.\text{ret } y)
\]

This allows us to use operations like functions, and we rely on operational semantics to capture the correct continuation when the operation is called.

The sugared notation for handlers closely follows the internal one, we just adapt a more \textit{match}-like style.

\[
\text{handler} \Rightarrow \begin{align*}
\text{handler} (\text{ret } (x : A) & \mapsto c_r; \{ op_{A \rightarrow B}(x; k) \mapsto c_{op}, \ldots \}) \Rightarrow \\
& | \text{ret } x \mapsto c_r | op \ x \ k \mapsto c_{op} \\
& \vdots
\end{align*}
\]

We also improve functions with multiple arguments by using a simpler notation. If the function does not invoke any effects until all arguments are provided, we omit the empty signature marked by \(!\}/!\} from intermediate computation types.

\[
\begin{align*}
A \rightarrow B \rightarrow C & \Rightarrow A \rightarrow (B \rightarrow C)!/!} \\
\text{fun } x \ y & \mapsto c \Rightarrow \text{fun } (x : A) \mapsto \text{ret } (\text{fun } (y : B) \mapsto c) \\
\text{let } f \ x = c \text{ in } \ldots & \Rightarrow \text{do } f \leftarrow \text{ret } (\text{fun } (x : A) \mapsto c) \text{ in } \ldots
\end{align*}
\]

We also adopt the common notation for ignored arguments \_ and some pattern matching, such as using () instead of a variable name to denote that a unit value is expected.
Fine-grained call-by-value style is useful for specifications and analysis, but is far less desirable for writing code. We thus translate to fine-grained CBV during desugaring by inserting appropriate do statements. All variables marked with ' in the following rules must be fresh.

\[(c_1, c_2) \Rightarrow \text{do } x'_1 \leftarrow c_1 \text{ in do } x'_2 \leftarrow c_2 \text{ in } (x'_1, x'_2)\]
Left \(c \Rightarrow \text{do } x' \leftarrow c \text{ in Left}_{A+B} x'\)
\(c_f \ c_x \Rightarrow \text{do } f' \leftarrow c_f \text{ in do } x' \leftarrow c_x \text{ in } f' \ x'\)
\(\text{with } c_h \text{ handle } c \Rightarrow \text{do } h' \leftarrow c_h \text{ in with } h' \text{ handle } c\)

In cases where the evaluation order is important, we will emphasize it by using desugared syntax.

We are able to write much more concisely by using the sugared syntax.

```
(* Using no syntactic sugar *)
do opt_apply <-
  ret (fun f -> ret (fun x ->
    match x with
    | Left a -> f a
    | Right b -> ret (Right ())
  ))
in
do report_success <- ret (fun x ->
  match x with
  | Left a -> Success(() ; y. ret y)
  | Right b -> Fail(() ; y. ret y)
 )
in
opt_apply report_success
```

```
(* A shorter version with sugared syntax *)
let opt_apply f x =
  match x with
  | Some a -> f a
  | None -> ret None
in
opt_apply (fun x -> if x then !Success () else !Fail ())
```
Chapter 4

Type system

We begin by defining a subtype relation in Section 4.1 and continue with judgements of the type system in Section 4.2. Due to templates, we need to check that types are well formed (Subsection 4.2.1) before we can assign types to terms (Subsection 4.2.2). In Section 4.3 we show that all components of the proposed system interact well. We conclude the chapter by presenting a procedure for instantiating templates to computations in Section 4.4.

4.1 Subtyping

The decision to include subtyping is twofold. Firstly, the addition of subtyping is often nontrivial, perhaps even more so when we are dealing with effect systems [41]. It would be foolish to dismiss it as a trivial extension in such a heavily coupled type system. Secondly, a language with an effect system requires either subtyping or a notion of effect polymorphism to be usable in practice. As an example, assume we have a handler of type

\[ h : A!(\text{Ping, Pong})/\{} \Rightarrow D \]

Clearly, we should be able to use it to handle computations of type \( A!(\text{Ping})/\{} \), since the handler covers all possible operations and even some that cannot occur. Similarly, if we need to provide a computation of type \( A!(\text{Ping})/\{} \), any computation of type \( A!/\{}/\{} \) should also be acceptable, as both types indicate a returned value of type \( A \) and possible calls of \text{Ping}.

It comes as no surprise that a similar problem occurs with equations, even in cases where effect signatures fit the requirements. If a handler respects equations \( E \), then it can always be used on a computation that assumes a subset of those equations. And for a more concrete example, let us assume that the nondeterministic choice is associative, in which case there is no reason to prevent the use of a handler that implements it as both associative and idempotent.

The idea of subtyping is to provide a relation between types, \( A \leq A' \), which holds if we can treat elements of \( A \) as if they were of type \( A' \). Another interpretation is that we can always safely replace a term of type \( A' \) with a term of type \( A \).

The subtyping relation

We define the subtyping relation separately for value types, computation types, signatures, and equations, but use the same \( \leq \) symbol for brevity. Appendix A.2 features all rules in a single figure.
We include reflexivity only for base types and later show that it also holds for other types.

\[
\begin{align*}
\text{unit} & \leq \text{unit} & \text{STyUnit} \\
\text{int} & \leq \text{int} & \text{STyInt} \\
\text{empty} & \leq \text{empty} & \text{STyEmpty}
\end{align*}
\]

\[
\begin{align*}
A & \leq A' & B & \leq B' & \text{STySum} \\
A + B & \leq A' + B' & A & \times B & \leq A' \times B' & \text{STyProd} \\
A & \leq A' & \text{STyList} \\
A & \leq A' & \text{STyFun} \\
A' & \leq A & C & \leq C' & \text{STyHandler}
\end{align*}
\]

We define subtyping for computation types by subtyping component-wise.

\[
A \leq A' \quad \Sigma \leq \Sigma' \quad E \leq E' \quad A!\Sigma/E \leq A'!\Sigma'/E' \quad \text{STyCTy}
\]

Signatures only tell us which effects may happen, so we can always safely increase the set of possible effects. A non-standard extension is to allow subtyping on the types of operations with the same name to fully utilize the possibilities of local signatures. We want that \(\Sigma \leq \Sigma'\) holds, if every operation in \(\Sigma\) is also present in \(\Sigma'\), and if the types of operations differ, we require them to be compatible (through subtyping).

\[
\{\} \leq \Sigma \quad \Sigma \leq \Sigma' \quad \text{op} : A' \rightarrow B' \in \Sigma' \quad A \leq A' \quad B' \leq B \quad \Sigma \cup \{\text{op} : A \rightarrow B\} \leq \Sigma' \quad \text{STySIGU}
\]

When subtyping signatures, the requirements for operation types are the opposite of those for subtyping function types. This stems from the fact that signatures act as some sort of an operation context. Interpretations of operations are given through handlers, where operation types occur in a contravariant position. Choosing such a notion of signature subtyping affects the rules for \(\text{WfTOP}\) and \(\text{TypeOP}\) (presented later), where additional premises are needed due to type annotations on operation calls.

With equations we stick to the basic subset relation, meaning that \(E \leq E'\) if every equation of \(E\) is also present in \(E'\).

\[
\{\} \leq E \quad E \leq E' \quad \Gamma ; Z + T_1 \sim T_2 \in E' \quad \text{STyEqs} \quad E \cup \{\Gamma ; Z + T_1 \sim T_2\} \leq E' \quad \text{STyEqsU}
\]

Another option would be to allow \(E \leq E'\) if all equations of \(E\) follow from equations of \(E'\). Checking when a theory entails another has to be done in a logic system, and we postpone that extension to future work due to two reasons:

- Subtyping would have to be defined as part of the mutually recursive definition of well-formedness, typing judgements, and logic rules. This would make the system even more coupled while providing little additional insight.
- The inference algorithm described in Chapter 8 cannot infer logic proofs. This means that whenever equation subtyping would occur, the user would have to provide a logic
proof, unlike with the current subset relation that can be automated. While we can argue that logic proofs for handlers are meant to be done by experts when they are writing libraries, subtyping can occur at any point in the code, making it cumbersome for end users.

This problem is further alleviated by theory-transforming handlers. We construct an identity handler $H$ with the cases \{ $\text{op}(x; k) \mapsto \text{op}(x; y, k, y)$ \}, which simply forward all operation calls. We then assign the handler a type that translates the theory $E$ to $E'$.

\[
\cdot \vdash H : A\Sigma/E \Rightarrow A\Sigma/E'
\]

Typing $H$ at such a type requires precisely the proof that all equations of $E$ follow from equations of $E'$. If we can provide such a proof, $H$ can be used as an explicit coercion. This also directly translates the problem of equation subtyping to the problem of handler correctness.

Despite not needing context subtyping in any rules, it is a useful notion for stating certain properties. Context subtyping proceeds structurally.

\[
\cdot \leq \cdot \quad \text{STyCtx} \quad \Gamma \leq \Gamma' \quad A \leq A'
\]

\[
\Gamma, x : A \leq \Gamma', x : A' \quad \text{STyCtx}
\]

### 4.2 Typing judgements

A condensed view of the judgements can be found in Appendix A.2.

Defining typing judgements is an intricate ordeal due to heavy coupling of the system. Computations are assigned computation types, which include equations that are built using templates; these may again contain computations that need to be typed. This forces us to simultaneously define judgements for the following:

- $\Gamma \vdash v : A$, which states that in context $\Gamma$, the value $v$ has a value type $A$.

- $\Gamma \vdash c : C$, which states that in context $\Gamma$, the computation $c$ has a computation type $C$.

- $\Gamma \vdash h : \Sigma \Rightarrow D$, which states that in context $\Gamma$, the cases $h$ cover operations listed in $\Sigma$ using computations of type $D$.

- $\Gamma \vdash h : \Sigma \Rightarrow D$ respects $E$, which states that in context $\Gamma$, the cases $h$ that cover operations $\Sigma$ are well defined, meaning they handle computations equivalent under $E$ into equivalent computations of type $D$.

- $\vdash A : \text{vtype}$, which states that the value type $A$ is well formed.

- $\vdash C : \text{ctype}$, which states that the computation type $C$ is well formed.

- $\vdash \Sigma : \text{sig}$, which states that the signature $\Sigma$ is well formed.

- $\vdash \Gamma : \text{ctx}$, which states that the value context $\Gamma$ is well formed.

- $\vdash Z : \text{tctx}$, which states that the template context $Z$ is well formed.

- $\Gamma ; Z \vdash T : \Sigma$, which states that in contexts $\Gamma$ and $Z$, the template $T$ is well formed with respect to the signature $\Sigma$.  

29
• $\vdash \mathcal{E} : \Sigma$, which states that equations $\mathcal{E}$ are well formed with respect to the signature $\Sigma$.

The mutually recursive definition does not stop there. To provide proofs for respects, we need a logic, and judgements of the logic may require information about types of terms. That forces us to define relations used in the logic at the same time as the relations of the type system, linking the system even further. In fact, when defining the logic presented in Section 5.5, we simultaneously define about 15 different relations.

### 4.2.1 Well-formed types, contexts, and templates

The base value types `unit`, `int`, and `empty` are well formed by default, while judgements for constructed value types proceed structurally.

\[
\begin{align*}
\vdash \text{unit} &: \text{vtype} & \vdash \text{int} &: \text{vtype} & \vdash \text{empty} &: \text{vtype} \\
\vdash A &: \text{vtype} & \vdash C &: \text{ctype} & \vdash A \rightarrow C &: \text{vtype} & \vdash C &: \text{ctype} & \vdash D &: \text{ctype} & \vdash C \Rightarrow D &: \text{vtype} \\
\vdash A &: \text{vtype} & \vdash B &: \text{vtype} & \vdash A \times B &: \text{vtype} & \vdash A &: \text{vtype} & \vdash B &: \text{vtype} & \vdash A + B &: \text{vtype} \\
\vdash A &: \text{vtype} & \vdash A \text{list} &: \text{vtype} \\
\end{align*}
\]

A computation type is well formed if all its components are well formed. It should be noted that the well-formedness of equations is checked with respect to the signature of the type, which ensures that the components are compatible.

\[
\begin{align*}
\vdash A &: \text{vtype} & \vdash \Sigma &: \text{sig} & \vdash \mathcal{E} &: \Sigma & \vdash A | \Sigma | \mathcal{E} &: \text{ctype} \\
\end{align*}
\]

A well-formed signature also ensures that it contains precisely one type assignment per operation name, which is crucial for proofs.

\[
\begin{align*}
\vdash \{\} &: \text{sig} & \vdash \Sigma &: \text{sig} & \vdash A &: \text{vtype} & \vdash B &: \text{vtype} & \text{op} \notin \Sigma & \vdash \Sigma \cup \{\text{op} : A \rightarrow B\} &: \text{sig} \\
\end{align*}
\]

We check a context or a template context by verifying that all of the assigned types are well formed. Unlike with signatures, we impose no additional requirements on variable names. The formalisation of the system uses a (harder-to-read) nameless representation, so we are not overly troubled by variable names.

\[
\begin{align*}
\vdash \cdot &: \text{ctx} & \vdash \Gamma &: \text{ctx} & \vdash A &: \text{vtype} & \vdash \Gamma, x : A &: \text{ctx} \\
\vdash \cdot &: \text{tctx} & \vdash Z &: \text{tctx} & \vdash A &: \text{vtype} & \vdash Z, z : A \rightarrow \ast &: \text{tctx} \\
\end{align*}
\]

Templates are a lot like terms, and checking well-formedness of templates proceeds in a manner similar to typing terms. We check with respect to a signature $\Sigma$ to ensure consistency of operation types.
Template variables \( z \) need to be applied to values of correct types. Just like the name \texttt{do pure} suggests, we ensure that the computation is pure. The type system does not keep track of nontermination, and it is possible to have a nonterminating computation in the pure sequencing construct, which does not coincide with certain notions of purity.

\[
\Gamma \vdash v : A \\
(\varepsilon : A \to \ast) \in Z \\
\Gamma ; Z \vdash v : \Sigma
\]

\[\text{WfTApp}\]

\[
\Gamma \vdash c : A! / / \\
\Gamma ; x : A ; Z \vdash T ; \Sigma \\
\Gamma ; Z \vdash \text{do pure } x \leftarrow c \text{ in } T ; \Sigma
\]

\[\text{WfTDo}\]

To check template \texttt{match} statements, we ensure that the values are of correct types and that all branches are well formed and compatible with the same signature \( \Sigma \).

\[
\Gamma \vdash v : \text{empty} \\
\Gamma ; Z \vdash \text{absurd } v : \Sigma
\]

\[\text{WfTAbsurd}\]

\[
\Gamma \vdash v : A \times B \\
\Gamma , x : A , y : B ; Z \vdash T ; \Sigma \\
\Gamma ; Z \vdash \text{match } v \text{ with } (x,y) \mapsto T ; \Sigma
\]

\[\text{WfTProdMatch}\]

\[
\Gamma \vdash v : A + B \\
\Gamma , x : A ; Z \vdash T_1 ; \Sigma \\
\Gamma , y : B ; Z \vdash T_2 ; \Sigma \\
\Gamma ; Z \vdash \text{match } v \text{ with Left } x \mapsto T_1 | \text{Right } y \mapsto T_2 ; \Sigma
\]

\[\text{WfTSumMatch}\]

\[
\Gamma \vdash v : A \text{ list} \\
\Gamma ; Z \vdash T_1 ; \Sigma \\
\Gamma , x : A ; x \cdot \text{ list} ; Z \vdash T_2 ; \Sigma \\
\Gamma ; Z \vdash \text{match } v \text{ with } [] \mapsto T_1 | x :: x \cdot \text{ list } \mapsto T_2 ; \Sigma
\]

\[\text{WfTListMatch}\]

The rule for operations includes explicit subtyping. We allow the annotations of the operation to differ from the types inferred from the signature, as long as they are compatible (through subtyping). We need this formulation in order to prove Lemma 4.3.4, which links well-formedness of templates and subtyping on signatures. This is a problem specific to the combination of type annotations on operations and the choice of signature subtyping, where operation types can differ.

\[
(op : A' \to B') \in \Sigma \\
A \leq A' \\
B' \leq B \\
\Gamma \vdash v : A \\
\Gamma , y : B ; Z \vdash T ; \Sigma \\
\Gamma ; Z \vdash op_{A \to B}(v; y.T) : \Sigma
\]

\[\text{WfTop}\]

Equations are sets that contain pairs of well-formed templates. Templates are only used as parts of equations, so it is sufficient to ensure well-formedness of contexts when checking equations.

\[
\vdash \{\} : \Sigma
\]

\[\text{WfEqs}\{\}\]

\[
\vdash \Gamma : \text{ctx} \\
\vdash Z : \text{tctx} \\
\vdash \mathcal{E} : \Sigma \\
\Gamma ; Z \vdash T_1 ; \Sigma \\
\Gamma ; Z \vdash T_2 ; \Sigma \\
\vdash \mathcal{E} \cup \{\Gamma ; Z \vdash T_1 \sim T_2\} : \Sigma
\]

\[\text{WfEqsU}\]

\[\text{4.2.2 Typing values, computations, and operation cases}\]

In all of the following judgements we implicitly assume that all contexts and types are well formed. The judgements are collected in Appendix A.2.
The judgements for variables and base values are standard.

\[
\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \quad \frac{}{\Gamma \vdash (): unit} \quad \quad \frac{}{\Gamma \vdash n : int}
\]

\[
\frac{}{\Gamma \vdash \text{Left}_{A+B} \, v : A+B} \quad \quad \frac{}{\Gamma \vdash \text{Right}_{A+B} \, v : A+B}
\]

\[
\frac{}{\Gamma \vdash \text{TypePair}}
\]

\[
\frac{}{\Gamma \vdash \{} : A \, \text{list}}
\]

\[
\frac{}{\Gamma \vdash \text{TypeCons}}
\]

Typing of functions also features no novelties. The judgement for handlers now requires a proof for the \textit{respects} relation, which ensures that the handlers are well defined with respect to the effect theory. The definition of \textit{respects} is delayed to Chapter 5; it is the only time the type system requires a proof from the logic.

\[
\frac{}{\Gamma, x : A \vdash c : C}
\]

\[
\frac{}{\Gamma \vdash \text{fun} \, (x : A) \mapsto c : A \rightarrow C}
\]

\[
\frac{}{\Gamma \vdash \text{handler} \, (\text{ret} \, (x : A) \mapsto c_r, h) : A!\Sigma/E \Rightarrow D}
\]

A value return has no possible operation calls and is therefore assigned the empty theory. For recursive functions, we need to include the variable \(f\) in the context when checking \(c_1\) to allow recursive calls.

\[
\frac{}{\Gamma \vdash \text{TypeRet}}
\]

\[
\frac{}{\Gamma \vdash \text{TypeApp}}
\]

\[
\frac{}{\Gamma \vdash \text{let rec} \, f \, x : A \rightarrow C = c_1 \, \text{in} \, c_2 : D}
\]

In sequencing we ensure that both computations use the same local theory, though the value types might differ. The handling construct is unchanged from previous approaches, but it now guarantees that only correct handlers can handle computations using theories.

\[
\frac{}{\Gamma \vdash \text{TypeDo}}
\]

\[
\frac{}{\Gamma \vdash \text{with} \, v \, \text{handle} \, c : D}
\]

The judgement for operations is a bit non-standard in the sense that we allow annotations to differ from the types in the signature. This formulation of the typing judgement
is necessary for the proof of language safety in Subsection 4.3.2. A more in-depth analysis is provided in the proof of Theorem 4.3.6.

\[
(\text{op} : A'_{\text{op}} \to B'_{\text{op}}) \in \Sigma \\
\begin{array}{ll}
A_{\text{op}} \leq A'_{\text{op}} & B'_{\text{op}} \leq B_{\text{op}} \\
\end{array}
\begin{array}{c}
\Gamma \vdash v : A_{\text{op}} \\
\Gamma, y : B_{\text{op}} \vdash c : A!\Sigma/\mathcal{E}
\end{array}
\quad \infer[\text{TypeOp}]{\Gamma \vdash \text{op}_{A_{\text{op}} \to B_{\text{op}}} (v; y.c) : A!\Sigma/\mathcal{E}}
\]

Similarly to WFTOP, the additional premises are required due to the combination of type annotations on operation calls and the stronger notion of signature subtyping.

Match statements must ensure that all of their branches have the same type. The type we assign to \text{absurd} comes from the type annotation to avoid ambiguity.

\[
\begin{array}{l}
\Gamma \vdash v : \text{empty} \\
\Gamma \vdash \text{absurd}_C v : C
\end{array}
\quad \infer[\text{TypeAbsurd}]{\Gamma \vdash v : A \times B}
\quad \infer[\text{TypeProdMatch}]{\Gamma \vdash \text{match} v \text{ with } (x,y) \mapsto c : C}
\]

\[
\begin{array}{l}
\Gamma \vdash v : A + B \\
\Gamma, x : A \vdash c_1 : C \\
\Gamma, y : B \vdash c_2 : C
\end{array}
\quad \infer[\text{TypeSumMatch}]{\Gamma \vdash \text{match} v \text{ with } \text{Left } x \mapsto c_1 \mid \text{Right } y \mapsto c_2 : C}
\]

\[
\begin{array}{l}
\Gamma \vdash v : A \text{ list} \\
\Gamma \vdash c_1 : C \\
\Gamma, x : A, xs : A \text{ list} \vdash c_2 : C
\end{array}
\quad \infer[\text{TypeListMatch}]{\Gamma \vdash \text{match} v \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 : C}
\]

Checking that operation cases are well typed is done in a structural way that jointly reduces the cases and the signature. This means that our handler implements cases precisely for the operations in the signature, resulting in closed handlers. Just as with \text{absurd} we follow the type annotation when typing an empty set of operation cases.

\[
\infer[\text{TypeCases}]{\Gamma \vdash \{ \} : D \Rightarrow D}
\]

\[
\begin{array}{l}
\Gamma \vdash h : \Sigma \Rightarrow D \\
\Gamma, x : A, k : B \vdash D \vdash c_{\text{op}} : D
\end{array}
\quad \infer[\text{TypeCasesU}]{\Gamma \vdash h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{\text{op}} \} : (\Sigma \cup \{ \text{op} : A \rightarrow B \}) \Rightarrow D}
\]

Due to the signature \Sigma being well formed, and thus only containing one type assignment per operation, well-typed operation cases contain only one case per operation. Considering that the rule for operation calls \text{TypeOp} features explicit subtyping, it might seem interesting that \text{TypeCasesU} requires perfectly matching annotations. This follows from the fact that we use subtyping only for computations, but not for operation cases (explained below).

To provide a way to use subtyping, we add two additional subsumption judgements, which allow us to transition to a less specific type.

\[
\infer[\text{TypeVSubsume}]{\Gamma \vdash v : A \leq A'}
\quad \infer[\text{TypeCSSubsume}]{\Gamma \vdash c : C \leq C'}
\]

We do not add rules for subtyping operation cases. In the rule \text{TypeCasesU} the type \( D \) is both in a covariant and contravariant (in the context) position, which effectively blocks any subtyping attempts for \( D \). Because operation cases can only be used inside a handler construct, we recover the subtyping functionality through subsumption for values. Assume we have cases \( h \) of type \( \{ \text{op}_1, \text{op}_2 \} \Rightarrow D \) but need a handler that handles only \( \text{op}_1 \). We have no way to assign \( h \) the type \( \{ \text{op}_1 \} \Rightarrow D \), which prevents us from directly using \( h \) in a handler of type \( A!\{ \text{op}_1 \}/\mathcal{E} \Rightarrow D \). Instead, we give \( h \) the type \( \{ \text{op}_1, \text{op}_2 \} \Rightarrow D \) and construct a handler of type \( A!\{ \text{op}_1, \text{op}_2 \}/\mathcal{E} \Rightarrow D \). The type of the handler is then corrected with \text{TypeVSubsume} to \( A!\{ \text{op}_1 \}/\mathcal{E} \Rightarrow D \).
4.3 Properties of the type system

As has been stated throughout this chapter, the type system is tightly interlinked, which leads to proofs being linked as well. With logic forming a part of the type system, most lemmas need to also be stated for logic judgements and proven simultaneously. However, in Chapter 5 we present multiple suitable logics, and parts of proofs that pertain only to types are not affected by the choice of logic. In order to state properties at a relevant time, we present lemmas for typing judgements here and revisit them in Chapter 5 to finalize the proofs.

Because the language is fairly rich, we avoid writing out full proofs and instead focus on more involved cases. The full proofs are instead provided in the formalisation.

4.3.1 Properties of subtyping

Lemma 4.3.1 (Reflexivity of subtyping).
- For a well-formed value type \( \vdash A : \text{vtype} \), it holds that \( A \leq A \).
- For a well-formed computation type \( \vdash C : \text{ctype} \), it holds that \( C \leq C \).
- For a well-formed signature \( \vdash \Sigma : \text{sig} \), it holds that \( \Sigma \leq \Sigma \).
- For a well-formed theory \( \vdash \Sigma : \text{sig} \), it holds that \( \Sigma \leq \Sigma \).
- For a well-formed context \( \vdash \Gamma : \text{ctx} \), it holds that \( \Gamma \leq \Gamma \).

\( \text{Proof (formalised).} \) We proceed by induction on type structure. The proof for value and computation types is entirely structural, and so is the proof for contexts. We encounter a slight obstacle in the case of signatures.

When showing that \( \Sigma \cup \{op : A \rightarrow B\} \leq \Sigma \cup \{op : A \rightarrow B\} \), we have the induction hypothesis \( \Sigma \leq \Sigma \), but the rule for subtyping signatures requires \( \Sigma \leq \Sigma \cup \{op : A \rightarrow B\} \).

This can be shown if \( op \) is not present in \( \Sigma \) with a different type. For this reason, we require well-formedness of signatures (and in turn for all other types in the lemma), since it provides the necessary uniqueness. The proof for equations is similar, but without the obstacle of uniqueness. \( \square \)

Lemma 4.3.2 (Transitivity of subtyping).
- For value types, if \( A_1 \leq A_2 \) and \( A_2 \leq A_3 \), then also \( A_1 \leq A_3 \).
- For computation types, if \( C_1 \leq C_2 \) and \( C_2 \leq C_3 \), then also \( C_1 \leq C_3 \).
- For signatures, if \( \Sigma_1 \leq \Sigma_2 \) and \( \Sigma_2 \leq \Sigma_3 \), then also \( \Sigma_1 \leq \Sigma_3 \).
- For equations, if \( E_1 \subseteq E_2 \) and \( E_2 \subseteq E_3 \), then also \( E_1 \subseteq E_3 \).
- For contexts, if \( \Gamma_1 \leq \Gamma_2 \) and \( \Gamma_2 \leq \Gamma_3 \), then also \( \Gamma_1 \leq \Gamma_3 \).

\( \text{Proof (formalised).} \) For value types, we start by induction on the derivation of subtyping for \( A_1 \leq A_2 \). In all cases we assert that we must have arrived at \( A_2 \leq A_3 \) using the same rule, due to the shape of \( A_2 \). As an example, in the case of \( A_1 \leq A_2 \) being the conclusion of \( \text{STyFun} \), we know that \( A_1 \) is of form \( B_1 \rightarrow C_1 \) and \( A_2 \) of form \( B_2 \rightarrow C_2 \). The only way to arrive at \( B_2 \rightarrow C_2 \leq A_3 \) is with \( \text{STyFun} \), so \( A_3 \) is of form \( B_3 \rightarrow C_3 \). We then obtain by induction that \( B_3 \leq B_1 \) and \( C_1 \leq C_3 \), and conclude that \( B_1 \rightarrow C_1 \leq B_3 \rightarrow C_3 \), completing the case.

This approach works for other cases as well. \( \square \)

Lemma 4.3.3 (Context subsumption). Assume we have two well-typed contexts for which \( \Gamma' \leq \Gamma \) holds.
- If \( \Gamma \vdash v : A \) holds, then we can show \( \Gamma' \vdash v : A \).
- If \( \Gamma \vdash c : C \) holds, then we can show \( \Gamma' \vdash c : C \).
If $\Gamma \vdash h : \Sigma \Rightarrow D$ holds, then we can show $\Gamma' \vdash h : \Sigma \Rightarrow D$.

If $\Gamma \vdash h : \Sigma \Rightarrow D$ respects $E$ holds, then we can show $\Gamma' \vdash h : \Sigma \Rightarrow D$ respects $E$.

Similar properties must hold for any logic judgements.

**Proof (formalised).** The proof is straightforward induction on derivation of typing judgements. The only interesting case is the one of variables. Assume we have $\Gamma \vdash x : A$, which implies that $x : A \in \Gamma$. By definition of context subtyping we know that $x : A' \in \Gamma'$ for some $A'$, so we can show $\Gamma' \vdash x : A'$. We also know that $A' \leq A$, so we can use $\text{TypeVSubsume}$.

This results in the desired conclusion $\Gamma' \vdash x : A$. Note that we have changed the names of types in $\text{TypeVSubsume}$ to fit our situation. □

While we added subsumption rules for values and computations, it is helpful to also assess that subtyping interacts nicely with well-formedness of equations.

**Lemma 4.3.4.** Suppose that $\Sigma \leq \Sigma'$ and that we can show $\vdash E : \Sigma$. Then we can also show that $\vdash E : \Sigma'$ holds.

**Proof (formalised).** To show the lemma, we need to prove a similar property for templates, where $\Gamma ; Z \vdash T : \Sigma$ entails $\Gamma ; Z \vdash T : \Sigma'$. This is done by induction on the derivation of well-formedness of templates, which is simple with Lemma 4.3.3. In the case of $\text{op}$, we make use of explicit subtyping for annotations in $\text{WFTop}$. The types of operations in $\Sigma'$ are different to those in $\Sigma$, but annotations in $T$ are fixed, so explicit subtyping is required. □

### 4.3.2 Safety

To show that the proposed operational semantics and type system interact correctly, we state and prove a safety theorem. The operational semantics heavily relies on substitution, so it is important to first consider the interaction between substitution and types. The substitution lemma is by no means trivial and represents a significant part of the formalisation effort. In fact, 7000 out of the 11000 lines of the formalisation are dedicated to substitution. We again only state part of the lemma and delay the treatment of substitution in logic for later chapters.

**Lemma 4.3.5** (Substitution). We use $(\Gamma_1, x : B, \Gamma_2)$ to denote a context which consists of $\Gamma_1$ followed by the assignment $x : B$, which is then followed by assignments of $\Gamma_2$. Assume that we have a well-typed value $(\Gamma_1, \Gamma_2) \vdash v' : B$.

- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash v : A$, then we can show

  $$(\Gamma_1, \Gamma_2) \vdash v[x \mapsto v'] : A.$$ 

- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash c : C$, then we can show

  $$(\Gamma_1, \Gamma_2) \vdash c[x \mapsto v'] : C.$$ 

- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash h : \Sigma \Rightarrow D$, then we can show

  $$(\Gamma_1, \Gamma_2) \vdash h[x \mapsto v'] : \Sigma \Rightarrow D.$$ 

1https://github.com/zigaLuksic/eEFF-formalization

35
If we have \((\Gamma_1, x : B, \Gamma_2) \vdash h : \Sigma \rightharpoonup \rightharpoonup D\) respects \(E\), then we can show \((\Gamma_1, \Gamma_2) \vdash h[x \mapsto v'] : \Sigma \rightharpoonup \rightharpoonup D\) respects \(E\).

Proof (formalised). The proof proceeds by induction on typing derivation. We omit the details as they are specific to the nameless style of the formalisation. □

The safety theorem guarantees that evaluation does not change the type of our program (preservation) and that evaluation can always continue until the program either returns a value or calls an operation (progress).

Theorem 4.3.6 (Safety).

**Preservation** If \(\cdot \vdash c : C\) and \(c \leadsto c'\), then \(\cdot \vdash c' : C\).

**Progress** If \(\cdot \vdash c : A!\Sigma/E\), then either

\(\diamond\) there exists a computation \(c'\) such that \(c \leadsto c'\);

\(\diamond\) \(c\) is of the form \(\text{ret } v\) for some value \(v\);

\(\diamond\) \(c\) is of the form \(\text{op } A \rightarrow B(v; k)\) for some \(\text{op} \in \Sigma\).

Proof (formalised).

**Preservation** is proven by induction on the step relation \(c \leadsto c'\). For example, in \(\text{APPFUN}\) we have \((\text{fun } (x : A) \mapsto \text{c}) v \leadsto \text{c}[x \mapsto v]\) and by assumption we know that \(\cdot \vdash (\text{fun } (x : A) \mapsto \text{c}) v : C\). We observe that the function is of type \(A \rightarrow C\) for some \(A\), by analysing possible proof derivations. This is made more difficult in the presence of subtyping and requires a short (but not interesting) proof by induction. It follows that \(x : A \vdash \text{c} : C\) and \(\cdot \vdash v : A\). By using the substitution lemma, we conclude that \(\cdot \vdash \text{c}[x \mapsto v] : C\). Other cases proceed similarly.

When dealing with \(\text{DOOP}\), we also observe the requirement for explicit subtyping in \(\text{TYPEOP}\).

\[
\text{do } x \leftarrow \text{op}_{A \rightarrow B}(v; y, c_1) \text{ in } c_2 \leadsto \text{op}_{A \rightarrow B}(v; y, \text{do } x \leftarrow c_1 \text{ in } c_2)
\]

Assume that \(\text{TYPEOP}\) does not feature explicit subtyping for operation annotations; in that case annotations must match the signature perfectly. Now consider the example

\[
\text{do } x \leftarrow \text{op}_{A \rightarrow B}(a; y.\text{ret } y) \text{ in } \text{op}_{A' \rightarrow B'}(a'; y'.\text{ret } y').
\]

where \(A \leq A'\) and \(B' \leq B\). Here \(\text{op}\) is just a name, so we have a case of the same operation with two different type annotations. The above can be typed at a signature \(\{\text{op} : A' \rightarrow B'\}\) if we use \(\text{TYPECSUBSUME}\) to change the signature of the first computation in the sequence. After applying \(\text{DOOP}\) we end up with

\[
\text{op}_{A \rightarrow B}(a; y.\text{do } x \leftarrow \text{ret } y \text{ in } \text{op}_{A' \rightarrow B'}(a; y'.\text{ret } y')).
\]

This can no longer be typed without subtyping as part of \(\text{TYPEOP}\). To type the first operation call, we need to reduce the signature through subsumption, but that means that the continuation needs to type at the reduced signature. This is prevented by the type annotations of the second call. If we include explicit subtyping for the type annotations, we no longer need to reduce the signature just to match the annotations, and the problem is resolved. It is perhaps interesting that only annotations on operations are problematic (just like in Lemma 4.3.4).
Progress can be shown either by induction on the type derivation or by induction on term structure. The cases of \texttt{ret} and \texttt{op} are trivial by formulation of progress. For most cases, we only need to establish the shape of certain terms. For instance, in the case of a \texttt{match} statement for lists, we have evaluation rules for when the value is an empty list or a constructed list. Because variables cannot be typed in an empty context, a value of type \texttt{A list} must either be an empty list or a constructed list, and both options have a possible step. The more interesting cases are those pertaining to sequencing and handling. In the case of sequencing, we have \( \cdot \downarrow \texttt{do } x \leftarrow c_1 \texttt{ in } c_2 : C \), where we use the induction hypothesis on \( c_1 \) (which is well typed). We can then use the appropriate evaluation rule, depending on whether \( c_1 \leadsto c'_1, c_1 = \texttt{ret} v \), or \( c_1 = \texttt{op}_{A \to B}(v; y.c) \). The case for handling is dispatched similarly to sequencing, but we additionally rely on the type of cases to guarantee that an appropriate handler case exists.

4.4 Templates to computations

As the name implies, templates can be used to construct computations of certain shapes. Consider the (sugared) template

\[
T_{\text{example}} := \text{RandomInt}(); \text{y.if } y > 0 \text{ then } z_1 \text{ y else } z_2 (\text{ })).
\]

It seems that the computation

\[
\text{RandomInt}(); \text{y.if } y > 0 \text{ then (ret (Some y)) else (ret None))}
\]

has the shape specified by \( T_{\text{example}} \). By adjusting computations in branches to function applications, we can rewrite the example to almost exactly match the shape of the template. The only difference is the two functions in place of template variables \( z_1 \) and \( z_2 \).

\[
\text{RandomInt}(); \text{y.if } y > 0 \text{ then (fun } x \mapsto \text{ret (Some x)) y else (fun } _x \mapsto \text{ret None)) (})
\]

The notation of EEFF draws little distinction between templates and ordinary language constructs, although they should be treated differently. We provide an instantiation procedure \( I \) that translates a template to a computation of type \( C \).

\[
(z_k \ n)^I_C = z_k \ n \quad (z_k \ becomes \ ordinary \ variable)
\]

\[
\text{(do pure } x \leftarrow c \text{ in } T^I_C = \text{do } x \leftarrow c \text{ in } T^I_C
\]

\[
op_{A \to B}(v; y.T^I_C) = \text{op}_{A \to B}(v; y.T^I_C)
\]

\[
(\text{absurd } v)^I_C = \text{absurd}_C v
\]

\[
(\text{match } v \text{ with } (x,y) \mapsto T^I_C = \text{match } v \text{ with } (x,y) \mapsto T^I_C
\]

\[
(\text{match } v \text{ with } [ ] \mapsto T_1 T^I_C | x :: xs \mapsto T_2 T^I_C = \text{match } v \text{ with } [ ] \mapsto T_1 T^I_C | x :: xs \mapsto T_2 T^I_C
\]

\[
(\text{match } v \text{ with Left } x \mapsto T_1 | \text{Right } y \mapsto T_2) T^I_C = \text{match } v \text{ with Left } x \mapsto T_1 T^I_C | \text{Right } y \mapsto T_2 T^I_C
\]

The second step of instantiation is to replace variables with values (which can also be variables). For a template \( (x_i)_i : (z_j)_j \vdash T : \Sigma \), we use a parallel substitution that we denote with \( [(x_i \mapsto v_i)_i, (z_j \mapsto u_j)_j] \). The substitution in combination with the instantiation replaces variables \( x_i \) with values \( v_i \) and template variables \( z_j \) with function values \( u_j \).
The previously discussed example can be obtained from $T_{\text{example}}$ by

$$T_{\text{example}}[z_1 \mapsto (\text{fun } x \mapsto \text{Some } x), z_2 \mapsto (\text{fun } _\_ \mapsto \text{None})]$$

If our type system is sensible, we expect that instantiating well-formed templates yields well-typed computations. This also shows how the wildcard type $*$ effectively becomes $A!\Sigma / E$ through instantiation.

**Lemma 4.4.1.** Assume we have well-formed contexts $\vdash \Gamma : \text{ctx}$ and $\vdash (z_i : A_i \rightarrow *)_i : \text{tctx}$, and a well-formed computation type $\vdash A!\Sigma / E : \text{ctype}$. For $\Gamma, (z_i : A_i \rightarrow *)_i \vdash T : \Sigma$, we can show that the instantiated template is well typed as a term

$$\Gamma, (z_i : A_i \rightarrow A!\Sigma / E)_i \vdash T_{A!\Sigma / E}^I : A!\Sigma / E$$

**Proof (formalised).** The proof proceeds by induction on template well-formedness. The only interesting part is the translation of pure sequencing, where we need to use the subsumption rule to lift the pure computation to a computation of type $A!\Sigma / E$. This shows that the only option, aside from pure sequencing, is to allow computations with signature $\Sigma$, but even that becomes problematic later on. □

The above Lemma only covers the translation part of instantiation and does not account for substitutions. But thanks to substitution lemmas (which also hold for parallel substitutions), we can generalise the result to arbitrary instantiations.
The effect-theory system of \texttt{EEFF} must be equipped with a logic in which we prove that handlers respect the effect theory. The choice of logic does not directly impact the typing judgements and is only linked to the type system through the \texttt{respects} relation. In this chapter, we present five logics of increasing complexity that we consider interesting or viable.

\section{Empty logic}

The empty logic provides no rules. This means there is no way to provide a proof for $\Gamma \vdash \Sigma : D \Rightarrow h \text{ respects } \mathcal{E}$, meaning that there is no way to successfully type a handler. We end up in an extension of the $\lambda$-calculus, where effects only act as signals that terminate the evaluation with no way to recover. This results in a primitive notion of algebraic effects, making it little more than an interesting edge case, unless some effects have predefined behaviour.

\section{Free logic}

The free logic contains a single rule that allows us to state that operation cases always respect an empty set of equations.

$$\Gamma \vdash \Sigma : D \Rightarrow h \text{ respects } \emptyset$$

This results in a language very similar to Eff, where equations are ignored. Handlers have types of shape $A!\Sigma/\emptyset \Rightarrow D$, and in order to use handlers, we cannot assume any equations in the types of handled computations. This shows that the approach of local effect theories subsumes the approach with no equations. In every logic where such a judgement is admissible, we obtain backwards compatibility, meaning that a program written for a language with no equations can be typed.

\section{Full logic}

On the other side of the spectrum is the full logic, which states that any handler respects every set of equations.

$$\Gamma \vdash \Sigma : D \Rightarrow h \text{ respects } \mathcal{E}$$
Such a logic has little use as a reasoning tool. In fact, the full logic is an example of an *unsound logic*, explained in Chapter 6 and shown in Proposition 7.3.3. It should not, however, be dismissed as entirely useless. The full logic assumes handler correctness without proof; but unlike the free logic, handler types can include equations in the full logic. The type system can provide help with tracking theories and ensuring compatibility of building blocks, but without guarantee that handlers are written correctly. This can be seen as "leaving correctness to the user" while providing other benefits. We use it in Chapter 8 to construct an inference algorithm.

### 5.4 Equational logic

The goal is to construct a logic that provides a way to *inherit* equations from types into the logic. The logic is kept simple, so we delay the discussion of mechanisms such as hypotheses and quantifiers to Section 5.5. Three new kinds of judgement are added to the mutually recursive definition from Chapter 4.

- \( \Gamma \vdash v_1 \equiv_A v_2 \) states that values \( v_1 \) and \( v_2 \) are considered equal at value type \( A \) in the context \( \Gamma \).
- \( \Gamma \vdash c_1 \equiv_C c_2 \) states that computations \( c_1 \) and \( c_2 \) are considered equal at computation type \( C \) in the context \( \Gamma \).
- \( \Gamma \vdash h_1 \equiv_{\Sigma\Rightarrow D} h_2 \) states that operation cases \( h_1 \) and \( h_2 \) are considered equal in the context \( \Gamma \) when handling operations from \( \Sigma \) to computations of type \( D \).

The rules are presented throughout this section and collected in Appendix A.3.

**Side conditions**

Just as with typing judgements there are requirements of well-formedness of types and context in the rules of the logic. The requirements are included as implicit side conditions in all rules.

- In \( \Gamma \vdash v_1 \equiv_A v_2 \) we assume that \( \Gamma \vdash v_1 : A \) and \( \Gamma \vdash v_2 : A \).
- In \( \Gamma \vdash c_1 \equiv_C c_2 \) we assume that \( \Gamma \vdash c_1 : C \) and \( \Gamma \vdash c_2 : C \).
- In \( \Gamma \vdash h_1 \equiv_{\Sigma\Rightarrow D} h_2 \) we assume that \( \Sigma \) is well formed and that there are signatures \( \Sigma_1 \) and \( \Sigma_2 \) with \( \Sigma \leq \Sigma_1 \) and \( \Sigma \leq \Sigma_2 \), for which \( \Gamma \vdash h_1 : \Sigma_1 \Rightarrow D \) and \( \Gamma \vdash h_2 : \Sigma_2 \Rightarrow D \).
- In \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( \mathcal{E} \) we assume that \( \Gamma \vdash h : \Sigma \Rightarrow D \) and that \( \mathcal{E} \) is well formed with regard to \( \Sigma \).

The implicit typing requirements also ensure well-formedness of the contexts and types. For operation cases, we also require that \( \Sigma \) is well formed, as it is not part of any required typing judgement. The more complex side conditions for operation cases are very nonstandard and best explained with an example.

\[
h_1 = \{ \operatorname{Ping} x k \mapsto k (\), \\
\quad \operatorname{Pong} x k \mapsto k 5 \} \\
h_2 = \{ \operatorname{Ping} x k \mapsto k () \}
\]

The effect cases \( h_1 \) and \( h_2 \) are obviously not equal, since they do not even handle the same set of operations. But if we use handlers \( H_1 := \operatorname{handler} (x \mapsto \operatorname{ret} x; h_1) \) and \( H_2 := \operatorname{handler} (x \mapsto \operatorname{ret} x; h_2) \) on a computation of type \( A!\{\operatorname{Ping}\}/\{\} \), their behaviour is the same. We therefore wish to consider them equal at \( \{\operatorname{Ping}\} \Rightarrow D \), but typing operation
cases requires very precise types, hence the subtyping. To avoid such side conditions, we either require a suitable notion of subtyping on handler cases or, we need to be content with a weaker notion of handler equality.

**Basic rules**

The logic equalities should be **reflexive**, **symmetric**, and **transitive**. In equational logic we can achieve all three while requiring very little.

\[
\begin{align*}
\Gamma \vdash v \equiv_A v' & \quad \text{VEQSYM} \\
\Gamma \vdash v' \equiv_A v & \\
\Gamma \vdash c \equiv_C c' & \quad \text{CEQSYM} \\
\Gamma \vdash c' \equiv_C c & \\
\Gamma \vdash v_1 \equiv_A v_2 & \quad \text{VEQTRANS} \\
\Gamma \vdash v_2 \equiv_A v_3 & \\
\Gamma \vdash v_1 \equiv_A v_3 & \\
\Gamma \vdash c_1 \equiv_C c_2 & \quad \text{CEQTRANS} \\
\Gamma \vdash c_2 \equiv_C c_3 & \\
\Gamma \vdash c_1 \equiv_C c_3 &
\end{align*}
\]

Equational logic contains a small set of rules that closely follow the structure of terms, so we are able to prove the remaining properties within the logic. Reflexivity follows from the inclusion of structural rules (Lemma 5.4.5), with symmetry and transitivity on operation cases being inherited from properties of computations (Lemma 5.4.6). We only add the minimal set of rules in order to avoid mistakes and to obtain better insight into the system.

**Structural rules**

Structural rules assert that terms constructed from equal subterms are considered equal. There are no dedicated subsumption rules in the presented logic, but we wish for them to be admissible from term structure (see Lemma 5.4.8). For that reason we add subtyping directly into the structural rules, but only when necessary. Perhaps surprisingly, this only pertains to \text{VEQVAR}, \text{VEQHANDLER}, and OOTB (introduced in Subsection 5.4.1).

When comparing terms, their type annotations need not match. The side condition that the terms must be well typed ensures that annotations are compatible with the type at which equality is considered. When equating handlers, we need proofs that at least one of the handlers is correct at the exact types. Operation cases are problematic for subtyping, even in the respects judgement.

\[
\begin{align*}
x : A' \in \Gamma & \quad A' \leq A & \text{VEQVAR} \\
\Gamma \vdash x \equiv_A x & & \text{VEQUNIT} \\
\Gamma \vdash () \equiv_{\text{unit}} () & & \text{VEQINT} \\
\Gamma \vdash v_1 \equiv_A v_1' & \quad \Gamma \vdash v_2 \equiv_B v_2' & \text{VEQPAIR} \\
\Gamma \vdash (v_1, v_2) \equiv_{A \times B} (v_1', v_2') & \\
\Gamma \vdash v \equiv_A v' & \text{VEQLEFT} \\
\Gamma \vdash \text{Left}_{A_1 + B_1} v \equiv_{A + B} \text{Left}_{A_2 + B_2} v' & \\
\Gamma \vdash v \equiv_B v' & \text{VEQRIGHT} \\
\Gamma \vdash \text{Right}_{A_1 + B_1} v \equiv_{A + B} \text{Right}_{A_2 + B_2} v' &
\end{align*}
\]
The implicit requirements are:

\[ \Gamma \vdash [ ]_A \equiv [ ]_B \]
\[ \Gamma \vdash \nu \equiv_A \nu' \]
\[ \Gamma \vdash \nu : \nu \equiv : \nu \equiv_A \nu' \]
\[ \Gamma \vdash [ ]_A \equiv [ ]_B \]
\[ \Gamma \vdash \nu \equiv \nu' \]

To show how side conditions help with type annotations, consider \( \text{VeqLeft} \).

\[ \Gamma \vdash \nu \equiv_A \nu' \]
\[ \Gamma \vdash \text{Left}_{A_1 + B_1} \nu \equiv_{A + B} \text{Left}_{A_2 + B_2} \nu' \]

The implicit requirements are \( (\Gamma \vdash \text{Left}_{A_1 + B_1} \nu : A + B) \) and \( (\Gamma \vdash \text{Left}_{A_2 + B_2} \nu : A + B) \). This can only hold if \( A_1 \leq A \) and \( B_1 \leq B \), and similarly for \( A_2 \) and \( B_2 \). From typing rules, it follows that \( \Gamma \vdash \nu : A_1 \) and \( \Gamma \vdash \nu' : A_2 \). By subsumption \( \nu \) and \( \nu' \) also have type \( A \), so the requirement \( \Gamma \vdash \nu \equiv_A \nu' \) makes sense.

Structural rules for computations carry no surprises. For \textit{absurd}, we do not check that \( \nu \) and \( \nu' \) are equal at \textit{empty}, because we treat all non-existent values as equal. Unlike other type annotations for values and computations, the annotations for recursively defined functions need to match precisely. The types in the annotations of \textit{let rec} appear in covariant and contravariant position, so we have no room for subtyping.

\[ \Gamma \vdash \nu \equiv_A \nu' \]
\[ \Gamma \vdash \text{ret} \nu \equiv_{A(\Sigma/E)} \text{ret} \nu' \]

\[ \Gamma \vdash \text{Left}_{A_1 + B_1} \nu \equiv_{A + B} \text{Left}_{A_2 + B_2} \nu' \]

42
The rules mirror the small-step semantics, and thanks to transitivity of logic equality, this sets, subsume the expected structural rules. The case where both operation cases are empty structure of the signature as opposed to the structure of the term. Luckily, the above rules follow from \( \Gamma \vdash \text{match } h \equiv \{ \} \). The judgements follow the structure of the signature \( \Sigma \).

\[
\Gamma \vdash h \equiv_{\{\}} \downarrow D \quad \text{HeqSig\{\}}
\]

\[
(\text{op}_{A_1 \rightarrow B_1}(x; k) \mapsto c_{op}) \in h \quad (\text{op}_{A_2 \rightarrow B_2}(x; k) \mapsto c'_{op}) \in h' \\
\Gamma, x : A, k : B \rightarrow D \vdash c_{op} \equiv_D c'_{op} \quad \Gamma + h \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} D \quad \text{HeqSig\{\}}
\]

These rules cover more use cases, but it is sometimes problematic that they follow the structure of the signature as opposed to the structure of the term. Luckily, the above rules subsume the expected structural rules. The case where both operation cases are empty sets, \( \Gamma + \{\} \equiv_{\{\}} \downarrow D \{\} \), follows from HeqSig\{\}.

**Lemma 5.4.1.** Suppose operation cases \( h \) and \( h' \) do not contain a case for the operation \( \text{op} \). If we can show \( \Gamma + h \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} D \) \( h' \) and \( \Gamma, x : A, k : B \rightarrow D \vdash c_{op} \equiv_D c'_{op} \), then we can also show

\[
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \cup \{\text{op}: A \rightarrow B\}} \downarrow D \quad h' \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]

Proof (formalised). The proof is made easier by first proving a weaker version, in which we do not extend the signature in the conclusion.

\[
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \quad h' \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]

We prove this by straightforward induction on the structure of \( \Sigma \). The original lemma then follows from HeqSig\{\}.

### \( \beta \) reductions and \( \eta \) expansions

To equate terms at different stages of evaluation, we extend the logic with \( \beta \)-reduction rules. The rules mirror the small-step semantics, and thanks to transitivity of logic equality, this

\[
\Gamma \vdash v \equiv_{A \times B} v' \\
\Gamma, x : A, y : B \vdash c \equiv c'
\]

\[
\Gamma \vdash \text{match } v \text{ with } (x, y) \mapsto c \equiv \{ \} \text{ match } v' \text{ with } (x, y) \mapsto c'
\]

\[
\Gamma \vdash v \equiv_{A + B} v' \\
\Gamma, x : A \vdash c_1 \equiv c'_1 \\
\Gamma, y : B \vdash c_2 \equiv c'_2
\]

\[
\Gamma \vdash \text{match } v \text{ with } \text{Left } x \mapsto c_1 | \text{Right } y \mapsto c_2 \\
\equiv \{ \} \text{ match } v' \text{ with } \text{Left } x \mapsto c'_1 | \text{Right } y \mapsto c'_2
\]

\[
\Gamma \vdash v \equiv_{\text{A list}} v' \\
\Gamma \vdash c_1 \equiv c'_1 \\
\Gamma, x : A, xs : \text{A list} \vdash c_2 \equiv c'_2
\]

\[
\Gamma \vdash \text{match } v \text{ with } [ ] \mapsto c_1 | x :: xs \mapsto c_2 \\
\equiv \{ \} \text{ match } v' \text{ with } [ ] \mapsto c'_1 | x :: xs \mapsto c'_2
\]

We always consider terms equal up to renaming of bound variables, so there is no distinction between \( \text{fun } (x : A) \mapsto \text{ret } x \) and \( \text{fun } (y : A) \mapsto \text{ret } y \).

**Rules for operation cases**

We provide a more general approach when dealing with operation cases, rather than just comparing their structure. Operation cases \( h \) and \( h' \) are equal at \( \Sigma \Rightarrow D \) if for any operation \( \text{op} \in \Sigma \), the case for \( \text{op} \) in \( h \) is equal to the case for \( \text{op} \) in \( h' \). The judgements follow the structure of the signature \( \Sigma \).

\[
\Gamma \vdash h \equiv_{\{\}} \downarrow D \quad \text{HeqSig\{\}}
\]

\[
(\text{op}_{A_1 \rightarrow B_1}(x; k) \mapsto c_{op}) \in h \\
(\text{op}_{A_2 \rightarrow B_2}(x; k) \mapsto c'_{op}) \in h' \\
\Gamma \vdash h \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} D \quad \Gamma + h \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} D \\
\Gamma \vdash h \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \quad h'
\]

\[
\Gamma \vdash h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \\
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]

\[
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \\
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]

\[
\Gamma \vdash h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \\
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]

We prove this by straightforward induction on the structure of \( \Sigma \). The original lemma then follows from HeqSig\{\}.

\[
\Gamma \vdash h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \equiv_{\Sigma \uplus \{\text{op}: A \rightarrow B\}} \downarrow D \\
\Gamma + h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c'_{op} \}
\]
and using the induction hypothesis for subterms.

The importance of structural rules and considering cases with a directly applicable structure of terms, which might help us by enabling further reductions. The importance of structural rules and considering cases with a directly applicable structure of terms is crucial.

The side condition of a well-typed computation directly follows from the safety Theorem 4.3.6. The proof proceeds by induction on structure of a well-typed directly follows from the safety Theorem 4.3.6. The proof proceeds by induction on structure of a well-typed computation.

Such operational judgements can be modelled by using structural rules combined with \( \beta \)-rules on subterms.

\[
\begin{align*}
\Gamma \vdash \text{do } x \leftarrow \text{ret } v \text{ in } c & \equiv c \left[ x \mapsto v \right] & \beta_{\text{DoRet}} \\
\Gamma \vdash \text{do } x \leftarrow \text{op}_{A \rightarrow B}(v; y, c_1) \text{ in } c_2 & \equiv c \left[ \text{op}_{A \rightarrow B}(v; y, \text{do } x \leftarrow c_1 \text{ in } c_2) \right] & \beta_{\text{DoOp}} \\
\Gamma \vdash \text{(fun } (x : A) \mapsto c) \; v & \equiv c \left[ x \mapsto v \right] & \beta_{\text{App}} \\
\Gamma \vdash \text{let rec } f \; x : A \rightarrow C = c_1 \text{ in } c_2 & \equiv_D c_2[f \mapsto (\text{fun } y : A \mapsto \text{let rec } f \; x : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])] & \beta_{\text{LetRec}} \\
\Gamma \vdash \text{with handler } \text{ret } (x : A) \mapsto c_r; h \text{ handle } \text{ret } v & \equiv c_r \left[ x \mapsto v \right] & \beta_{\text{HandlerRet}} \\
H = \text{handler } \text{ret } (x : A) \mapsto c_r; h \rightarrow (\text{op}_{A \rightarrow B}(x; k) \mapsto c_{op}) \in h & \equiv c_{op}[x \mapsto v, k \mapsto (\text{fun } y : B \mapsto \text{with } H \text{ handle } c)] & \beta_{\text{HandlerOp}} \\
\Gamma \vdash \text{match } (v_1, v_2) \text{ with } (x, y) \mapsto c & \equiv c \left[ x \mapsto v_1, y \mapsto v_2 \right] & \beta_{\text{MatchPair}} \\
\Gamma \vdash \text{match } \text{Left}_{A+B} \; v \text{ with } \text{Left } x \mapsto c_1 | \text{Right } y \mapsto c_2 & \equiv c_1 \left[ x \mapsto y \right] & \beta_{\text{MatchLeft}} \\
\Gamma \vdash \text{match } \text{Right}_{A+B} \; v \text{ with } \text{Left } x \mapsto c_1 | \text{Right } y \mapsto c_2 & \equiv c_2 \left[ y \mapsto v \right] & \beta_{\text{MatchRight}} \\
\Gamma \vdash \text{match } [ ] \text{ with } [ ] \mapsto c_1 | x :: xs \mapsto c_2 & \equiv c_1 \left[ x :: xs \mapsto c_2 \right] & \beta_{\text{MatchNil}} \\
\Gamma \vdash \text{match } v :: vs \text{ with } [ ] \mapsto c_1 | x :: xs \mapsto c_2 & \equiv c_2 \left[ x \mapsto v, xs \mapsto vs \right] & \beta_{\text{MatchCons}}
\end{align*}
\]

Note that there are no rules that account for small-step judgements such as

\[
\frac{c_1 \rightarrow c'_1}{\text{do } x \leftarrow c_1 \text{ in } c_2 \rightarrow \text{do } x \leftarrow c'_1 \text{ in } c_2}
\]

Such operational judgements can be modelled by using structural rules combined with \( \beta \)-rules on subterms.

**Lemma 5.4.2.** Suppose we have a well-typed computation \( \Gamma \vdash c : C \) and that \( c \rightarrow c' \). Then we can show that \( \Gamma \vdash c \equiv_C c' \).

**Proof (formalised).** The side condition of \( c' \) being well-typed directly follows from the safety Theorem 4.3.6. The proof proceeds by induction on structure of \( c \rightarrow c' \). Most cases have a directly applicable \( \beta \)-rule while the others can be resolved by applying an appropriate structural rule and using the induction hypothesis for subterms.

Another common set of rules are \( \eta \)-expansions. These rules allow us to elaborate on the structure of terms, which might help us by enabling further reductions. The importance
of $\eta$-laws is shown in Example 5.5.12.

\[ \Gamma \vdash v \equiv_{\text{unit}} (\eta) \]

\[ \Gamma \vdash f \equiv_{\lambda A \rightarrow C \text{ fun}} (x : A) \mapsto f x \]

\[ \Gamma, e : \text{empty}, \Gamma_2 \vdash c \equiv_{\text{absurd}} v \]

\[ \Gamma, p : A \times B, \Gamma_2 \vdash c \equiv_{\text{match}} v \]

\[ \Gamma \vdash c \equiv_{\text{match}} v \text{ with } (x, y) \mapsto (c[p \mapsto (x, y)]) \]

\[ \Gamma \vdash c \equiv_{\text{match}} v \text{ with } \text{Left} x \mapsto (c[s \mapsto \text{Left}(A + B) x]) \]

\[ \Gamma \vdash c \equiv_{\text{match}} v \text{ with } \text{Right} y \mapsto (c[l \mapsto x :: xs]) \]

\[ \Gamma \vdash c \equiv_{\text{do}} x \leftarrow c \text{ in } \text{ret } x \]

**5.4.1 Inheriting from theory**

Algebraic theories contain information on effect behaviour, and it is crucial to provide a way to transfer equations from computation types into the logic. In Section 4.4 we have demonstrated a way to instantiate templates to computations. An equation relates two templates, so it is natural to equate the pair of instantiated templates.

The simplest attempt is to instantiate templates in their general form by only translating them to computations.

\[ \Gamma_1, \Gamma_2 \vdash c[s \mapsto v] \]

\[ \Gamma_1, \Gamma_2 \vdash c[l \mapsto v] \]

\[ \Gamma_1, \Gamma_2 \vdash c \equiv_{\text{do}} x \leftarrow c \text{ in } \text{ret } x \]

The rule is not a good fit. Substitution is admissible with the current rules, but adding **BADRULE** creates a problem. To demonstrate the setback of the proposed rule, we take a look at an example.

\[ x : \text{int}; z : \text{unit} \rightarrow * \vdash \text{Ping}(x; _z()) \sim \text{Pong}(x; _z()) \]

The equation states that operations Ping and Pong are considered to be the same. We now use the rule BADRULE (at a fitting type $C$).

\[ x : \text{int}, z : \text{unit} \rightarrow C \vdash \text{Ping}(x; _z()) \equiv_{C} \text{Pong}(x; _z()) \]

If substitution was safe, we should be able to show that the terms are equal after using the substitution $[x \mapsto 6]$.

\[ z : \text{unit} \rightarrow C \vdash \text{Ping}(6; _z()) \equiv_{C} \text{Pong}(6; _z()) \]

We quickly notice that none of the rules available to us are able to produce such a conclusion. In fact, the more general form of instantiation that includes parallel substitution was introduced to resolve this issue. In the corrected rule, we ensure that we have a suitable
collection of well-typed terms and substitute them directly into the translated templates. This also changes the context of the logic judgement to the context in which the terms were typed.

\[
((x_i : A_i)_i ; (z_j : B_j \to \ast)_j ; T_1 \sim T_2) \in \mathcal{E}
\]

\[
A!\Sigma/\mathcal{E} \leq C \quad (\Gamma \vdash v_i : A_i)_i \quad (\Gamma \vdash u_j : B_j \to A!\Sigma/\mathcal{E})_j
\]

\[
\Gamma \vdash T'_1[A!\Sigma/\mathcal{E}][\{x_i \leftrightarrow v_i\}, \{z_j \leftrightarrow u_j\}] \equiv C \quad T'_2[A!\Sigma/\mathcal{E}][\{x_i \leftrightarrow v_i\}, \{z_j \leftrightarrow u_j\}]
\]

The name OOTB is a reference to how inherited equations appear “out of the blue”. It allows us to use the equation \( T_1 \sim T_2 \) without establishing it, as long as it is listed in \( \mathcal{E} \). The rule contains an explicit subtyping judgement, which significantly simplifies the proof of Lemma 5.4.9.

### 5.4.2 The respects relation

The last step is to define a rule for \textit{respects}, the motivation for the inclusion of logic in the type system. In the rule for typing a handler we require that the handler is correct with regard to the effect theory.

\[
\frac{\Gamma ; x : A \vdash c_r \vdash D \quad \Gamma \vdash h : \Sigma \Rightarrow D \quad \Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } \mathcal{E} \quad \Gamma \vdash \text{handler (ret (x : A) \mapsto c_r; h)} : A!\Sigma/\mathcal{E} \Rightarrow D}{\Gamma \vdash \text{handler (ret (x : A) \mapsto c_r; h)} : A!\Sigma/\mathcal{E} \Rightarrow D}
\]

A handler intuitively respects an equation if we can show that it handles both sides of the equation into equal results. While we can instantiate templates into computations, we cannot show correctness by just wrapping instantiated computations with the handler.

\[
\frac{H = \text{handler (ret (x : A) \mapsto c_r; h)} \quad \Gamma \vdash \text{with H handle } T'_1 \equiv_D \text{ with H handle } T'_2 \equiv_D \quad \Gamma \vdash H : C \Rightarrow D \text{ respects } \{\Gamma ; Z \vdash T_1 \sim T_2\}}{\Gamma \vdash \text{BADRule}}
\]

Terms in logic judgements need to be well typed. Using the handler \( H \), whose typing derivation we are currently constructing, is therefore not possible. The above rule is simple but it is not clear how to adjust the type system. We instead look for another option that does not require modifications of the type system but avoids circularity in type derivation.

The idea is to directly simulate handler application on the templates. We generalise the template instantiation \( T'_1[D] \) to \( T'_h[D] \), which now unfolds operation cases over the template during translation. For \( h = \{op_{A \to B}(x; k) \mapsto c_{op}\}_op \) we define

\[
(z_i \mapsto v)^h_D = z_i \mapsto v
\]

\[
(op_{A \to B}(v; y.T))^h_D = c_{op}[x \mapsto v, k \mapsto (\text{fun } (y : B) \mapsto T^h_D)]
\]

\[
(\text{do pure } x \leftarrow c \text{ in } T^h_D) = \text{do pure } x \leftarrow c \text{ in } T^h_D
\]

\[
(\text{absurd } v)^h_D = \text{absurd}_D v
\]

\[
(\text{match } v \text{ with } (x,y) \mapsto T^h_D) = \text{match } v \text{ with } (x,y) \mapsto (T^h_D)
\]

\[
(\text{match } v \text{ with } [ ] \mapsto T^h_D) = \text{match } v \text{ with } [ ] \mapsto (T^h_D)
\]

\[
(\text{match } v \text{ with } \text{Left } x \mapsto T_1 | \text{Right } y \mapsto T_2)^h_D = \text{match } v \text{ with } \text{Left } x \mapsto (T^h_D) | \text{Right } y \mapsto (T^h_D)
\]
Because we cannot explicitly handle arbitrary continuations represented by template variables, we instead consider them at a new type. For a handler of type $C \Rightarrow D$, think of template variables $z_i$ as having the type $B_i \rightarrow C$ before the procedure and $B_i \rightarrow D$ afterwards. Using a handler only affects the operation calls, which are now replaced with appropriate computations $c_{op}$. This procedure also justifies the restriction of purity in do pure sequencing, since we cannot unfold operation cases over an arbitrary computation. But operation cases have no effect on pure computations, so we do not need to handle the computation in do pure.

In comparing $T^h_D$ to $T^I_D$, we see that $I$ should not be understood as a notation for “instantiation”, but rather “identity cases”. Identity cases only propagate operations outwards, having $op_{A \rightarrow B}(x; k) \mapsto op_{A \rightarrow B}(x; y.ky)$ for every operation.

**Example 5.4.3.** We borrow part of Example 5.5.10 to showcase the generalised template instantiation. Assume we have operation cases which interpret Choose : unit → bool to always return true.

$$h_{\text{true}} := \{\text{Choose}(\_; k) \mapsto k \text{ true}\}$$

The equation used for the example is idem from Subsection 3.2.1.

$$\text{Choose}(\_; y.\text{if } y \text{ then } z() \text{ else } z()) \sim z()$$

We apply $(\_)_D^{h_{\text{true}}}$ on the left template of the equation and simplify it step by step.

$$(\text{Choose}(\_; y.\text{if } y \text{ then } z() \text{ else } z()))_D^{h_{\text{true}}}$$

$$(k \text{ true})[k \mapsto \{\text{fun } y \mapsto (\text{if } y \text{ then } z() \text{ else } z())_D^{h_{\text{true}}}\}]$$

$$(k \text{ true})[k \mapsto \{\text{fun } y \mapsto \text{if } y \text{ then } (z())_D^{h_{\text{true}}} \text{ else } (z())_D^{h_{\text{true}}}\}]$$

$$(k \text{ true})[k \mapsto \{\text{fun } y \mapsto \text{if } y \text{ then } z() \text{ else } z()\}]$$

$$(\text{fun } y \mapsto \text{if } y \text{ then } z() \text{ else } z())\text{ true}$$

We make a small check that the result is sensible. We mark $z$ with its outgoing type, meaning $z_C : \text{unit} \rightarrow C$, for clarity. Assume $H : C \Rightarrow D$ is a handler with cases $h_{\text{true}}$. If we were to simulate the result of using $H$ on the template, we would expect something along the lines of

$$(\text{fun } y \mapsto \text{if } y \text{ then } (\text{with } H \text{ handle } z_C()) \text{ else } (\text{with } H \text{ handle } z_C()))\text{ true}\}.$$ 

However we are unable to use $H$, because we wish to use this technique to type $H$. We therefore generalise (with $H$ handle $z_C()$) : $D$ to an arbitrary computation of type $D$, namely $z_D()$, which is precisely what we do in $(\_)_D^{h_{\text{true}}}$.

It is important to ensure that $T^h$ correctly simulates handling of templates. We start by making sure that the procedure produces well-typed computations. In Chapter 7 we then use denotational semantics to prove that handling an instantiated template $T^I$ with cases $h$, is equivalent to $T^h$. This can be seen by combining Lemma 7.1.1 and Lemma 7.2.5.

**Lemma 5.4.4.** Assume a well-typed template $(x_i : A_i)i ; (z_j : B_j \rightarrow *)j + T : \Sigma$ and well-typed operation cases $\Gamma \vdash h : \Sigma \Rightarrow D$. Then we can show that

$$\Gamma,(x_i : A_i)i,(z_j : B_j \rightarrow D)j + T^h_D : D.$$
Proof (formalised). The proof is done by induction on the well-formedness judgement of $T$. The proof proceeds similarly to the instantiation Lemma 4.4.1, with the difference of operation calls. Because $h$ is well typed for $\Sigma$, for which $T$ is well formed, we are guaranteed to find a suitable $c_{\text{op}}$ for operations in $T$. To successfully use the substitution lemma, we need to type the captured continuation ($\text{fun} \ (y : B) \mapsto T^h_D$), and are able to do so by using the induction hypothesis. 

We now use $T^h$ to define $\text{respects}$, which ensures that handling both sides of the equation produces equal results.

\[ \Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } \{ \} \quad \text{RespectEqs\{\}} \]

\[ \Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } E \quad \Gamma, (x_i : A_i)_i, (z_j : B_j \rightarrow D)_j \vdash T^h_D \equiv_D T^h_{T^h_D} \quad \text{RespectEqsU} \]

A slight oversight of this approach is that the value case of handlers plays no role in checking whether a handler respects equations. We simply assume that template variables have been handled by assigning them a new type $B \rightarrow D$, but the precise information on what happens to returned values is lost in the process. There are examples of handlers that ultimately respect their equations, but fail to type with the current $\text{respects}$ relation. 

\[ \text{handler } \ (\text{ret } x \mapsto 0; \{ \text{Choose} (\_ : k) \mapsto k \ \text{true} \}) \]

The above handler clearly respects the commutativity of $\text{Choose}$, because it always returns 0, but looking only at operation cases we are unable to prove so.

5.4.3 Properties

Lemma 5.4.5.

- For well-typed values $\Gamma \vdash v : A$, we can show $\Gamma \vdash v \equiv_A v$.
- For well-typed computations $\Gamma \vdash c : C$, we can show $\Gamma \vdash c \equiv_C c$.
- For well-typed operation cases $\Gamma \vdash h : \Sigma \Rightarrow D$, we can show $\Gamma \vdash h \equiv_{\Sigma \Rightarrow D} h$.

Proof (formalised). The proof proceeds by induction on the typing derivation. We have reflexivity rules for base types and variables; otherwise, we can use the appropriate structural rules and use induction on subterms. For operation cases we use Lemma 5.4.1. 

Lemma 5.4.6.

- If we have $\Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_2$, we can also show $\Gamma \vdash h_2 \equiv_{\Sigma \Rightarrow D} h_1$.
- From $\Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_2$ and $\Gamma \vdash h_2 \equiv_{\Sigma \Rightarrow D} h_3$, we can show $\Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_3$.

Proof (formalised). We focus on transitivity and the proof for symmetry is similar. We use induction on the structure of $\Sigma$, with the case of $\{\}$ being trivial. In the case of a constructed signature $\{ \text{op} : A \rightarrow B \} \cup \Sigma'$, we notice only $\text{HeqSigU}$ results in $\{ \text{op} : A \rightarrow B \} \cup \Sigma' \Rightarrow D$. This lets us deconstruct the assumption $\Gamma \vdash h_1 \equiv_{\{ \text{op} : A \rightarrow B \} \cup \Sigma' \Rightarrow D} h_2$ into $\Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_2$ and $\Gamma, x : A : B \vdash D \Rightarrow c_{\text{op}}^h \equiv_D c_{\text{op}}^h$, where $c_{\text{op}}^h$ denotes the computation used for handling op in $h_1$. After deconstructing the assumption for $h_2$ and $h_3$, we use induction to obtain $\Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_3$ due to the smaller signature, and transitivity for computations gives us $\Gamma, x : A : B \vdash D \Rightarrow c_{\text{op}}^1 \equiv_D c_{\text{op}}^3$. We conclude the proof by using $\text{HeqSigU}$. 

Lemma 5.4.7. This is the continuation of Lemma 4.3.3. Assume we have two well-typed contexts for which $\Gamma' \leq \Gamma$ holds.
If \( \Gamma \vdash v_1 \equiv_A v_2 \) holds, then we can show \( \Gamma' \vdash v_1 \equiv_A v_2 \).

If \( \Gamma \vdash c_1 \equiv_C c_2 \) holds, then we can show \( \Gamma' \vdash c_1 \equiv_C c_2 \).

If \( \Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2 \) holds, then we can show \( \Gamma' \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2 \).

**Proof (formalised).** Simple induction on derivation of term equality.

Instead of adding subtyping through rules in the logic system, we use it only where needed. The three places where subtyping is required are the rule for variables, the structural rule for handlers, and the rule for inheriting equations. This suffices to show that we are always able to relax our equality judgements to a super-type.

**Lemma 5.4.8** (Subtyping in logic).

- Assume we have \( \Gamma \vdash v_1 \equiv_A v_2 \) and \( v' : \text{vtype} \) where \( A \leq A' \). Then we can also show \( \Gamma \vdash v_1 \equiv_{A'} v_2 \).

- Assume we have \( \Gamma \vdash c_1 \equiv_C c_2 \) and \( C' : \text{vtype} \) where \( C \leq C' \). Then we can also show \( \Gamma \vdash c_1 \equiv_{C'} c_2 \).

- Assume we have \( \Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2 \) and \( \Sigma' : \text{sig} \) where \( \Sigma' \leq \Sigma' \). Then we can also show \( \Gamma \vdash h_1 \equiv_{\Sigma' \Rightarrow_D} h_2 \).

**Proof (formalised).** We separate the proof into two parts. We first prove the lemma for operation cases. Unlike with values and computations, the side conditions for operation cases are not clearly trivial. From \( \Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2 \) we know that there are \( \Sigma_1 \) and \( \Sigma_2 \) with \( \Sigma \leq \Sigma_1 \) and \( \Sigma \leq \Sigma_2 \), for which \( \Gamma \vdash h_1 : \Sigma_1 \Rightarrow_D D \) and \( \Gamma \vdash h_2 : \Sigma_2 \Rightarrow_D D \). We dispatch the side conditions by using transitivity of subtyping to show \( \Sigma' \leq \Sigma \leq \Sigma_1 \) (and similarly for \( \Sigma_2 \)), so we can reuse the existing typing judgements for \( h_1 \) and \( h_2 \). We proceed by induction on structure of \( \Sigma' \), with the case of \( \{ \} \) being trivial. For a constructed signature, we first show that for \( op \in \Sigma \), from the assumption \( \Gamma \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2 \) it follows that the cases for \( op \) are also equal. This can be shown as a side lemma with a simple proof by induction. The proof of the current lemma is then completed by applying HEQSIGU.

The proof for values and computations is more straightforward. We use induction on the derivation of \( \Gamma \vdash v_1 \equiv_A v_2 \) and \( \Gamma \vdash c_1 \equiv_C c_2 \). Most cases are made trivial by Lemma 5.4.7. In the case of variables, handlers, and OOTB, it becomes clear that we really need the “built-in” subtyping.

The full definition of the substitution Lemma 4.3.5 also needs to include the appropriate logic judgements. The proofs can be found in the formalisation.

**Lemma 5.4.9** (Substitution). Assume that we have a well-typed value \((\Gamma_1, \Gamma_2) \vdash v : B\).

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash v_1 \equiv_A v_2\), then we can show \((\Gamma_1, \Gamma_2) \vdash v_1[x \mapsto v] \equiv_A v_2[x \mapsto v]\).

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash c_1 \equiv_C c_2\), then we can show \((\Gamma_1, \Gamma_2) \vdash c_1[x \mapsto v] \equiv_C c_2[x \mapsto v]\).

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash h_1 \equiv_{\Sigma \Rightarrow_D} h_2\), then we can show \((\Gamma_1, \Gamma_2) \vdash h_1[x \mapsto v] \equiv_{\Sigma \Rightarrow_D} h_2[x \mapsto v]\).

The above Lemma is not the only interaction of substitution and logic equations. Instead of observing what happens when applying the same substitution on two related terms, we also take a look at performing two different substitutions on the same term, where we substitute a variable with two related values.
Lemma 5.4.10. Assume that we have a pair of related values \((\Gamma_1,\Gamma_2) \vdash v \equiv_B v'\).

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash u : A\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash u[x \mapsto v] \equiv_A u[x \mapsto v'].\]

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash c : C\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash c[x \mapsto v] \equiv_C c[x \mapsto v'].\]

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash h : \Sigma \Rightarrow D\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash h[x \mapsto v] \equiv_{\Sigma \Rightarrow D} h[x \mapsto v'].\]

Proof (formalised). The proof proceeds with the induction on typing derivation. We first push the substitution to subterms. If the term is the variable \(x\), it gets replaced with \(v\) and \(v'\) respectively, so we use the assumption \((\Gamma_1, \Gamma_2) \vdash v \equiv_B v'\). Otherwise, we proceed by using structural rules for values and computations, and Lemma 5.4.1 for operation cases. The subterms are then related by induction. \(\square\)

We can join the two Lemmas about substitution into a more powerful version of the substitution Lemma for logic.

Corollary 5.4.11. Assume that we have a pair of related values \((\Gamma_1,\Gamma_2) \vdash v \equiv_B v'\).

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash v_1 \equiv_A v_2\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash v_1[x \mapsto v] \equiv_A v_2[x \mapsto v'].\]

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash c_1 \equiv_C c_2\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash c_1[x \mapsto v] \equiv_C c_2[x \mapsto v'].\]

- If we have \((\Gamma_1, x : B, \Gamma_2) \vdash h_1 \equiv_{\Sigma \Rightarrow D} h_2\), then we can show
  \[(\Gamma_1, \Gamma_2) \vdash h_1[x \mapsto v] \equiv_{\Sigma \Rightarrow D} h_2[x \mapsto v'].\]

Proof (formalised). The proof is stated for values, but the proofs for computations and computation cases follow the same pattern. We know that \((\Gamma_1, \Gamma_2) \vdash v : B\) since it is a side condition of \((\Gamma_1, \Gamma_2) \vdash v \equiv_B v'\). From Lemma 5.4.9 we get \((\Gamma_1, \Gamma_2) \vdash v_1[x \mapsto v] \equiv_A v_2[x \mapsto v].\)

Next we use Lemma 5.4.10 to obtain \((\Gamma_1, \Gamma_2) \vdash v_2[x \mapsto v] \equiv_A v_2[x \mapsto v'].\) The rest follows from transitivity. \(\square\)

5.4.4 Examples

Example 5.4.12. The first example is kept simple to familiarise the reader with constructing proofs in logic. We revisit the theory of nondeterminism provided as an example theory in Section 3.2. We use Booleans, conditionals, and other syntactic sugar, and also omit contexts and types when they are easily inferred. We also ignore side conditions for well-formedness and terms being well typed, as they play little role in proof ideas.

Recall the signature for nondeterministic choice.

\[\Sigma_\oplus := \{\text{Choose} : \text{unit} \rightarrow \text{bool}\}\]

In Section 3.2 we introduced a shorter notation for equations of nondeterminism:

\[T_1 \oplus T_2 := \text{Choose}((()); \text{if } y \text{ then } T_1 \text{ else } T_2)\]

\[\bar{x} := z()\]
which allows us to write equations much more concisely.

\[ \tilde{z} \oplus \tilde{z} \sim \tilde{z} \quad \text{(IDEM)} \]
\[ \tilde{z}_1 \oplus \tilde{z}_2 \sim \tilde{z}_2 \oplus \tilde{z}_1 \quad \text{(COMM)} \]
\[ \tilde{z}_1 \oplus (\tilde{z}_2 \oplus \tilde{z}_3) \sim (\tilde{z}_1 \oplus \tilde{z}_2) \oplus \tilde{z}_3 \quad \text{(ASSOC)} \]

We construct a simple handler, which always chooses the left branch i.e. returns the value `true` on all invocations of `Choose`. The focus of this proof is the set of operation cases:

\[ h_{\text{true}} := \{ \text{Choose}(_; k) \mapsto k \text{ true} \} : \Sigma \Rightarrow D \]

The use of an arbitrary computation type \( D \) implies that we do not need any specific property of continuations.

Our goal is to show that \( h_{\text{true}} \) respects as many of the equations as possible. Before we plunge into proofs, let us first take a glance at which equations we expect to hold. A handler using \( h_{\text{true}} \) will always select the option left of \( \oplus \), so we expect that after handling with \( h_{\text{true}} \), we end up with:

\[ \tilde{z} \oplus \tilde{z} \sim \tilde{z} \]
\[ \tilde{z}_1 \oplus \tilde{z}_2 \sim \tilde{z}_2 \oplus \tilde{z}_1 \]
\[ \tilde{z}_1 \oplus (\tilde{z}_2 \oplus \tilde{z}_3) \sim (\tilde{z}_1 \oplus \tilde{z}_2) \oplus \tilde{z}_3 \]

This leads us to suspect that we will be able to show that \( h_{\text{true}} \) respects IDEM and ASSOC, but fail to show correctness with regard to COMM.

**IDEM**: We start by partially desugaring \( \oplus \) and \( \tilde{z} \) in IDEM to obtain a form that fits our logic system.

\[ \text{Choose}(); y.\text{if } y \text{ then } z() \text{ else } z() \sim z() \]

To show \( \Gamma \vdash h_{\text{true}} : \Sigma \Rightarrow D \) respects IDEM, we proceed by applying \( (\_)^{h_{\text{true}}} \) to both sides of the equation and obtain a logic equation at \( D \). The step-by-step process is presented in Example 5.4.3.

\[
(\text{Choose}(); y.\text{if } y \text{ then } z() \text{ else } z())^{h_{\text{true}}} \equiv_D (z())^{h_{\text{true}}}
\]

\[
(\text{fun } y \mapsto \text{if } y \text{ then } z() \text{ else } z())\text{ true} \equiv_D z()
\]

In proofs we implicitly rely on transitivity of \( \equiv \). We see that the term on the left side of the equation can be reduced through evaluation, meaning that we can use \( \beta\text{APP} \).

\[
(\text{fun } y \mapsto \text{if } y \text{ then } z() \text{ else } z())\text{ true} \equiv_D \text{if true then } z() \text{ else } z()
\]

By transitivity we only need to show

\[
\text{if true then } z() \text{ else } z() \equiv_D z(),
\]

effectively reducing the left side. We again use a \( \beta \)-reduction, this time for conditionals (which is sugared \( \beta\text{MATCHLEFT} \)).

\[
z() \equiv_D z()
\]

At this point we use Lemma 5.4.5 for reflexivity to conclude our proof.

**ASSOC**: We desugar the equation \( \tilde{z}_1 \oplus (\tilde{z}_2 \oplus \tilde{z}_3) \sim (\tilde{z}_1 \oplus \tilde{z}_2) \oplus \tilde{z}_3 \) to conditionals.

\[
\text{Choose}(); y.\text{if } y \text{ then } \tilde{z}_1() \text{ else } \text{Choose}(); y'.\text{if } y' \text{ then } \tilde{z}_2() \text{ else } \tilde{z}_3())
\]

\sim

\[
\text{Choose}(); y.\text{if } y \text{ then } \text{Choose}(); y'.\text{if } y' \text{ then } \tilde{z}_1() \text{ else } \tilde{z}_2()) \text{ else } \tilde{z}_3())
\]

51
We proceed by applying $(_ D)^{htue}$ to both sides.

\[(\text{fun } y \mapsto \text{if } y \text{ then } z_1 () \text{ else } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_2 () \text{ else } z_3 ()) \text{ true}) \text{ true})\]

\[\equiv_D\]

\[(\text{fun } y \mapsto \text{if } y \text{ then } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_1 () \text{ else } z_2 ()) \text{ true}) \text{ else } z_3 ()) \text{ true})\]

The terms are rather large, so we reduce each one separately, starting from the left side of the equation. We switch to a more terse style, where we chain reductions one after another.

\[(\text{fun } y \mapsto \text{if } y \text{ then } z_1 () \text{ else } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_2 () \text{ else } z_3 ()) \text{ true}) \text{ true}) \equiv_D (\text{reduce application})\]

\[\text{if true then } z_1 () \text{ else } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_2 () \text{ else } z_3 ()) \text{ true}) \equiv_D (\text{reduce conditional})\]

\[z_1 ()\]

We reduce the right side of the equation similarly.

\[(\text{fun } y \mapsto \text{if } y \text{ then } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_1 () \text{ else } z_2 ()) \text{ true}) \text{ else } z_3 ()) \text{ true}) \equiv_D (\text{reduce application})\]

\[\text{if true then } ((\text{fun } y' \mapsto \text{if } y' \text{ then } z_1 () \text{ else } z_2 ()) \text{ true}) \text{ else } z_3 () \equiv_D (\text{reduce conditional})\]

\[(\text{fun } y' \mapsto \text{if } y' \text{ then } z_1 () \text{ else } z_2 ()) \text{ true}) \equiv_D (\text{reduce application})\]

\[\text{if true then } z_1 () \text{ else } z_2 () \equiv_D (\text{reduce conditional})\]

\[z_1 ()\]

With both sides having been reduced to $z_1 ()$, we conclude the proof through reflexivity.

**COMM:** We suspect that our handler does not respect the equation for commutativity, but we entertain the thought of attempting to fashion a proof. This time, we need to show

\[(\text{fun } y \mapsto \text{if } y \text{ then } z_1 () \text{ else } z_2 ()) \text{ true}) \equiv_D (\text{fun } y \mapsto \text{if } y \text{ then } z_2 () \text{ else } z_1 ()) \text{ true})\]

By reducing both sides through $\beta$-laws we end up with $z_1 () \equiv_D z_2 ()$ and are unable to continue, since—in general—$z_1$ is not equal to $z_2$.

**Example 5.4.13.** In Section 2.3 we showcased how to use handlers for mutable state, and in Section 3.2 we introduced the usual equations of the state theory. We use $\text{state}$ as an abstract type that represents the contents of state.

\[\Sigma_{\text{state}} := \{\text{Get : unit} \rightarrow \text{state}, \text{Set : state} \rightarrow \text{unit}\}\]

\[\text{Get()} ; s . \text{Get}(); s'.z(s,s') \sim \text{Get}(); s.z(s,s)\]  \hspace{1cm} (\text{GetGet})

\[\text{Get}(); s . \text{Set}(s; _ .z()) \sim \text{set}()\]  \hspace{1cm} (\text{GetSet})

\[\text{Set}(s; _ .\text{Get}(); s'.z(s,s')) \sim \text{Set}(s; _ .z s)\]  \hspace{1cm} (\text{SetGet})

\[\text{Set}(s; _ .\text{Set}(s'; _ .z()) \sim \text{Set}(s'; _ .z())\]  \hspace{1cm} (\text{SetSet})

\[\mathcal{E}_{\text{state}} := \{\text{GetGet, GetSet, SetGet, SetSet}\}\]

Recall the definition of the $\text{state_handler}$ presented as an example in Chapter 2.
We first desugar the operation cases.

\[
\begin{align*}
\text{state
d_handler} = \ & \text{handler} \\
| \text{Get} \ k \ -> \ (\text{fun} \ s -> (k \ s) \ s) \\
| \text{Set} \ x \ k \ -> \ (\text{fun} \ _ -> (k () \ x) \\
| \ x -> (\text{fun} \ s -> (x, s))
\end{align*}
\]

The precise type of operation cases tells us at which type we need to show correctness.

\[
\begin{align*}
\cdot \vdash h_{\text{state}} : \Sigma_{\text{state}} \Rightarrow (\text{state} \to A \{\}/\{\}/\{\}/\{\}/\{\}) \text{ respects } E_{\text{state}}
\end{align*}
\]

To avoid confusion with too many variables named \(s\), we use \(s\) and \(s'\) to denote arguments of the functions used in the handler, while state values from equation contexts are named \(x\) and \(x'\). We begin with the equation \(\text{SetSet}\) (adjusted to new names).

\[
\begin{align*}
\text{Set}(x; \ _\text{Set}(x'; \ _\text{z} ())) \sim \text{Set}(x'; \ _\text{z} ())
\end{align*}
\]

To show correctness, we need to provide a proof for

\[
\begin{align*}
\text{ret \ fun \ s} \ & \rightarrow (do \ k' \ left (fun \ _ \rightarrow (fun \ s' \ left (do \ k'' \ left (fun \ _ \rightarrow \text{z} ()) () in k'' \ x'))) () in k' \ x) \\
\ & \equiv (do \ k' \ left (fun \ _ \rightarrow \text{z} ()) () in k' \ x')
\end{align*}
\]

The terms \(k' \ x\) and \(k' \ x'\) are different, and we cannot use \(\text{VEQDO}\) directly, but the subterms can be reduced further. Combining \(\text{VEQDO}\) and reflexivity on \(c_2 \equiv c_2\), we see that \(c_1 \equiv c'_1\) implies \(do \ x \ left c_1 \ in \ c_2\) \equiv \(do \ x \ left c'_1 \ in \ c_2\), which gives us a way to reduce subterms.

\[
\begin{align*}
\text{do} \ k' \ & \leftarrow (fun \ _ \rightarrow (fun \ s' \ left (do \ k'' \ left (fun \ _ \rightarrow \text{z} ()) () in k'' \ x'))) () in k' \ x \\
& \equiv (do \ k' \ left (fun \ _ \rightarrow \text{z} ()) () in k' \ x')
\end{align*}
\]

\[
\begin{align*}
\text{do} \ k' \ & \leftarrow (fun \ s' \ left (do \ k'' \ left (fun \ _ \rightarrow \text{z} ()) () in k'' \ x'))) () in k' \ x \\
& \equiv (do \ k' \ left (fun \ _ \rightarrow \text{z} ()) () in k' \ x')
\end{align*}
\]

\[
\begin{align*}
\text{do} \ k'' \ & \leftarrow (fun \ _ \rightarrow \text{z} ()) () in k'' \ x'
\end{align*}
\]
The right side of the original equation can be reduced with a single step. We end up with

\[
\text{do } k'' \leftarrow z() \text{ in } k'' x' \equiv A() / 1 \text{ do } k' \leftarrow z() \text{ in } k' x'.
\]

These two terms are equal up to the renaming, which concludes our proof for \texttt{SetSet}.

The equations \texttt{GetGet} and \texttt{SetGet} have similar proofs, but the proof for \texttt{GetGet} is slightly more curious. To show \( \Gamma \vdash h_{\text{state}} : \Sigma_{\text{state}} \Rightarrow D \) respects \{\texttt{GetGet}\}, we need to prove the following:

\[
\text{ret fun } s \mapsto
\text{do } k' \leftarrow \text{ (fun } s' \mapsto \text{ (ret fun } s'' \mapsto \text{ (do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s')) s \text{ in } k' s') \equiv_{\text{state} \mapsto A(!)(/)(/)/} z()
\]

This time, we cannot use the structural rules to remove \texttt{ret} and \texttt{fun}. We proceed by reducing the body of the function on the left side.

\[
\text{do } k' \leftarrow \text{ (fun } s' \mapsto \text{ (ret fun } s'' \mapsto \text{ (do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s')) s \text{ in } k' s')
\equiv_{\text{state} \mapsto A(!)(/)(/)/} \langle \text{ reduce application in subterm } \rangle
\text{do } k' \leftarrow \text{ ret } \text{ (fun } s'' \mapsto \text{ (do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s')) s \text{ in } k' s'
\equiv_{\text{state} \mapsto A(!)(/)(/)/} \langle \text{ use } \beta \text{DoRet } \rangle
\text{(k' } s)[k' \mapsto \text{ (fun } s'' \mapsto \text{ (do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s'))]
\equiv_{\text{state} \mapsto A(!)(/)(/)/} \langle \text{ simplify substitution } \rangle
\text{(fun } s'' \mapsto \text{ (do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s')) s
\equiv_{\text{state} \mapsto A(!)(/)(/)/} \langle \text{ reduce application } \rangle
\text{do } k'' \leftarrow \text{ (fun } _\rightarrow z()() \text{ in } \text{ k'' } s)
\equiv_{\text{state} \mapsto A(!)(/)(/)/} \langle \text{ reduce application in subterm } \rangle
\text{do } k'' \leftarrow z() \text{ in } \text{ k'' } s
\]

We end up having to show

\[
\text{ret fun } s \mapsto \text{ (do } k'' \leftarrow z() \text{ in } \text{ k'' } s) \equiv_{\text{state} \mapsto A(!)(/)(/)/} z()
\]

which is, in fact, not true. In a surprising turn of events, the usual state handler does not respect the \texttt{GetSet} equation in the presence of recursion. In a language with strictly terminating programs, we could use computational induction (see Section 5.5) to complete the proof, but that is not possible with nontermination. For example, assume \texttt{loop} is some nonterminating computation.

\[
c_1 := \text{ with state }\_\text{handler handle } \text{Get}((); \text{ s.Set(s; }\_\text{.loop)})
c_2 := \text{ with state }\_\text{handler handle } \text{loop}
\]

If the state handler respected \texttt{GetSet}, then the computations \(c_1\) and \(c_2\) should be equal. However, if we run \(c_1\), we obtain a function that accepts an argument and then goes into a loop, while in the case of \(c_2\), we immediately go into an infinite loop. The distinction is rather small, but ultimately not harmless. We can recover some of the functionality through weaker versions of \texttt{GetSet}, which has a proof similar to that of \texttt{SetSet}.

\[
\text{Get}((); \text{s.Set(s; }\_\text{.z s}) \sim \text{Get}((); \text{s.z s}) \quad \text{(WeakGetSet)}
\]

54
5.5 Predicate logic with induction

A powerful tool for reasoning in the presence of algebraic effects is the principle of computational induction \([35, 6]\), which takes into account the structure of computations. To extend the equational logic from Section 5.4 with computational induction we need to introduce a number of changes. Formulating the induction judgement requires logical quantifiers and hypotheses, so it is natural to switch to a predicate logic. A terse collection of the predicate logic rules can also be found in Appendix A.4.

The equation judgements are replaced with the more general formulae, defined in Figure 5.1. We do not include negation, as it can be defined as \(\neg \varphi := \varphi \Rightarrow \bot\).

**formulae** \(\varphi, \psi ::=\)

- \(v_1 \equiv_A v_2\) value equation
- \(c_1 \equiv_C c_2\) computation equation
- \(h_1 \equiv_{\Sigma = D} h_2\) cases equation
- \(\top\) truth
- \(\bot\) falsity
- \(\varphi_1 \land \varphi_2\) conjunction
- \(\varphi_1 \lor \varphi_2\) disjunction
- \(\varphi_1 \Rightarrow \varphi_2\) implication
- \(\forall x : A. \varphi\) universal quantification
- \(\exists x : A. \varphi\) existential quantification

**hypotheses** \(\Psi ::= \ldots \mid \Psi, \varphi\)

Figure 5.1: Syntax of formulae and hypotheses.

The mutually recursive definition of the type system from Chapter 4 is extended with the following:

- \(\Gamma \mid \Psi \vdash \varphi\), which states that in the context \(\Gamma\), assuming hypotheses \(\Psi\), the formula \(\varphi\) holds.
- \(\Gamma \vdash \varphi : \text{form}\), which states that the formula \(\varphi\) is well formed in the context \(\Gamma\).
- \(\Gamma \vdash \Psi : \text{hyp}\), which states that all hypotheses of \(\Psi\) are well formed in \(\Gamma\).

Side conditions of equations are moved to well-formedness of formulae for a more streamlined approach. We keep the stronger notion of equations on handler cases, so we allow them to type at different super-type signatures. The rules for other formulae proceed structurally.

\[
\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1 \equiv_A v_2 : \text{form}}_{\text{WFVeq}} \quad \frac{\Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C}{\Gamma \vdash c_1 \equiv_C c_2 : \text{form}}_{\text{WFCeq}}
\]

\[
\frac{\vdash \Sigma : \text{sig} \quad \Sigma \leq \Sigma_1 \quad \Sigma \leq \Sigma_2 \quad \Gamma \vdash h_1 : \Sigma_2 \Rightarrow D \quad \Gamma \vdash h_2 : \Sigma_2 \Rightarrow D}{\gamma h_1 \equiv_{\Sigma = D} h_2 : \text{form}}_{\text{WPHeq}}
\]

55
with only some of the old rules requiring slight corrections. In equational logic, symmetry hypotheses. Because none of the old rules use hypotheses, we simply change

Changes to rules of equational logic

Checking that hypotheses are well formed requires simply checking each separate formula.

In all rules for $\Gamma \vdash \varphi$ we implicitly assume $\vdash \Gamma : \text{ctx}$, $\Gamma \vdash \Psi : \text{hyp}$ and $\Gamma \vdash \varphi : \text{form}$.

**Remark 5.5.1.** In the following subsections we cover the changes and additions to the rules of the logic. It should be noted that we refrain from adding a substitution rule where from the premises $v_1 \equiv v_2$ and $\varphi[x \mapsto v_1]$ it would follow that $\varphi[x \mapsto v_2]$. The decision was made in order to simplify the formalisation, since substitution tends to be difficult to work with. The resulting logic still interacts nicely with substitution (see Lemma 5.5.6 and Lemma 5.5.7) but is not as powerful. The restricted substitution properties that our equality relation enjoys are analogous to those of the "weakly extensional" equality in the Dialectica interpretation of higher-order arithmetic [44].

Changes to rules of equational logic

Since predicate logic is an extension of equational logic, we adapt all rules to account for hypotheses. Because none of the old rules use hypotheses, we simply change $\Gamma$ to $\Gamma | \Psi$, with only some of the old rules requiring slight corrections. In equational logic, symmetry and transitivity are inherited on operation cases (Lemma 5.4.6), which is not the case in predicate logic, because there are now multiple ways to obtain $\equiv_{\Sigma = D}$. We are similarly forced to include the structural rule $\text{HEQEXTEND}$, structured similar to Lemma 5.4.1.

The inclusion of hypotheses also requires the inclusion of subsumption rules, and we in turn remove the hardcoded requirements of $\text{VEQVAR}$, $\text{VEQHANDLER}$, and $\text{OOTB}$. We can also remove the rule $\text{ηDO}$, since it can be proven within the predicate logic, as is shown.
in Example 5.5.9.

\[
\begin{align*}
\Gamma, x : A &\vdash D \text{ respects } E \\
\Gamma, x : A | \Psi &\vdash c \equiv_D c' \\
\Gamma, h : \Sigma &\vdash D \text{ respects } E \\
\end{align*}
\]

The other judgements that require a bit more attention are judgements for \textit{respects}. Side conditions remain \(\vdash E : \Sigma\) and \(\Gamma \vdash h : \Sigma \Rightarrow D\), but we ensure that the proof of correctness is done with no hypotheses.

\[
\begin{align*}
\Gamma, h : \{\} &\vdash \Sigma \text{ respects } D \\
\Gamma, h : E &\vdash \Sigma \text{ respects } D \\
\end{align*}
\]

Enabling hypotheses in \textit{respects} requires threading hypotheses through all constructs of the type system, resulting in significant modifications. We see little benefit in conditionally correct handlers and consider this detrimental to the idea of a lightweight approach to algebraic effect theories.

Rules for logical connectives

The predicate logic features all of the standard introduction and elimination rules for logical connectives, including \textit{ISHYP} that allows the use of hypotheses.

\[
\begin{align*}
\varphi \in \Psi &\quad \text{ISHYP} \\
\Gamma \vdash \varphi &\quad \text{\texttt{In}} \\
\Gamma \vdash \top &\quad \text{\texttt{El}} \\
\Gamma \vdash \varphi &\quad \text{\texttt{El}} \\
\Gamma \vdash \varphi \land \varphi' &\quad \text{\texttt{ElLeft}} \\
\Gamma \vdash \varphi' &\quad \text{\texttt{ElRight}} \\
\Gamma \vdash \varphi \lor \varphi' &\quad \text{\texttt{ElLeft}} \\
\Gamma \vdash \varphi' &\quad \text{\texttt{ElRight}} \\
\end{align*}
\]
∀

let rec

g

check results in a correct induction principle. We settle for using a simple recursive loop

tions are treated the same, so we argue that using any nonterminating computation for the

denotational based approach [6]. In denotational semantics, all nonterminating computa-

instead draw inspiration from the induction principle described in Bauer and Pretnar’s

further complicate the use of a nontermination operation, as it impacts term types. We

the approach is more thorough, we wish to remain in a simpler logic, and effect systems

further extended with inequalities and nontermination is simulated as an operation. While

The first two requirements are expressed naturally in the predicate logic, but nontermina-

tion is a bit tougher. In the original logic for algebraic effects and handlers [38], the logic is

The basis for the principle of computational induction is the observation that evaluating

a computation results either in a value return, operation call, or nontermination.

5.5.1 Induction principle

The theorem basis for the principle of computational induction is the observation that evaluating

a computation results either in a value return, operation call, or nontermination.

Assume we have a schema that takes a computation \( c \) and produces an admissible

formula, \( \varphi(c) \). To show that \( \varphi(c) \) holds for any computation \( c : A!Σ/Ε \) it suffices to show the following:

- It holds for any returned value of type \( A \).
- It holds for operation calls of any operation \( op : A_{op} \rightarrow B_{op} \in Σ \), where we assume
  that \( \varphi \) holds for all continuation resumptions, i.e. \( ∀y : B_{op}. \varphi(ky) \), with \( k \) being the continuation.
- It holds for a nonterminating computation.

The first two requirements are expressed naturally in the predicate logic, but nontermination

is a bit tougher. In the original logic for algebraic effects and handlers [38], the logic is

further extended with inequalities and nontermination is simulated as an operation. While

the approach is more thorough, we wish to remain in a simpler logic, and effect systems

further complicate the use of a nontermination operation, as it impacts term types. We

instead draw inspiration from the induction principle described in Bauer and Pretnar’s
denotational based approach [6]. In denotational semantics, all nonterminating computa-

tions are treated the same, so we argue that using any nonterminating computation for the

check results in a correct induction principle. We settle for using a simple recursive loop

let rec g _ : unit → C = g () in g ()

We first present the rule using a formula schema, as it is easier to follow. Because our

logic offers no direct quantification over computations, we directly instantiate the schema

in the conclusion at a chosen computation \( Γ \vdash c : A!Σ/Ε \).

\[
\begin{array}{c}
\frac{\Gamma \vdash \varphi(c \cdot_\Sigma/Ε) \quad \Gamma, x : A \vdash \varphi(\text{ret } x)}{
\Gamma, x : A_{op} \vdash B_{op} \rightarrow D \mid Ψ, (\forall y : B_{op}. \varphi(ky)) \vdash \varphi(op_{A_{op} \rightarrow B_{op}}(x; ky))}_{op : A_{op} \rightarrow B_{op} ∈ Σ}
\end{array}
\]

\[
\frac{\Gamma, x : A \mid Ψ \vdash \varphi(\text{let rec } g _ : \text{unit } \rightarrow A!Σ/Ε = g () \text{ in } g ())}{\Gamma \vdash \varphi(c)}
\]

The above rule clearly showcases the idea behind computational induction but has a few issues in the current setting. Formula schemas for computations are not part of the

language, which in turn makes it difficult to check that all possible resulting formulae are

well formed. It is even more difficult to check that the resulting predicate is admissible.

We therefore propose an equivalent formulation which relies on functions. Any computa-

tion \( c \) can be rewritten as the equivalent \( (\text{fun } _ \rightarrow \text{e } c ()) \). We can express \( ∀c : C. \varphi(c) \) as

\( ∀f : \text{unit } \rightarrow C. \varphi(f ()) \) with the benefit that universal quantification over values exists in

58
the logic.

\[
\text{INDUCTION} \\
\text{\quad } f \text{ is admissible in } \varphi \quad \Gamma, x : A \vdash \varphi[f \mapsto (\text{fun } \_ \mapsto \text{ret } x)] \\
\text{\quad } \Gamma, x : A_{op}, k : B_{op} \rightarrow D_{op} \mid \Psi, (\forall y : B_{op}, \varphi[f \mapsto (\text{fun } \_ \mapsto k y)]) \\
\text{\quad } \vdash \varphi[f \mapsto (\text{fun } \_ \mapsto op_{A_{op} \rightarrow B_{op}} (x; y, k y))]) \\
\text{\quad } \Gamma \mid \Psi \vdash \varphi[f \mapsto (\text{fun } \_ \mapsto \text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g ())] \\
\text{\quad } \Gamma \mid \Psi \vdash \forall (f : \text{unit} \rightarrow A!\Sigma/E). \varphi
\]

Well-formedness of \( \varphi \) follows from side conditions; we just need to ensure admissibility of the formula \( \varphi \). We say that \( f \) is admissible in \( \varphi \) if any of the following holds:

\( \diamond \) the variable \( f \) does not appear in \( \varphi \),

\( \diamond \) \( \varphi \) is of form \( \top, \bot, \varphi_1 \land \varphi_2 \), or \( \varphi_1 \lor \varphi_2 \), where in the last two cases \( f \) is admissible in \( \varphi_1 \) and \( f \) is admissible in \( \varphi_2 \),

\( \diamond \) \( \varphi \) is of form \( \varphi_1 \Rightarrow \varphi_2 \) where \( f \) does not occur in \( \varphi_1 \) and \( f \) is admissible in \( \varphi_2 \),

\( \diamond \) \( \varphi \) is of form \( \forall x. \varphi' \) where \( x \neq f \) and \( f \) is admissible in \( \varphi' \),

\( \diamond \) \( \varphi \) is of form \( v \equiv_A v', c \equiv_C c', \text{ or } h \equiv_{\Sigma=\Sigma} h' \).

These restrictions result in admissible formulae, as required by the computational induction principle.

The predicate logic is poorly equipped for relating different nonterminating computations. Let us denote \( \text{loop} := (\text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g ()) \) and consider the following:

\[
\text{\quad } \text{do } x \leftarrow \text{loop in ret } x \equiv \text{loop}
\]

Even though both computations loop forever, they are syntactically different, so we are stuck. We do not attempt to solve the general problem of relating nonterminating computations, but since the induction rule directly injects \( \text{loop} \) into \( \varphi \), we expect \( \text{loop} \) to be the usual source of nontermination. We therefore hardcode the destructive propagation of nontermination for \( \text{loop} \) for \text{do} and \text{handle}, which are the only constructs that allow evaluation of subterms.

\[
\Gamma \mid \Psi \vdash \text{do } x \leftarrow (\text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g ()) \text{ in } c \\
\equiv_C \text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g () \\
\Gamma \mid \Psi \vdash \text{with } v \text{ handle (let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g ()) \\
\equiv_C \text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } g ()
\]

5.5.2 Properties

**Lemma 5.5.2.**

\( \diamond \) For well-typed values \( \Gamma \vdash v : A \), we can show \( \Gamma \mid . \vdash v \equiv_A v \).

\( \diamond \) For well-typed computations \( \Gamma \vdash c : C \), we can show \( \Gamma \mid . \vdash c \equiv_C c \).

\( \diamond \) For well-typed operation cases \( \Gamma \vdash h : \Sigma \Rightarrow D \), we can show \( \Gamma \mid . \vdash h \equiv_{\Sigma=\Sigma} h \).

**Proof (formalised).** The proof proceeds by simple induction on the derivation of typing. \( \square \)

**Lemma 5.5.3** (Hypothesis weakening). Assume we have \( \Gamma \vdash \Psi' : \text{hyp} \), with \( \Psi \subseteq \Psi' \). If we can show \( \Gamma \mid \Psi \vdash \varphi \), then we can also show \( \Gamma \mid \Psi' \vdash \varphi \).

**Proof (formalised).** Done by straightforward induction on derivation of \( \Gamma \mid \Psi \vdash \varphi \).\( \square \)
Corollary 5.5.4. Reflexivity can be used at an arbitrary set of hypotheses by combining Lemma 5.5.2 and Lemma 5.5.3.

Lemma 5.5.5. This is the continuation of Lemma 4.3.3. Assume we have two well-typed contexts for which $\Gamma' \leq \Gamma$ holds.

- If $\Gamma \vdash \varphi : \text{form}$ holds, then we can show $\Gamma' \vdash \varphi : \text{form}$.
- If $\Gamma \vdash \Psi : \text{hyp}$ holds, then we can show $\Gamma' \vdash \Psi : \text{hyp}$.
- If $\Gamma \mid \Psi \vdash \varphi$ holds, then we can show $\Gamma' \mid \Psi \vdash \varphi$.

Proof (formalised). We prove all three by mutually recursive induction with Lemma 4.3.3.

Lemma 5.5.6 (Substitution). Assume that we have a well-typed value $(\Gamma_1, \Gamma_2) \vdash v : B$.

- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash \varphi : \text{form}$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash \varphi \left[ x \mapsto v \right] : \text{form}$.
- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash \Psi : \text{hyp}$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash \Psi \left[ x \mapsto v \right] : \text{form}$.
- If we have $(\Gamma_1, x : B, \Gamma_2) \mid \Psi \vdash \varphi$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash \varphi \left[ x \mapsto v \right]$.

Lemma 5.5.7. Assume that we have a pair of related values $(\Gamma_1, \Gamma_2) \mid \Psi \vdash v \equiv_B v'$.

- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash u : A$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash u \left[ x \mapsto v \right] \equiv_A u \left[ x \mapsto v' \right]$.
- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash c : C$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash c \left[ x \mapsto v \right] \equiv_{\subseteq} c \left[ x \mapsto v' \right]$.
- If we have $(\Gamma_1, x : B, \Gamma_2) \vdash h : \Sigma \Rightarrow \Sigma$, then we can show $(\Gamma_1, \Gamma_2) \mid \Psi \vdash h \left[ x \mapsto v \right] \equiv_{\Sigma \Rightarrow \Sigma} h \left[ x \mapsto v' \right]$.

Remark 5.5.8. We can show results similar to a weaker version of Corollary 5.4.11, where the set of hypotheses is empty.

5.5.3 Examples

Example 5.5.9. A nice introductory example to induction is showing that the rule $\eta$Do can be proven within the predicate logic.

The formula we prove by induction is

$$\Gamma \mid \Psi \vdash \text{do } x \leftarrow c \text{ in } \text{ret } x \equiv_{\subseteq} c^{\eta\text{Do}}$$

The formula we prove by induction is

$$\Gamma \mid \Psi \vdash \forall f : \text{unit} \rightarrow A!\Sigma/E. (\text{do } x \leftarrow f () \text{ in ret } x \equiv_{A!\Sigma/E} f ())$$

Return: We start with the function that immediately returns a value $(\text{fun } \_ \mapsto \text{ret } x')$, with $x' : A$ added to the context.

$$\text{do } x \leftarrow (\text{fun } \_ \mapsto \text{ret } x') () \text{ in ret } x \equiv_{A!\Sigma/E} (\text{fun } \_ \mapsto \text{ret } x') ()$$
After reducing applications on both sides we obtain

\[
\text{do } x \leftarrow \text{ret } x' \text{ in } x \equiv_{\Sigma \oplus E} \text{ret } x',
\]

and the proof is completed by \( \beta \text{DoRet} \).

**Nontermination:** This time we substitute \( f \) with \( (\text{fun } _ \mapsto \rightarrow \text{loop}) \) where we shorten \( \text{loop} = \text{let rec } g _ \ = g () \in g () \). As with the return case, we first reduce the function applications.

\[
\text{do } x \leftarrow \text{loop in } x \equiv_{\Sigma \oplus E} \text{loop}
\]

This is a clear example of the need for \( \text{DoLoop} \), which equates the two nonterminating computations.

**Operations:** We choose an arbitrary operation \( \text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma \) and substitute \( f \) with \( (\text{fun } _ \mapsto \rightarrow \text{op}(x; y.k \ y)) \). This time, we also have an additional hypothesis:

\[
\psi_{IH} : \forall y : B_{\text{op}}, \text{do } x \leftarrow (\text{fun } _ \mapsto \rightarrow k \ y) () \in x \equiv_{\Sigma \oplus E} (\text{fun } _ \mapsto \rightarrow k \ y) ()
\]

Before we start with the proof we clean up the hypothesis, so it is easier to see when and how we use it.

\[
\psi_{IH} : \forall y : B_{\text{op}}, \text{do } x \leftarrow k \ y \text{ in } x \equiv_{\Sigma \oplus E} k \ y
\]

We now return to the proof of the operation case. After reducing the applications, we complete the proof by using the induction hypothesis.

\[
\text{do } x \leftarrow (\text{fun } _ \mapsto \rightarrow \text{op}(x; y.k \ y)) () \text{ in } x
\]

\[
\equiv_{\Sigma \oplus E} \langle \text{reduce application} \rangle
\]

\[
\text{do } x \leftarrow \text{op}(x; y.k \ y) \text{ in } x
\]

\[
\equiv_{\Sigma \oplus E} \langle \beta \text{DoOp} \rangle
\]

\[
\text{op}(x; y.\text{do } x \leftarrow k \ y \text{ in } x)
\]

\[
\equiv_{\Sigma \oplus E} \langle \text{use } \psi_{IH} \text{ in subterm with IsHYP and } \forall \text{El} \rangle
\]

\[
\text{op}(x; y.k \ y)
\]

\[
\equiv_{\Sigma \oplus E} \langle \text{reverse } \beta \text{App} \rangle
\]

\[
(\text{fun } _ \mapsto \rightarrow \text{op}(x; y.k \ y))(())
\]

**Changing to correct form:** The formula we obtained by induction works for functions, while the \( \eta \text{Do} \) uses computations. The precise formulation is achieved by using \( \forall \text{El} \) to replace \( f \) with \( (\text{fun } _ \mapsto \rightarrow c) \) and then reducing the trivial applications.

**Example 5.5.10.** Induction allows us to type a useful handler for nondeterminism, which returns the largest possible resulting value in a nondeterministic program.

\[
\Sigma_{\oplus} := \{ \text{Choose : unit } \rightarrow \text{bool} \}
\]

This example is restricted to a subset of the nondeterminism theory, the equation \( \text{COMM} \).

\[
\text{Choose}((); y.\text{if } y \text{ then } z_1 () \text{ else } z_2 ()) \sim \text{Choose}((); y.\text{if } y \text{ then } z_2 () \text{ else } z_1 ())
\]

Assume we can define a maximum function \( \text{max} : \text{int } \rightarrow \text{int } \rightarrow (\text{int}!\{\}/\{\}) \) for which we can also show commutativity of arguments.

\[
\forall x : \text{int}. \forall y : \text{int. } (\text{max } x \ y \equiv_{\text{int}!\{\}/\{\}} \text{max } y \ x)
\]
The operation cases we are interested in, denoted by $h_{\text{max}}$, return the largest of the two possibilities of \texttt{Choose}.

$$h_{\text{max}} := \{\texttt{Choose}(\_; k) \mapsto \text{do } x_1 \leftarrow (k \text{ true}) \text{ in } \text{do } x_2 \leftarrow (k \text{ false}) \text{ in } \text{max } x_1 \ x_2\}$$

We proceed to show $\Gamma \vdash h_{\text{max}} : \Sigma_\oplus \Rightarrow \text{int!}\{\}/\{}$ respects \{COMM\}, by first reducing the left side of the implicitly handled equation.

$$\begin{align*}
do x_1 \leftarrow (\text{fun } y \mapsto \text{if } y \text{ then } z_1 () \text{ else } z_2 ()) \text{ true in } \\
do x_2 \leftarrow (\text{fun } y \mapsto \text{if } y \text{ then } z_1 () \text{ else } z_2 ()) \text{ false in } \\
\text{max } x_1 \ x_2
&\equiv_{\text{int!}\{\}/\{}} \text{ (reduce applications) } \\
do x_1 \leftarrow (\text{if } \text{true then } z_1 () \text{ else } z_2 ()) \text{ in } \\
do x_2 \leftarrow (\text{if } \text{false then } z_1 () \text{ else } z_2 ()) \text{ in } \\
\text{max } x_1 \ x_2
&\equiv_{\text{int!}\{\}/\{}} \text{ (reduce conditionals) } \\
do x_1 \leftarrow z_1 () \text{ in } \text{do } x_2 \leftarrow z_2 () \text{ in } \text{max } x_1 \ x_2
\end{align*}$$

We reduce the right side of the equation in a similar manner and end up with an equation with no obvious reductions left.

$$\begin{align*}
do x_1 \leftarrow z_1 () \text{ in } \text{do } x_2 \leftarrow z_2 () \text{ in } \text{max } x_1 \ x_2
&\equiv_{\text{int!}\{\}/\{}} \\
do x_1 \leftarrow z_2 () \text{ in } \text{do } x_2 \leftarrow z_1 () \text{ in } \text{max } x_1 \ x_2
\end{align*}$$

We are saved by the type $\text{int!}\{\}/\{}$ at which the equality is required. While computations do not commute in general, the evaluation order is not important when working with pure computations. To that end, we prove PureComm:

$$k : \text{int} \to \text{int} \to (\text{int!}\{\}/\{}). \Gamma \vdash \text{do } x_1 \leftarrow f () \text{ in } \text{do } x_2 \leftarrow g () \in k \ x_1 \ x_2 \equiv \text{do } x_2 \leftarrow g () \text{ in } \text{do } x_1 \leftarrow f () \in k \ x_1 \ x_2$$

For the purposes of readability, we move the proof of PureComm to Example 5.5.11. We use PureComm on the right side of the equation. This is done through a double \texttt{VEL}, where we instantiate $f$ to $z_1$ and $g$ to $z_2$, followed by a substitution $[k \mapsto \text{max}]$. After applying PureComm we end up having to prove

$$\begin{align*}
do x_1 \leftarrow z_1 () \text{ in } \text{do } x_2 \leftarrow z_2 () \text{ in } \text{max } x_1 \ x_2
&\equiv_{\text{int!}\{\}/\{}} \\
do x_2 \leftarrow z_1 () \text{ in } \text{do } x_1 \leftarrow z_2 () \text{ in } \text{max } x_1 \ x_2
\end{align*}$$

To avoid confusion around names, we rename the bound variables on the right side by switching the names of $x_1$ and $x_2$. 

$$\begin{align*}
do x_1 \leftarrow z_1 () \text{ in } \text{do } x_2 \leftarrow z_2 () \text{ in } \text{max } x_1 \ x_2
&\equiv_{\text{int!}\{\}/\{}} \\
do x_1 \leftarrow z_1 () \text{ in } \text{do } x_2 \leftarrow z_2 () \text{ in } \text{max } x_2 \ x_1
\end{align*}$$

At the beginning of our example we assumed we have a proof that the order of arguments for $\text{max}$ is not important, which we now use to conclude the proof.
Example 5.5.11. We show PureComm, which states that at type \(\text{int}!\{}/\{\}\), we are allowed to switch the order of computation evaluation.

\[
k : \text{int} \to \text{int} \to (\text{int}!\{}/\{\}) | . \to (\forall f, g : \text{unit} \to \text{int}!\{}/\{\}).
\]

\[
\text{do } x_1 \leftarrow f () \text{ in do } x_2 \leftarrow g () \text{ in } k x_1 x_2 \equiv \text{do } x_2 \leftarrow g () \text{ in do } x_1 \leftarrow f () \text{ in } k x_1 x_2
\]

The proof proceeds by Induction for \(f\). Thanks to the pure type, we only check value returns and nontermination. For the case where \(f\) is of shape \((\text{fun } _\to \to \text{ret } x)\), we start with \(\forall \text{In}\) to remove the quantifier \(\forall g\) and move \(g\) to the context. The rest is straightforward.

\[
\text{do } x_1 \leftarrow (\text{fun } _\to \to \text{ret } x) () \text{ in do } x_2 \leftarrow g () \text{ in } k x_1 x_2
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \beta\text{APP} \text{ in subterm} \rangle
\]

\[
\text{do } x_1 \leftarrow \text{ret } x \text{ in do } x_2 \leftarrow g () \text{ in } k x_1 x_2
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \beta\text{DoRet} \rangle
\]

\[
\text{do } x_2 \leftarrow g () \text{ in } k x x_2
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \text{reverse } \beta\text{DoRet} \text{ in } k x x_2 \rangle
\]

\[
\text{do } x_2 \leftarrow g () \text{ in } k x \text{ in } \text{do } x_1 \leftarrow (\text{fun } _\to \to \text{ret } x) () \text{ in } k x_1 x_2
\]

We denote \(\text{loop} := \text{let rec } g’ _\_ = g’ () \text{ in } g’ ()\) and proceed with the case where \(f\) enters an infinite loop. We again start with \(\forall \text{IN}\) to obtain an equation formula. After reducing applications on both sides, we end up with

\[
\text{do } x_1 \leftarrow \text{loop } \text{in do } x_2 \leftarrow g () \text{ in } k x_1 x_2
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \beta\text{APP} \rangle
\]

\[
\text{do } x_2 \leftarrow g () \text{ in do } x_1 \leftarrow \text{loop } \text{in } k x_1 x_2
\]

By using DoLoop, we can reduce both sides a little further.

\[
\text{loop } \equiv_{\text{int}!\{}/\{\}} \text{do } x_2 \leftarrow g () \text{ in } \text{loop}
\]

We use \(\forall \text{El}\) to switch back to a universally quantified formula.

\[
\forall g. \ (\text{loop } \equiv_{\text{int}!\{}/\{\}} \text{do } x_2 \leftarrow g () \text{ in } \text{loop})
\]

This form is suitable for Induction on \(g\), where both possibilities for \(g\) can be shown fairly simply.

\[
\text{do } x_2 \leftarrow (\text{fun } _\to \to \text{ret } x) () \text{ in } \text{loop}
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \beta\text{APP} \rangle
\]

\[
\text{do } x_1 \leftarrow \text{ret } x \text{ in } \text{loop}
\]

\[
\equiv_{\text{int}!\{}/\{\}} \langle \beta\text{DoRet} \rangle
\]

\[
\text{loop}
\]

Example 5.5.12. We conclude the chapter with a more complex example. Assume we are tasked with creating a simple library for working with generators. When using generators that produce elements of type \(\text{A}\), the user is provided with the signature

\[
\Sigma_{\text{gen}} := \text{Next} : \text{unit} \to \text{A} \text{ option}
\]
The generators may be finite, meaning that at some point they start returning None. The behaviour of generators needs to be consistent if we hope for possible code optimisations. We prescribe that after Next returns None, all further invocations return None as well, which allows us to treat the first unsuccessful call as a signal that the generator has run dry. With $E_{\text{gen}}$, we denote the equation

$$\left( \text{Next}(); \ y . \ \text{match} \ y \ with \right)$$

$$\left( \begin{array}{l}
\text{Some} \ x \mapsto \text{Next}(); \ y' . \ y' \n \text{None} \mapsto \ z \ \text{None}
\end{array} \right) \sim \text{Next}(); \ y . \text{Next}(); \ y' . \ y'$$

Branching allows us to write conditional equations, where we specify in what circumstances the behaviour occurs. Should the first call return a value Some $v$, we generate another element and pass it to the program. In the case of None, the equation tells us that calling Next for a second time is futile as the generated element will be None. This might be clearer when the equation is written in sugared form.

$$\left( \begin{array}{l}
\text{match} \ !\text{Next} () \ with \\
\text{Some} \ x \mapsto \ z (!!\text{Next} ()) \\
\text{None} \mapsto \ z \ \text{None}
\end{array} \right) \sim !\text{Next} (); \ z (!!\text{Next} ())$$

A simple handler that can be used for value generation simply feeds values from a list to the program. We need some way to reduce the list of elements; for instance, through mutable state. It is possible to have a generator handler that takes care of its own mutable state, but we try to avoid multitool handlers for the sake of composability. The decision is therefore to assume an external mutable-state theory with Get and Set.

1. let gen_from_lst = handler

   | effect Next () k ->
     let lst = !Get () in
     match lst with
     | [] -> k None
     | x :: xs -> do !Set xs in k (Some x)

The handler gen_from_lst is meant to be used together with a handler for mutable state, such as state_handler from Example 5.4.13.

The other benefit of using an external state is gaining access to its effect theory. However, as we have seen in Example 5.4.13, the state_handler does not actually implement the full state theory. Luckily, a weaker state theory $\Sigma_{\text{wstate}}$ suffices, which state_handler respects.

$$\text{Set}(s; \ _ \text{Get} (); \ s'.z \ s') \sim \text{Set}(s; \ _ \text{z} \ s) \quad (\text{SetGet})$$

$$\text{Get}((); \ s . \text{Set}(s; \ _ \text{z} \ s)) \sim \text{Get}(); \ s . z \ s \quad (\text{WeakGetSet})$$

The equation WeakGetSet can be derived from SetGet and GetGet, so using the standard state theory is also possible. The goal now is to show

$$\vdash \text{gen_from_list} : (\Sigma_{\text{gen}} \Rightarrow B!\Sigma_{\text{state}}/E_{\text{wstate}}) \text{ respects } E_{\text{gen}}.$$
We show it specifically for list, but the same approach works for sums and products as well. For an operation \( op \) we construct a computation \( c_{\text{match}} \):

\[
(\Gamma_1, l : A \text{ list}, \Gamma_2) \rightarrow op(v_{op}; y.\text{match} \ l \ \text{with} \ [ ] \rightarrow c_1 \ | \ x :: \ xs \mapsto c_2) : C.
\]

If we use \( \eta \text{ LIST} \) with a value \((\Gamma_1, \Gamma_2 \vdash v : A \ \text{list})\) and the computation \( c_{\text{match}} \), we obtain

\[
c_{\text{match}}[l \mapsto v] \equiv_C \text{match} \ v \ \text{with} \ [ ] \rightarrow (c_{\text{match}}[l \mapsto [ ]]_A) \ | \ x :: \ xs \mapsto (c_{\text{match}}[l \mapsto x :: \ xs]),
\]

which is expanded to

\[
op(v_{op}; y.\text{match} \ v \ \text{with} \ [ ] \rightarrow c_1 \ | \ x :: \ xs \mapsto c_2) = C
\]

\[
\text{match} \ v \ \text{with} \ [ ] \rightarrow \ op(v_{op}; y.\text{match} \ [ ]_A \ \text{with} \ [ ] \rightarrow c_1 \ | \ x :: \ xs \mapsto c_2) \ | \ x :: \ xs \mapsto \ op(v_{op}; y.\text{match} \ x :: \ xs \ \text{with} \ [ ] \rightarrow c_1 \ | \ x :: \ xs \mapsto c_2).
\]

By using \( \beta \text{MATCHNIL} \) and \( \beta \text{MATCHCONS} \), we end up with

\[
op(v_{op}; y.\text{match} \ v \ \text{with} \ [ ] \rightarrow c_1 \ | \ x :: \ xs \mapsto c_2) = C
\]

\[
\text{match} \ v \ \text{with} \ [ ] \rightarrow \ op(v_{op}; y.c_1) \ | \ x :: \ xs \mapsto \ op(v_{op}; y.c_2).
\]

We now return to proof of \((\cdot \vdash \text{gen \ from \ list} : \Sigma_{\text{gen}} \Rightarrow D \ \text{respects} \ \Sigma_{\text{gen}})\) when mapping to type \( D := B!\Sigma_{\text{state}}/E_{\text{wstate}} \). We skip ahead to the cleaned-up version of \( E_{\text{gen}} \) after \((\cdot \vdash \text{match} \ \text{with} \ [ ] \mapsto \ | \ x :: \ xs \mapsto \ ... \) to avoid getting distracted.

\[
\text{Get} (\cdot) ; s.\text{match} \ s \ \text{with} \ [ ] \mapsto
\]

\[
\text{Get} (\cdot) ; s'.\text{match} \ s' \ \text{with} \ [ ] \mapsto \ z \ \text{None}
\]

\[
x' :: x s' \mapsto \text{Set}(x s', \ _\cdot z \ \text{(Some} \ x'))
\]

\[
\text{Get} (\cdot) ; s.\text{match} \ s \ \text{with} \ [ ] \mapsto \ z \ \text{None}
\]

\[
x :: x s \mapsto \ ... .
\]

We are unable to focus solely on the equality of branches, because they are not actually equal. The crucial part is the result of the first \text{Get} call. Because we do not alter the state between \text{Get} calls, we expect the second call to return the same value. Equations from \( \Sigma_{\text{wstate}} \) only describe behaviour of consecutive calls, and because the result of the operation call is being matched, we are unable to push \text{Get} into the branches.

By using \( E_{\text{wstate}} \) we can introduce a “useless” call of \text{Set} that can then be pushed into the branches. For this, we have to inherit \text{WeakGetSet} through OOTB, which we do explicitly this time. Recall the full \text{WeakGetSet} equation.

\[
\cdot ; z : \text{state} \rightarrow \ast \vdash \text{Get} (\cdot) ; s.\text{Set}(s ; \_z \ s) \sim \text{Get} (\cdot) ; s.\text{z} \ s \quad (\text{WeakGetSet})
\]

To instantiate of \text{WeakGetSet}, we are required to specify a value for \( z \). The value is chosen in a way that allows us to use the instantiation in our proof.

\[
\text{match} \ s \ \text{with} \ [ ] \mapsto
\]

\[
c_{\text{inst}} := \text{Get} (\cdot) ; s'.\text{match} \ s' \ \text{with} \ [ ] \mapsto \ z \ \text{None}
\]

\[
x' :: x s' \mapsto \text{Set}(x s', \ _z \ \text{(Some} \ x'))
\]

\[
x :: x s \mapsto \ ...
\]

65
After cleaning up function applications, we end up with the desired equation.

\[
\text{Get}(\_); \text{s.Get}(s; \._z s))_{D}[z \mapsto (\text{fun } s \mapsto c_{\text{inst}})] \equiv_{D} \text{Get}(\_); \text{s.z s))_{D}[z \mapsto (\text{fun } s \mapsto c_{\text{inst}})]
\]

We use \text{CEQSYM} to flip the sides so we can use transitivity to reduce our proof term. In the following proof, we simply state that OOTB is used as a reduction, but the process involves all of the above.

We return to the proof of \text{respects} and try to reduce the branch for empty lists to \((z \text{ None})\). Since the branch \([x' :: xs' \mapsto \text{Set}(xs', _z (\text{Some } x'))]\) plays no role in the proof, we also replace it with \((\_ )\) in this part of the proof.

\[
\text{Get}(\_); \text{s.match } s \text{ with}
| [\_ ] \mapsto \text{Get}(\_); \text{s'.match } s' \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \text{Set}(xs', _z (\text{Some } x')))
| x :: xs \mapsto \ldots)
\]

\[
\equiv_{D} \text{ (OOTB used for WeakGetSet to add Get to outer Get )}
\]

\[
\text{Get}(\_); \text{s.match } s \text{ with}
| [\_ ] \mapsto \text{Get}(\_); \text{s'.match } s' \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \ldots)
| x :: xs \mapsto \ldots)
\]

\[
\equiv_{D} \text{ (push Set inside match )}
\]

\[
\text{Get}(\_); \text{s.match } s \text{ with}
| [\_ ] \mapsto \text{Set}(s; \_ \text{Get}(\_); \text{s'.match } s' \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \ldots))
| x :: xs \mapsto \text{Set}(s; \_ \ldots))
\]

\[
\equiv_{D} \text{ (OOTB used for SetGet to remove Get )}
\]

\[
\text{Get}(\_); \text{s.match } s \text{ with}
| [\_ ] \mapsto \text{Set}(s; \_ \text{match } s \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \ldots)
| x :: xs \mapsto \text{Set}(s; \_ \ldots))
\]

\[
\equiv_{D} \text{ (pull Set from match )}
\]

\[
\text{Get}(\_); \text{s.Set}(s; \_ \text{match } s \text{ with}
| [\_ ] \mapsto \text{match } s \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \ldots)
| x :: xs \mapsto \ldots)
\]

\[
\equiv_{D} \text{ (OOTB used for WeakGetSet to remove Set )}
\]

\[
\text{Get}(\_); \text{s.match } s \text{ with}
| [\_ ] \mapsto \text{match } s \text{ with } [\_ ] \mapsto z \text{ None}| x' :: xs' \mapsto \ldots)
| x :: xs \mapsto \ldots)
\]

We have eliminated the second call of \text{Get}, but a duplication of match remains. We
again make use of \( \eta \text{LISTMATCH} \), but this time with \( s \) as the list value and the computation

\[
\text{match } l \text{ with } \\
| [] \mapsto \text{match } l \text{ with } [ ] \mapsto z \text{ None } | x' :: xs' \mapsto \text{Set}(xs', _{-}z (\text{Some } x')) \\
| x :: xs \mapsto \ldots
\]

We reduce the result by using \( \beta \text{MATCHNIL} \) and \( \beta \text{MATCHCONS} \) to produce

\[
\text{match } s \text{ with } \\
| [] \mapsto \\
\text{match } s \text{ with } \\
| [] \mapsto z \text{ None } \quad \equiv \quad | [] \mapsto z \text{ None } \quad | x :: xs \mapsto \ldots \\
| x' :: xs' \mapsto \text{Set}(xs', _{-}z (\text{Some } x')) \\
| x :: xs \mapsto \ldots
\]

We use the above equality to remove the \textit{match} duplication inside of \texttt{Get}, and the proof for \texttt{respects} is complete.
Chapter 6

Language semantics

The effect-theory system is built to guarantee handler correctness, which we prove by using denotational semantics. We split the construction of denotational semantics into two phases. The first part is concerned with mathematical interpretation of terms while ignoring equations entirely. This is then remedied in the second part, where we use equations to construct partial equivalence relations. Because effect theories do not affect the semantics of types and terms, all well-typed handlers receive a denotation as opposed to the original treatment [37].

Chapter 6 describes how to interpret types and terms as mathematical objects. In our previous work [28] we described an approach using sets, which has to be transferred to domain theory to account for recursion. The inclusion of subtyping results in nonunique typing derivations, and we need to ensure coherence of denotational semantics. In Section 6.3 we draw inspiration from the work of Bauer and Pretnar [6] and introduce skeleton semantics, which are coherent by construction. We then follow the approach of Reynolds [40] and link the two versions of semantics to prove that the denotational semantics is coherent.

In Chapter 7, we relate the mathematical interpretations of terms by a partial equivalence relation that stems from equations. The relation only relates correct handlers, which corresponds to the approach of Plotkin and Pretnar [37], where only correct handlers receive a denotation. With this relation we devise a natural condition on when a logic is sound and show that the equational and predicate logics of Chapter 5 are sound. We then show that the type system of EEFF, coupled with a sound logic, guarantees handler correctness.

Denotations of types, terms, and theories are collected in Appendix B.

6.1 Tools from domain theory

Using domains is a standard way of dealing with recursion in denotational semantics. We take a look at some common constructions and properties, which are encountered in definitions and proofs throughout the chapter. Readers unfamiliar with domain theory can consult Domains and Lambda-Calculi [5].

A predomain is a partially ordered set which contains the suprema of ascending sequences (chains). The order relation on the set $D$ is denoted by $\leq_D$ and for a chain $x_0 \leq_D x_1 \leq_D \ldots$ we sometimes use the shorter notation $(x_i)_i$. We use $\bigvee_i x_i$ to denote the supremum of the chain $(x_i)_i$. Domains are predomains with a least element, which is denoted by $\bot_D$. The subscripts of $\leq_D$ and $\bot_D$ are omitted when easily inferred.

A map $f$ from the predomain $D$ to a predomain $E$ is monotone if $d_1 \leq_D d_2$ implies that $f(d_1) \leq_E f(d_2)$. We say that $f$ commutes with the suprema of chains if for any chain
\((d_i)_i\), it holds that \(f(\bigvee_i d_i) = \bigvee_i f(d_i)\).

A map between predomains is \textit{continuous} if it is monotone and commutes with the suprema of chains. We use the notation \(D \rightarrow E\) for sets of continuous maps from \(D\) to \(E\). A map is \textit{strict} if it maps \(\bot_D\) to \(\bot_E\) and sets of strict maps are denoted by \(D \rightarrow E\).

The notation for anonymous functions is \((\lambda x \in X. f)\) where \(f\) marks the function body, which may depend on the variable \(x\). When the domain of the function is known, we shorten the notation to \((\lambda x.f)\).

\textbf{Constructions on predomains and domains}

If \(D\) is a predomain and \(E\) a domain, then \(D \rightarrow E\) is also a domain with pointwise ordering. This means that \(f \leq_{D \rightarrow E} g\) if and only if \(\forall d \in D. f(d) \leq_E g(d)\). From the order it follows that for a chain \(f_0 \leq f_1 \leq \ldots\) we have \((\bigvee_i f_i)(d) = \bigvee_i f_i(d)\). Strict continuous maps are a subdomain of continuous maps.

A Cartesian product of predomains \(D \times E\) is a predomain, where \((d_1, e_1) \leq_{D \times E} (d_2, e_2)\) if and only if \(d_1 \leq_D d_2\) and \(e_1 \leq_E e_2\). For the suprema we can show \(\bigvee_i (a_i, b_i) = (\bigvee_i a_i, \bigvee_i b_i)\).

A disjoint union of two predomains \(D + E\) is also a predomain, where \(\iota_1(d_1) \leq_{D + E} \iota_1(d_2)\) if and only if \(d_1 \leq_D d_2\), and symmetrically \(\iota_2(e_1) \leq_{D + E} \iota_2(e_2)\) if and only if \(e_1 \leq_E e_2\). Elements from different components cannot be compared. Chains in \(D + E\) are either of form \(\iota_1(d_0) \leq \iota_1(d_1) \leq \ldots\), where \((d_i)_i\) is a chain in \(D\), or \(\iota_2(e_0) \leq \iota_2(e_1) \leq \ldots\), where \((e_i)_i\) is a chain in \(E\). In both cases suprema distribute over \(\iota_1\) and \(\iota_2\) respectively, so \(\bigvee_i \iota_1(d_i) = \iota_1(\bigvee i d_i)\) and \(\bigvee_i \iota_2(e_i) = \iota_2(\bigvee i e_i)\).

If \(D\) is a predomain, then \(D^*\) is the predomain of finite sequences of elements of \(D\), where \(d_0, d_1, \ldots, d_n \leq D^*\) \(e_0, e_1, \ldots, e_m \leq D^*\) if and only if \(n = m\) and \(d_i \leq_D e_i\) for all \(i = 0, \ldots, n\). Sequences of differing lengths cannot be compared. The short notation for \(d_0, d_1, \ldots, d_n\) is \((d_i)_{i=0}^n\) and \(\epsilon\) denotes the empty sequence. The notation \(d :: ds\) stands for the sequence \(ds\), extended with \(d\) as the first element. From the definition of \(\leq_{D^*}\), it follows that if \((d_{i,0})_{i=0}^n \leq (d_{i,1})_{i=0}^n \leq (d_{i,2})_{i=0}^n \leq \ldots\) is a chain in \(D^*\), then the lengths of all sequences in the chain must be equal. Furthermore, we obtain component-wise chains \((d_{k,i})_k\) for each \(i = 0 \ldots n\), where \(n\) is the length of sequences in the chain. It follows that suprema are component-wise, meaning that \(\bigvee_k (d_{k,i})_{i=0}^n = (\bigvee_k d_{k,i})_{i=0}^n\).

\textbf{Computation domains}

To construct domains for representing computations, we rely on results of Bauer and Pretnat [6]. Let \(A\) be a predomain and \(I\) an index set, together with collections of predomains \((A_i)_{i\in I}\) and \((B_i)_{i\in I}\). For the type \(A^I/\Sigma\) we want the index set to be \(\Sigma\) and \((A_{op})_{op\in\Sigma}\) and \((B_{op})_{op\in\Sigma}\) the domains for types of operations in the signature. We want a domain that satisfies the domain equation

\[D = \left( A + \prod_{i\in I} A_i \times (B_i \rightarrow D) \right)_\bot.\]

Such a domain \(T\) exists [6], and its elements are (possibly infinite) trees. Their leaves are either tagged with \(\bot\), or tagged as \(\text{in}_{\mathcal{A}_1}(a)\), where \(a\) is an element of \(A\). The nodes are tagged as \(\text{in}_{\mathcal{A}_1}(a, k)\) with an index \(k\), an element \(a \in A_i\), and a function \(k\) for branching on elements of \(B_i\). Such a solution is denoted by \(T_I(A, (A_i)_{i\in I}, (B_i)_{i\in I})\) and has the order:

\[
\begin{align*}
    c_1 \leq_T c_2 & \iff c_1 = \bot \\
    & \lor (c_1 = \text{in}_{\mathcal{A}_1}(a_1) \land c_2 = \text{in}_{\mathcal{A}_1}(a_2) \land a_1 \leq_A a_2) \\
    & \lor (c_1 = \text{in}_i(a_1, k_1) \land c_2 = \text{in}_i(a_2, k_2) \land a_1 \leq_{A_i} a_2 \land \forall b \in \prod B_i. k_1(b) \leq_T k_2(b)).
\end{align*}
\]

70
Chains $c_0 \leq_T c_1 \leq_T \ldots$ have only three possible shapes:

- the chain is constantly $\bot$,
- it may start as a chain of $\bot$, but from some index onwards elements are of shape $c_j = \text{in}_{\text{val}}(a_j)$ and $(a_j)_j$ is a chain in $A$,
- it may start as a chain of $\bot$, but from some index onwards elements are of shape $c_j = \text{in}_1(a_j, k_j)$. Elements $(a_j)_j$ form a chain in $A_i$, and for any $b \in B_i$ it holds that $(k_j(b))_j$ is a chain in $T$.

The solution $T$ is minimal in the sense that it admits a recursion and an induction principle. The recursion principle states that for any domain $D$, a continuous map $f_{\text{val}} : A \to D$, and a collection of maps $f_i : (A_i \times B_i \to D) \to D$, there exists a unique strict continuous map $f : T \to D$, such that

$$f(\text{in}_{\text{val}}(a)) = f_{\text{val}}(a)$$

$$f(\text{in}_i(a, k)) = f_i(a, (f \circ k)).$$

The induction principle works for admissible predicates. A predicate $\varphi$ is admissible if $\varphi(\bot)$ holds, and if for any chain $(c_i)_i$, for which $\forall i. \varphi(c_i)$, it also holds that $\varphi(\bigvee_i c_i)$. The induction principle states that in order to show $\forall c \in T. \varphi(c)$, it is enough to show the following:

- for any $a \in A$ we can show $\varphi(\text{in}_{\text{val}}(a))$,
- for every $i \in I, a \in A_i$, and $k \in B_i \to T$, we can show $\varphi(\text{in}_i(a, k))$ under the assumption that $\forall b \in B_i. \varphi(k(b))$ holds.

6.2 Type and term semantics

6.2.1 Semantics of types

Well-formed types are interpreted as either predomains or domains. We always shorten $\llbracket A : \text{vtype} \rrbracket$ to $\llbracket A \rrbracket$ as the judgement provides no further information.

Value types are assigned predomains, which in the case of function and handler types are also domains. For handlers, we restrict the function domain to strict functions.

- $\llbracket \text{unit} \rrbracket = \{\star\}$
- $\llbracket \text{int} \rrbracket = \mathbb{N}$
- $\llbracket \text{empty} \rrbracket = \emptyset$
- $\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$
- $\llbracket A + B \rrbracket = \llbracket A \rrbracket + \llbracket B \rrbracket$
- $\llbracket A \text{ list} \rrbracket = \llbracket A \rrbracket^\ast$
- $\llbracket A \to C \rrbracket = \llbracket A \rrbracket \to \llbracket C \rrbracket$
- $\llbracket C \to D \rrbracket = \llbracket C \rrbracket \to \llbracket D \rrbracket$

Due to the inclusion of $\text{empty}$, we also need to consider empty functions $\emptyset \to X$. The empty function is uniquely determined by its codomain, and for each set $X$ we denote $\text{emptyf}_{\text{un}X} : \emptyset \to X$.

Computation types are interpreted as domains. Equations do not play a role in this part of denotational semantics and are therefore left out when constructing domains. The denotation of a signature is a map from a predomain to the computation domain described in Section 6.1.

$$\llbracket \Sigma \rrbracket X = T_{\Sigma}(X, (\llbracket A_{op} \rrbracket)_{op \in \mathcal{E}}, (\llbracket B_{op} \rrbracket)_{op \in \mathcal{E}})$$

$$\llbracket A_{\Sigma/\mathcal{E}} \rrbracket = \llbracket \Sigma \rrbracket \llbracket A \rrbracket$$

71
We also require structures to model operation cases. An interpretation \( \mathcal{H} \) of a signature \( \Sigma \) over a set \( Y \) is a family of continuous functions \( H_{op}: \prod A \times (\prod B) \rightarrow Y \) for each \( op: A \rightarrow B \in \Sigma \). We define the set of interpretations as

\[
\text{interp}_\Sigma(Y) = \prod_{op: A \rightarrow B \in \Sigma} (\prod A) \times (\prod B) \rightarrow Y.
\]

If \( Y \) is a domain, then \( \text{interp}_\Sigma(Y) \) is a predomain with \( H \leq H' \) if and only if for all \( op: A \rightarrow B \), we have \( H_{op} \leq H'_{op} \). For any signature \( \Sigma \) and any set \( X \), we define a free interpretation \( F_{X, \Sigma} \in \text{interp}_\Sigma(\prod \Sigma X) \) by

\[
(F_{X, \Sigma})_{op}(a, \kappa) = \text{in}_{op}(a, \kappa).
\]

For any interpretation \( H: \text{interp}_\Sigma(Y) \), we can lift a function \( f: X \rightarrow Y \) to a strict continuous function \( \text{lift}_H f: \prod \Sigma X \rightarrow Y \). The definition uses the recursion principle of \( \prod \Sigma X \) with the cases

\[
\begin{align*}
(\text{lift}_H f)(\text{in}_{\text{val}}(x)) &= f(x), \\
(\text{lift}_H f)(\text{in}_{\text{op}}(x, \kappa)) &= H_{op}(x, \text{lift}_H f \circ \kappa).
\end{align*}
\]

### 6.2.2 Semantics of subtyping

Subtyping is used to modify the type of a term without changing its meaning. Similarly, we interpret subtyping as a coercion \( \prod [A \leq A']: \prod [A] \rightarrow \prod [A'] \). In EEFF, subtyping only affects signatures, so value type coercions are mostly structural propagation.

\[
\begin{aligned}
\text{unit} \leq \text{unit} &= \text{id}_{\text{unit}} & \text{int} \leq \text{int} &= \text{id}_{\text{int}} & \text{empty} \leq \text{empty} &= \text{id}_{\text{empty}} \\
\prod A + B \leq \prod A' + B' &= \lambda x. \begin{cases} t_1(\prod A \leq A') & : x = t_1(a) \\ t_2(\prod B \leq B') & : x = t_2(b) \end{cases} \\
\prod A \times B \leq \prod A' \times B' &= \lambda (a, b). (\prod A \leq A') a \cdot (\prod B \leq B') b \\
\prod \text{list} \leq \prod \text{list} &= \lambda (a)_{i=0}^{n}. (\prod A \leq A') a_i \\
\prod A \rightarrow \prod C \leq \prod A' \rightarrow \prod C' &= \lambda f. (\prod C \leq C') f \circ (\prod A' \leq A) \\
\prod C \Rightarrow \prod D \leq \prod C' \Rightarrow \prod D' &= \lambda g. (\prod D \leq D') g \circ (\prod C' \leq C)
\end{aligned}
\]

The coercion \( \prod A!\Sigma/E \leq \prod A'!\Sigma'/E' \) also needs to correct types of operations. Signature coercions result in interpretations, \( \prod [\Sigma \leq \Sigma'] X \in \text{interp}_\Sigma(\prod \Sigma' X) \), and for every operation \( op: A \rightarrow B \in \Sigma \) for which \( op: A' \rightarrow B' \in \Sigma' \), we define

\[
(\prod [\Sigma \leq \Sigma'] X)_{op}(x, \kappa) = \text{in}_{op}(A \leq A', \kappa \circ B' \leq B).
\]

We then define the semantics of subtyping on computations with a lift.

\[
\prod A!\Sigma/E \leq \prod A'!\Sigma'/E' = \text{lift}_{\Sigma \leq \Sigma'}(\lambda a. \text{in}_{\text{val}}(A \leq A') a)
\]

Since \( E \) plays no role in the denotation of terms, it also plays no role in denotations of subtyping.

Certain proofs rely on transitivity of the subtyping relation. We therefore show that the semantics of subtyping composes in the expected manner.
Lemma 6.2.1.

\(\exists [A' \leq A''] \circ [A \leq A'] = [A \leq A'']\)
\(\exists [C' \leq C''] \circ [C \leq C'] = [C \leq C'']\)

Proof. The proof proceeds by induction on derivation of \(A \leq A'\).

- The cases for base types, products, sums, and lists are structural and straightforward.
- We do the proof for function types, but the same approach works for handler types as well.

\[
\begin{align*}
\left[ A' \to C' \leq A'' \to C'' \right] & \circ \left[ A \to C \leq A' \to C' \right] \\
= (\lambda f . \left[ C' \leq C'' \right] \circ f \circ \left[ A'' \leq A' \right]) \circ (\lambda f . \left[ C \leq C' \right] \circ f \circ \left[ A' \leq A \right]) \\
= (\lambda f . \left[ C' \leq C'' \right] \circ \left[ C \leq C' \right] \circ f \circ \left[ A' \leq A \right] \circ \left[ A'' \leq A' \right]) \\
= (\lambda f . \left[ C \leq C'' \right] \circ f \circ \left[ A'' \leq A \right]) \\
= \left[ A \to C \leq A'' \to C'' \right]
\end{align*}
\]

Line three uses the induction hypotheses for sub-derivations.

- It is a bit trickier to show

\[
\left[ A'!\Sigma'/E' \leq A''!\Sigma''/E'' \right] \circ \left[ A!\Sigma/E \leq A'!\Sigma'/E' \right] = \left[ A!\Sigma/E \leq A''!\Sigma''/E'' \right].
\]

We show that both options have the same result when applied to an arbitrary argument \(c\) by using the induction principle from Section 6.1 for the argument \(c\). Lifts are strict and continuous, so the predicate \(\varphi(c) := (\left[ C' \leq C'' \right] \left[ C \leq C' \right] c) = \left[ C \leq C'' \right] c)\) is admissible.

- Assume that \(c = \text{in}_{va1}(a)\). From induction on subtyping derivation, we have induction hypotheses concerning type coercions between \(A, A', \text{and } A''\).

\[
\begin{align*}
\left[ A'!\Sigma'/E' \leq A''!\Sigma''/E'' \right] & \left[ A!\Sigma/E \leq A'!\Sigma'/E' \right] (\text{in}_{va1}(a)) \\
= \left[ A'!\Sigma'/E' \leq A''!\Sigma''/E'' \right] (\text{in}_{va1}(A \leq A') a) \\
= \text{in}_{va1}(A \leq A') a) \\
= \left[ A!\Sigma/E \leq A''!\Sigma''/E'' \right] (\text{in}_{va1}(a))
\end{align*}
\]

- Assume that \(c = \text{in}_{op}(a, \kappa)\) and \(op : A_{op} \to B_{op} \in \Sigma, op : A_{op}' \to B_{op}' \in \Sigma', op : A_{op}'' \to B_{op}'' \in \Sigma''\). The induction on the subtyping derivation gives us induction hypotheses for coercions between the types of the operation. We shorten the computation types to \(C, C', \text{and } C''\) for brevity. The IH from induction on structure of \(c\) is

\[
\forall b. \left[ C' \leq C'' \right] \left[ C \leq C' \right] (\kappa(b)) = \left[ C \leq C'' \right] (\kappa(b)),
\]

which is equivalent to

\[
\left[ C' \leq C'' \right] \circ \left[ C \leq C' \right] \circ \kappa = \left[ C \leq C'' \right] \circ \kappa.
\]
We get the result by using induction hypotheses.

\[
\begin{align*}
\langle C' \leq C'' \rangle (\langle C \leq C' \rangle (\in_{op}(a, \kappa))) \\
= (\langle C' \leq C'' \rangle (\in_{op}(\langle A_{op} \leq A'_{op} \rangle a, \langle C \leq C' \rangle \circ \kappa \circ \langle B'_{op} \leq B_{op} \rangle))) \\
= \in_{op}(\langle A_{op} \leq A'_{op} \rangle (\langle A_{op} \leq A'_{op} \rangle a), \\
\langle C' \leq C'' \rangle \circ \langle C \leq C' \rangle \circ \kappa \circ \langle B'_{op} \leq B_{op} \rangle \circ \langle B'_{op} \leq B_{op} \rangle) \\
= \in_{op}(\langle A_{op} \leq A'_{op} \rangle a, \langle C \leq C'' \rangle \circ \kappa \circ \langle B_{op} \leq B_{op} \rangle) \\
= (\langle C \leq C'' \rangle)(\in_{op}(a, \kappa))
\end{align*}
\]

\[\square\]

### 6.2.3 Semantics of values and computations

We construct term denotations on typing derivations \(\Gamma \vdash v : A\). The typing derivation of a term is not necessarily unique; for instance, we can arrive at \(\cdot \vdash () : \text{unit}\) through either of the following derivations:

\[
\begin{array}{cc}
\cdot \vdash () : \text{unit} & \cdot \vdash () : \text{unit} \\
\end{array}
\]

We nonetheless define denotational semantics for terms through typing derivations, and prove that the result is coherent (Section 6.4). This means that \(\llbracket \Gamma \vdash v : A \rrbracket\) will be the same mathematical object no matter which typing derivation we take for \(\Gamma \vdash v : A\).

The denotations of well-formed contexts are Cartesian products of domains. We again short
\(\cdot \vdash c : \text{ctx}\) to \([\cdot]\). If \(\eta \in \llbracket \Gamma \rrbracket\) and \(a \in \llbracket A \rrbracket\), we write \((\eta, a)\) for the element of \(\llbracket \Gamma, x : A \rrbracket\).

\[
\llbracket \cdot \rrbracket = \{\star\}
\]

\[
\llbracket \Gamma, x : A \rrbracket = \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket
\]

Typing judgements are interpreted as maps from \(\llbracket \Gamma \rrbracket\) to the appropriate (pre)domain.

\[
\begin{align*}
\llbracket \Gamma \vdash v : A \rrbracket & : \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket \\
\llbracket \Gamma \vdash c : C \rrbracket & : \llbracket \Gamma \rrbracket \to \llbracket C \rrbracket \\
\llbracket \Gamma \vdash h : \Sigma \Rightarrow D \rrbracket & : \llbracket \Gamma \rrbracket \to \interp_D(\llbracket D \rrbracket)
\end{align*}
\]

When the context and type can easily be inferred, we abbreviate the denotations of typing judgements to \(\llbracket v \rrbracket\), \(\llbracket c \rrbracket\), and \(\llbracket h \rrbracket\). In all following sections we assume \(\eta \in \llbracket \Gamma \rrbracket\), unless specified otherwise.

### Values and operation cases

The definition follows the recursive structure of type derivations. Variables are interpreted as projections to the correct component of \(\eta\), and the data constructors of the language are mapped to their mathematical counterparts.

\[
\begin{align*}
\llbracket \Gamma \vdash () : \text{unit} \rrbracket \eta & = \star \\
\llbracket \Gamma \vdash n : \text{int} \rrbracket \eta & = n \\
\llbracket \Gamma \vdash \text{Left}_{A \times B} v : A + B \rrbracket \eta & = t_1(\llbracket v \rrbracket \eta) \\
\llbracket \Gamma \vdash \text{Right}_{A \times B} v : A + B \rrbracket \eta & = t_2(\llbracket v \rrbracket \eta) \\
\llbracket \Gamma \vdash \text{Left} A \rrbracket \eta & = \varepsilon \\
\llbracket \Gamma \vdash \text{Right} A \rrbracket \eta & = \varepsilon
\end{align*}
\]

\[74\]
Functions and handlers are assigned continuous functions. For functions, the definition is structural:

\[ \Gamma \vdash (\text{fun } (x: A) \mapsto c) : A \rightarrow C \eta = \lambda a \in [A]. \Gamma, x : A \vdash c : C(\eta,a) \]

The denotation of a handler is the lifting of its return clause to the interpretation given by the operation clauses:

\[ \Gamma \vdash \text{handler } (\text{ret } (x : A) \mapsto c_r; h) : A ! / \Sigma / E \Rightarrow D \eta = \text{lift}_{h\emptyset}(\lambda a \in [A]. \Gamma, x \vdash c_r : C(\eta,a)) \]

Strictness follows from the definition of lift. Well-typed operation cases are assigned a family of functions from \( \text{interp}_\Sigma(D) \).

\[ (\Gamma \vdash \{ \text{op}_{A \rightarrow B}(x;k) \mapsto c_{op}\}_{op} : \Sigma \Rightarrow D \eta)_{op} = \]
\[ \lambda a \in [A_{op}]. \lambda k \in [B_{op} \rightarrow D]. \Gamma, x : A_{op}, k : B_{op} \vdash D \vdash c_{op} : D(\eta,a,k) \]

In the case of \( \text{TYPEV\text{SUBSUME}} \) for the subtype \( A \leq A' \) we use the appropriate coercion.

\[ \Gamma \vdash v : A' \eta = [A \leq A']((\Gamma \vdash v : A \eta)) \]

Computations

For value returns and operation calls, we have matching in \( \text{val}_{v1} \) and \( \text{in}_{op} \) constructors in \([C] \).

The derivation \( \text{TYPEOP} \) includes explicit subtyping that requires appropriate coercions. Assume that \( op \) is of type \( A' \rightarrow B' \) in the signature of \( C \).

\[ \Gamma \vdash \text{ret } v : C \eta = \text{in}_{\text{val}}([v] \eta) \]
\[ \Gamma \vdash \text{op}_{A \rightarrow B}(v; y.c) : C \eta = \text{in}_{\text{op}}([A \leq A']([v] \eta); \lambda b. \Gamma, y : B \vdash c : C(\eta, [B' \leq B \eta])) \]

For \text{absurd} we make use of the empty function \( \text{emptyfun}_{C1} \).

\[ \Gamma \vdash \text{absurd}_{C} v : C \eta = \text{emptyfun}_{C1}([v] \eta) \]

The interpretations of match statements depend on the denotation of the matched value.

\[ \Gamma \vdash \text{match } v \text{ with } (x,y) \mapsto c : C \eta = \]
\[ \Gamma, x : A, y : B \vdash c : C(\eta,a,b) \quad \text{(for } [v] \eta = (a,b)) \]

\[ \Gamma \vdash \text{match } v \text{ with Left } x \mapsto c_1 \mid \text{Right } y \mapsto c_2 : C \eta = \]
\[ \eta = \begin{cases} \Gamma, x : A \vdash c_1 : C(\eta,a) ; [v] \eta = \eta_1 a, & \text{if } [v] \eta = a \in \text{in}_{\text{val}}([v] \eta) \\ \Gamma, y : B \vdash c_2 : C(\eta,b) ; [v] \eta = \eta_2 b, & \text{if } [v] \eta = b \in \text{in}_{\text{val}}([v] \eta) \end{cases} \]

\[ \Gamma \vdash \text{match } v \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 : C \eta = \]
\[ \eta = \begin{cases} \Gamma \vdash c_1 : C \eta & \text{if } [v] \eta = \varepsilon, \\ \Gamma, x : A, xs : A \text{ list} \vdash c_2 : C(\eta,a_0)(a_{n-1}) ; [v] \eta = a_0, \ldots, a_n & \text{if } [v] \eta = c \in \text{in}_{\text{val}}([v] \eta) \end{cases} \]

Functions and handlers of \( EEFF \) are interpreted as mathematical functions, so we can apply them to the denotation of the argument.

\[ \Gamma \vdash v_1 v_2 : C \eta = ([v_1] \eta)([v_2] \eta) \]
\[ \Gamma \vdash \text{with } v \text{ handle } c : D \eta = ([v] \eta)([c] \eta) \]

75
The denotation of sequencing is constructed through lift. Whenever \( c_1 \) returns a value, we continue with \( c_2 \); so in the denotation we construct the effect tree \( \llbracket c_1 \rrbracket_{\eta} \), and then in the in\_val leaves, we continue with \( \llbracket c_2 \rrbracket \). Sequencing propagates operation calls, so we do not modify the in\_op nodes, which coincides with the free interpretation \( F_{\llbracket B \rrbracket_{\Sigma}} \).

\[
\llbracket \Gamma \vdash x \leftarrow c_1 \text{ in } c_2 : B!\Sigma/E \rrbracket_{\eta} = (\text{lift}_{F_{\llbracket B \rrbracket_{\Sigma}}} (\lambda a \in \llbracket A \rrbracket . [\llbracket \Gamma, x : A \vdash c_2 : B!\Sigma/E \rrbracket_{\eta}, a]))(\llbracket c_1 \rrbracket_{\eta})
\]

Recursive functions are the reason for switching to domains. The denotation for the \texttt{let rec} construct is

\[
\llbracket \Gamma \vdash \text{let rec } f x : A \to C = c_1 \text{ in } c_2 : D \rrbracket_{\eta} = \llbracket \Gamma, f : A \to C \vdash c_2 : D \rrbracket_{\eta}(\tilde{f})
\]

with \( \tilde{f} \) being the least fixed point of the function definition \( c_1 \).

\[
\tilde{f} = \mu f. \lambda a \in \llbracket A \rrbracket . [\llbracket \Gamma, x : A, f : A \to C \vdash c_1 : C \rrbracket_{\eta}, a, f]
\]

In the case of judgement \texttt{TypCSubsume} for \( C \leq C' \) we use the appropriate coercion.

\[
\llbracket \Gamma \vdash c : C' \rrbracket_{\eta} = \llbracket C \leq C' \rrbracket(\llbracket \Gamma \vdash c : C \rrbracket_{\eta})
\]

6.3 Skeletal semantics

In order to prove coherence of denotational semantics, we draw from the semantics of Eff [6], which solves the problem by utilising skeleton semantics. The core idea of skeletons is to remove the need for subtyping at the cost of less-precise types. In our case this means abandoning effect information, which is the only reason for the inclusion of subtyping. Because our terms are sufficiently annotated, the removal of effect annotation allows for unique typing derivations on the skeleton terms, which nullifies any concerns about incoherent denotational semantics.

The resulting semantics is coherent by construction, but feels unsatisfactory when dealing with effects, raising further concerns on how to adapt equations to the skeleton setting. We therefore only use skeleton semantics as a tool to show coherence of the denotational semantics from Section 6.2.

6.3.1 Skeletons

If we remove effect information from computation types, we no longer require two separate sorts of types.

\[
\text{skelentype } S ::= \text{unit} | \text{empty} | \text{int} | S_1 \to S_2 | S_1 \Rightarrow S_2 | S_1 \times S_2 | S_1 + S_2 | \text{S list}
\]

Regular types are turned to skeleton types (skeletonized) by traversing the type and removing all effect information. Skeletonized terms are denoted by \( A^s \) and \( C^s \). For value types, we simply propagate skeletonization in a structural manner and whenever a computation type is encountered, we use \( (A!\Sigma/E)^s = A^s \). Contexts are skeletonized by skeletonizing the types in assignments, and we skeletonize terms \( v^s \) by skeletonizing the types in all annotations.

Lemma 6.3.1. If we have \( A_1 \leq A_2 \), then the skeletal types are equal \( A_1^s = A_2^s \). Similarly, for \( C_1 \leq C_2 \) it follows that \( C_1^s = C_2^s \).

Proof (formalised). The proof consists of a simple induction on the derivation of the subtyping judgement. \( \square \)
Skeletal terms are equipped with a typing relation whose typing judgements are nearly identical to the ones described in Chapter 4. To keep note of the similarity, we name the adapted rules with the prefix Skel. Rules TypeVSubsume and TypeCSubsume are now redundant. Most rules do not interact with the signature or equations and thus need no adjustments, and the ones that do are corrected below. Typing of operation cases $\Gamma \vdash h : \Sigma \Rightarrow D$ is switched to a shape that has no signature, $\Gamma \vdash s : \Sigma \Rightarrow S$.

$$\frac{\Gamma \vdash v : S_1}{\Gamma \vdash \text{op}_{S_1 	o S_2}(v) : S} \quad \text{SkelTypeOp}$$

$$\frac{\Gamma \vdash c_1 : S_1 \quad \Gamma, x : S_1 \vdash c_2 : S_2}{\Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 : S_2} \quad \text{SkelTypeDo}$$

$$\frac{\Gamma \vdash \{ j : * \Rightarrow S \}}{\Gamma \vdash \{ \} : * \Rightarrow S} \quad \text{SkelTypeCases\{}}$$

$$\frac{\Gamma, x : S, k : S_2 \vdash c : S \quad \Gamma \vdash h : * \Rightarrow S}{\Gamma \vdash \{ \text{op}_{S_1 	o S_2}(x; k) \mapsto c \} \cup h : * \Rightarrow S} \quad \text{SkelTypeCases\cup}$$

$$\frac{\Gamma, x : S_1 \vdash c_r : S_2 \quad \Gamma \vdash h : * \Rightarrow S_2}{\Gamma \vdash \text{handler (ret } (x : S_1) \mapsto c_r; h) : S_1 \Rightarrow S_2} \quad \text{SkelTypeHandler}$$

It should be noted that the rule for handlers no longer requires respects. The resulting skeleton type system mirrors the original one, but has unique type derivations.

**Lemma 6.3.2.**

- If we have $\Gamma \vdash v : A$, then we can show $\Gamma^s \vdash v^s : A^s$.
- If we have $\Gamma \vdash c : C$, then we can show $\Gamma^s \vdash c^s : C^s$.
- If we have $\Gamma \vdash h : \Sigma \Rightarrow D$, then we can show $\Gamma^s \vdash h^s : * \Rightarrow D^s$.

*Proof (formalised).* The proof proceeds by induction on the typing derivation. In the case of a subsumption rule, we use Lemma 6.3.1, making the induction hypothesis match the required conclusion. In all other cases we use the skeleton version of the rule on the induction hypotheses (so in the case of TypeDo, we now use SkelTypeDo). □

**Lemma 6.3.3.** Typing derivations for skeleton values, computations, and operation cases are unique.

*Proof (formalised).* With the removal of subsumption rules we have precisely one typing rule applicable to a term constructor. We first show typing uniqueness, meaning that if $\Gamma \vdash v : S_1$ and $\Gamma \vdash v : S_2$, then $S_1 = S_2$ (this also holds for computations and operation cases). The proof is straightforward induction on one of the typing derivations. Since there is only one rule applicable for every term constructor, we know that the other typing derivation ends with the same rule. This allows us to use inversion to obtain sub-derivations, which are suitable for induction.

By combining uniqueness of applicable typing rules with uniqueness of the assigned type, we obtain uniqueness of typing derivation. □

### 6.3.2 Semantics of skeleton types

Skeletal types $S$ have two interpretations, depending on whether the skeletal type represents a skeletonized value type or a skeletonized computation type. We denote the two
interpretations as $\langle S \rangle_v$ and $\langle S \rangle_c$ respectively. Skeletal value type interpretations are defined as follows:

$$
\begin{align*}
\langle \text{unit} \rangle_v &= \{\ast\} \\
\langle \text{int} \rangle_v &= \mathbb{N} \\
\langle \text{empty} \rangle_v &= \emptyset \\
\langle S_1 \times S_2 \rangle_v &= \langle S_1 \rangle_v \times \langle S_2 \rangle_v \\
\langle S_1 + S_2 \rangle_v &= \langle S_1 \rangle_v + \langle S_2 \rangle_v \\
\langle S \text{ list} \rangle_v &= \langle S \rangle_v^v \\
\langle S_1 \Rightarrow S_2 \rangle_v &= \langle S_1 \rangle_c \rightarrow \langle S_2 \rangle_c
\end{align*}
$$

Skeletal computation types carry no effect information, so any operation calls are theoretically possible. To that end, we construct $\Omega$, the signature of all effects. In $EEFF$, operations with the same name may have different types in different parts of the program, and we account for that by considering operation types as part of the operation name in skeleton semantics. Operations $\text{op} : S_1 \rightarrow S_2$ and $\text{op} : S'_1 \rightarrow S'_2$ are only considered equal if the types match, despite both operations sharing the same name. Operation calls in the language are type annotated, which makes this a rather minor setback.

$$
\Omega := \{ \text{op}_{S_1 \rightarrow S_2} : S_1 \rightarrow S_2 \}_{op_{S_1 \rightarrow S_2}}
$$

As with non-skeleton types, the skeleton computation type interpretation uses the computation domain.

$$
\langle \Omega \rangle X = T_\Omega(X, (\llbracket S_1 \rrbracket)_v \text{op}_{S_1 \rightarrow S_2} \in \Omega, (\llbracket S_2 \rrbracket)_c \text{op}_{S_1 \rightarrow S_2} \in \Omega)
$$

6.3.3 Semantics of skeleton values and computations

When constructing the semantics of skeletons, we need to make sure that we use the correct version of the type semantics.

$$
\begin{align*}
\langle \Gamma \vdash s : V \rangle : \langle \Gamma \rangle \rightarrow \langle S \rangle_v \\
\langle \Gamma \vdash c : S \rangle : \langle \Gamma \rangle \rightarrow \langle S \rangle_c \\
\langle \Gamma \vdash h : * \Rightarrow S \rangle : \langle \Gamma \rangle \rightarrow \text{interp}_\Omega(\langle S \rangle_c)
\end{align*}
$$

The denotational semantics of skeleton terms is nearly identical to the one in Section 6.2. We therefore abbreviate more aggressively and only discuss important changes.

$$
\begin{align*}
\langle \cdot \rangle &= \{\ast\} \\
\langle \Gamma, x : S \rangle &= \langle \Gamma \rangle \times \langle S \rangle_v \\
\langle \Gamma \vdash (\cdot) : \text{unit} \rangle &= \ast \\
\langle \Gamma \vdash (x_k : S_k)_k \vdash x_i : S_i \rangle &= \eta_i \\
\langle \Gamma \vdash n : \text{int} \rangle &= n \\
\langle \Gamma \vdash (v_1, v_2) : S_1 \times S_2 \rangle &= (\langle v_1 \rangle_\eta, \langle v_2 \rangle_\eta) \\
\langle \Gamma \vdash \text{Left}_{S_1 + S_2} v : S_1 + S_2 \rangle &= \text{u}_1(\langle v \rangle_\eta) \\
\langle \Gamma \vdash \text{Right}_{S_1 + S_2} v : S_1 + S_2 \rangle &= \text{u}_2(\langle v \rangle_\eta) \\
\langle \Gamma \vdash \text{fun} (x : S_1) \mapsto c : S_1 \rightarrow S_2 \rangle &= \lambda a . \langle c \rangle_\eta(\eta, a) \\
\langle \Gamma \vdash \text{handler} (\text{ret} (x : S_1) \mapsto c_r : h) : S_1 \Rightarrow S_2 \rangle &= \text{lft}_{\llbracket \eta \rrbracket}(\langle \lambda a . \langle c_r \rangle_\eta(\eta, a) \rangle)
\end{align*}
$$

Operation cases only cover a few operations, but their denotation must provide a function for every operation in $\Omega$. If $h$ does not contain a case for operation $\text{op} : S_1 \rightarrow S_2$ then we set the component to a function that always returns $\bot$.

$$
\langle \Gamma \vdash h : * \Rightarrow S \rangle \text{op}_{S_1 \rightarrow S_2} = \begin{cases} 
\lambda a . \lambda k . \llbracket c \rrbracket_\eta(\eta, a, k) & \text{if } (\text{op}_{S_1 \rightarrow S_2}(x : k) \mapsto c_\text{op}) \in h \\
\lambda a . \lambda k . \bot & \text{otherwise}
\end{cases}
$$

78
For the denotation of operations we make sure to tag \( \text{in}_{op} \) with the appropriate type.

\[
\langle \Gamma \vdash_s \text{ret } v : S \rangle \eta = \text{in}_{\text{val}}(\langle v \rangle \eta)
\]

\[
\langle \Gamma \vdash_s \text{op}_{S_1 \to S_2}(v; y.c) : S \rangle \eta = \text{in}_{\text{op}_{S_1 \to S_2}}(\langle v \rangle \eta; \lambda b . \langle c \rangle(\langle b \rangle, \eta))
\]

Denotations of other computation remain largely unchanged.

\[
\langle \Gamma \vdash_s \text{absurd}_S v : S \rangle \eta = \text{emptyfun}_{\langle S \rangle}, (\langle v \rangle \eta)
\]

\[
\langle \Gamma \vdash_s \text{match } v \text{ with } (x, y) \mapsto c : S \rangle \eta =
\langle c \rangle(\eta, a, b) \quad \text{for } \langle v \rangle \eta = (a, b)
\]

\[
\langle \Gamma \vdash_s \text{match } v \text{ with } \text{Left } x \mapsto c_1 \mid \text{Right } y \mapsto c_2 : S \rangle \eta =
\begin{cases}
\langle c_1 \rangle(\eta, a) ; \langle v \rangle \eta = \iota_1 a, \\
\langle c_2 \rangle(\eta, b) ; \langle v \rangle \eta = \iota_2 b,
\end{cases}
\]

\[
\langle \Gamma \vdash_s \text{match } v \text{ with } [ ] \mapsto c_1 \mid x :: x \mapsto c_2 : S \rangle \eta =
\begin{cases}
\langle c_1 \rangle \eta ; \langle v \rangle \eta = e, \\
\langle c_2 \rangle(\eta, a_0, (a_i)_{i=1}^n) ; \langle v \rangle \eta = a_0, \ldots, a_n
\end{cases}
\]

\[
\langle \Gamma \vdash_s v_1 \cdot v_2 : S \rangle \eta = (\langle v_1 \rangle \eta)(\langle v_2 \rangle \eta)
\]

\[
\langle \Gamma \vdash_s \text{with } v \text{ handle } c : S \rangle \eta = (\langle v \rangle \eta)(\langle c \rangle \eta)
\]

\[
\langle \Gamma \vdash_s \text{do } x \leftarrow c_1 \text{ in } c_2 : S \rangle \eta =
\text{lift}_{\langle S \rangle, a}((\lambda a . \langle \Gamma, x : S' \vdash_s c_2 : S(\langle \eta, a \rangle)))(\langle c_1 \rangle \eta)
\]

\[
\langle \Gamma \vdash_s \text{let rec } f x : S_1 \to S_2 = c_1 \text{ in } c_2 : S \rangle \eta =
\langle c_2 \rangle(\eta, \mu f . \lambda a . \langle c_1 \rangle(\langle \eta, a, f \rangle))
\]

### 6.4 Coherence of denotational semantics

The denotational semantics from Section 6.2 closely follows the type system information, but it is not entirely clear whether it is coherent. On the other hand, skeleton semantics from Section 6.3 ignores effect information for the benefit of coherence. However, looking at the construction of both semantics, it is obvious that terms are assigned nearly identical mathematical objects. Following Reynolds [40], we link the two semantics through a logical relation in order to use coherence of skeletal semantics to show coherence of the regular semantics.

At first glance, it might seem that the problem is simpler; after all, we just switched every \( \Sigma \) with \( \Omega \), and we know that subtyping allows increasing the effect signature. The problem lies in the fact that we increase \textit{all} signatures, even the ones in contravariant positions. We are therefore forced to adapt a more advanced approach than just simple embeddings.

#### 6.4.1 Relation definition

We define the \( x \)-ray relation that connects the semantics of terms to their skeleton counterparts.

\[
(\sim_A) \subseteq \llbracket A \rrbracket \times (\llbracket A^\circ \rrbracket)_v 
\]

\[
(\sim_C) \subseteq \llbracket C \rrbracket \times (\llbracket C^\circ \rrbracket)_c
\]

\[
(\sim_{\Sigma^P}) \subseteq (\text{interp}_C(\llbracket D \rrbracket)) \times (\text{interp}_C(\llbracket D^\circ \rrbracket)_c)
\]
We only consider these relations for well-formed types. The definition for \( \prec \) on value types is as follows:

- \( \prec_{\text{empty}} \) is the empty relation,
- \( \star \prec_{\text{unit}} \star \),
- \( n \prec_{\text{int}} m \) if and only if \( n = m \),
- \( (a, b) \prec_{A \times B} (a', b') \) if and only if \( a \prec_A a' \land b \prec_B b' \),
- \( x \prec_{A + B} x' \) if and only if
  - \( x = t_1(a), x' = t_1(a') \), and \( a \prec_A a' \) or
  - \( x = t_2(b), x' = t_2(b') \), and \( b \prec_B b' \).
- \( (a_i)_{i=0}^n \prec_{A_1 \ldots \text{int}} (a'_i)_{i=0}^m \) if and only if \( n = m \) and \( a_i \prec_A a'_i \) for all \( i = 0, \ldots, n \),
- \( f \prec_{A \rightarrow C} f' \) if and only if \( \forall a, a'. \ (a \prec_A a' \Rightarrow f(a) \prec_C f'(a')) \),
- \( g \prec_{C = D} g' \) if and only if \( \forall c, c'. \ (c \prec_C c' \Rightarrow g(c) \prec_D g'(c')) \).

For functions and handlers, we require that the skeleton counterparts behave correctly only on related arguments. For \( \prec_C \) we start by defining a relation that "checks \( n \) levels deep", denoted \( \prec^n_C \):

- \( \forall c, c'. \ c \prec^n_C c' \)
- \( c \prec^n_{A!\Sigma/E} c' \) if and only if
  - \( c = \bot \) and \( c' = \bot \), or
  - \( c = \text{in}_{n!1}(a), c' = \text{in}_{n!1}(a'), \) and \( a \prec_A a' \), or
  - \( c = \text{in}_{op}(a, \kappa) \) and \( c' = \text{in}_{op}(a', \kappa') \) where \( a \prec_{op} a' \) and for all \( b \prec_{op} b' \), it follows that \( \kappa(b) \prec_{A!\Sigma/E}^{(n-1)} \kappa'(b') \), under the condition that \( op : A_{op} \rightarrow B_{op} \in \Sigma \) and \( A_{op} s = S_1, B_{op} s = S_2 \).

The above definition introduces no circularities, because \( A_{op} \) and \( B_{op} \) are part of \( A!\Sigma/E \) and \( n \) in \( \prec^n \) decreases. The fact that \( (\prec^n) \) is a subset of \( (\prec^{n-1}) \) can be checked by straightforward induction on \( n \). Finally, we define

\[
\text{c} \prec C \text{ c}' \iff (c, c') \in \bigcap_n (\prec^n_C).
\]

We could define \( \prec_C \) as a least relation closed under certain rules, but having an explicit construction greatly benefits us in later proofs, as it admits a simple induction principle on \( n \).

For operation cases we say that \( H \prec_{\Sigma = D} H', \) if and only if for all \( op : A \rightarrow B \in \Sigma, a \prec_A a', \) and \( \kappa \prec_{B \rightarrow D} \kappa' \), we have

\[
H_{op}(a, \kappa) \prec D H'_{op}(a', \kappa').
\]

The notion of \( \prec \) naturally extends to contexts, where \( \eta \prec_{\Gamma} \eta' \) means that the components of \( \eta \) and \( \eta' \) are related by appropriate x-ray relations.

We later need to use the x-ray relation for recursion and induction, so we ensure that \( \prec \) is chain complete. If two chains are related component-wise, we need the suprema to be related as well.
Lemma 6.4.1.

- Suppose \((a_i)_i\) is a chain in \([A]\) and \((a'_i)_i\) a chain in \([A']\). If \(a_i \succ_A a'_i\) for all \(i\), then it holds that \(\bigvee_i a_i \succ \bigvee_i a'_i\).
- Suppose \((c_i)_i\) is a chain in \([C]\) and \((c'_i)_i\) a chain in \([C']\). If \(c_i \succ_C c'_i\) for all \(i\), then it holds that \(\bigvee_i c_i \succ \bigvee_i c'_i\).
- Suppose \((H_i)_i\) is a chain in \(\text{interp}_c([D])\) and \((H'_i)_i\) a chain in \(\text{interp}_c([D'])\). If \(H_i \succ_{=D} H'_i\) for all \(i\), then it holds that \(\bigvee_i H_i \succ \bigvee_i H'_i\).

Proof. We proceed by induction on derivation of type well-formedness.

- Chains in predomins for \text{unit} and \text{int} are repetitions of the same element, so the proof is trivial. There are no chains for \text{empty}.

- We prove it for \(A \times B\), with proofs for \(A + B\) and \(A \text{ list}\) being similar.

  Assume we have two chains \(((a_i, b_i))_i\) and \(((a'_i, b'_i))_i\) for which \((a_i, b_i) \succ_{A \times B} (a'_i, b'_i)\).

  It follows that \((a_i)_i\) and \((a'_i)_i\) are chains in \([A]\) and \([A']\) respectively. By definition of \(\succ_{A \times B}\) it holds \(\forall i. a_i \succ_A a'_i\). Therefore, we can use the induction hypothesis for \(A\) and \(B\) to obtain \(\bigvee_i a_i \succ_A \bigvee_i a'_i\) and \(\bigvee_i b_i \succ_B \bigvee_i b'_i\), respectively. From \(\bigvee_i (a_i, b_i) = (\bigvee_i a_i, \bigvee_i b_i)\), it follows that \(\bigvee_i (a_i, b_i) \succ \bigvee_i (a'_i, b'_i)\).

- We show it for \(A \to C\), and the proof for \(C \Rightarrow D\) is similar.

  Assume chains \((f_i)_i\) and \((f'_i)_i\) for which \(f_i \succ_{A \Rightarrow C} f'_i\). To show \(\bigvee_i f_i \succ \bigvee_i f'_i\), we show that for all \(a \succ_A a'\), it holds that \((\bigvee_i f_i)(a) \succ (\bigvee_i f'_i)(a')\). Suprema of functions are defined pointwise, so it is equivalent to show \(\bigvee_i f_i(a) \succ \bigvee_i f'_i(a')\).

  Assume we have \(a \succ a'\). If \((f_i)_i\) is a chain in \([A \to C]\), then \((f_i(a))_i\) is a chain in \([C]\) by definition of \(\leq_{A \to C}\). Similarly, \((f'_i(a'))_i\) is a chain in \([C']\). From \(f_i \succ f'_i\) it follows that \(f_i(a) \succ f'_i(a')\), since \(a \succ a'\). Therefore, for chains \((f_i(a))_i\) and \((f'_i(a'))_i\), it holds that \(\forall i. f_i(a) \succ f'_i(a')\). With that, we use the induction hypothesis for \(C\) to conclude \(\bigvee_i f_i(a) \succ \bigvee_i f'_i(a')\).

- We show \(\bigvee_i c_i \succ_{A \Sigma / \emptyset} \bigvee_i c'_i\) by showing \(\forall n. \bigvee_i c_i \succ_n \bigvee_i c'_i\), which we do by induction on \(n\). The case for \(\succ^0\) is trivial. For the induction step, we proceed by case analysis on the shape of chain \((c_i)_i\). By assumption of \(\forall i. c_i \succ c'_i\), it follows that the shapes of \(c'_i\) and \(c_i\) match, and we now consider all three possibilities for chains (presented in Section 6.1).

  - Assume both chains are constantly \(\bot\); then their suprema are also \(\bot\), and we know that \(\bot \succ^a \bot\) by definition.
  - Assume at some point the chains transition to elements \(\text{in}_{\text{val}}(a_i)\) and \(\text{in}_{\text{val}}(a'_i)\).

     From \(\forall i. \text{in}_{\text{val}}(a_i) \succ \text{in}_{\text{val}}(a'_i)\) we know that \(\forall i. a_i \succ_a a'_i\). With that, we use the IH for \(A\) on chains \((a_i)_i\) and \((a'_i)_i\) to arrive at \(\bigvee_i a_i \succ \bigvee_i a'_i\). The suprema distribute over the \(\text{in}_{\text{val}}\) constructor, so it holds that \(\bigvee_i \text{in}_{\text{val}}(a_i) \succ^a \bigvee_i \text{in}_{\text{val}}(a'_i)\).
  - We now assume that at some point the chains transition to elements \(\text{in}_{\text{op}}(a_i, \kappa_i)\) and \(\text{in}_{\text{op}}(a'_i, \kappa'_i)\). The names and types of operations match because \(\forall i. \text{in}_{\text{op}}(a_i, \kappa_i) \succ \text{in}_{\text{op}}(a'_i, \kappa'_i)\). This further implies that \(\forall i. a_i \succ a'_i\) and that for any \(b \succ b'\), we have \(\forall i. \kappa_i(b) \succ \kappa'_i(b')\). To show \(\bigvee_i \text{in}_{\text{op}}(a_i, \kappa_i) \succ^a \bigvee_i \text{in}_{\text{op}}(a'_i, \kappa'_i)\), we need \(\bigvee_i a_i \succ \bigvee_i a'_i\) and that any \(b \succ b'\) implies \((\bigvee_i \kappa_i)(b) \succ (\bigvee_i \kappa'_i)(b')\). We know that \((a_i)_i\) and \((a'_i)_i\) are chains, and \(\forall i. a_i \succ a'_i\) so by IH of \(A_{\text{op}}\) we have \(\bigvee_i a_i \succ \bigvee_i a'_i\).
Next we assume \( b \succ b' \). The hypothesis from induction on \( n \) states that for any chains \((c_i)\) and \((c'_i)\) for which \( \forall i. c_i \succ c'_i \), we have \( \bigvee_i c_i \sim^{n-1} \bigvee_i c'_i \). We know that \( \kappa_i(b) \) and \( \kappa'_i(b') \) are chains and that \( \forall i. \kappa_i(b) \sim \kappa'_i(b') \), so by induction \( \bigvee_i \kappa_i(b) \sim^{n-1} \bigvee_i \kappa'_i(b') \). This is equivalent to \( \bigvee_i \kappa_i(b) \sim^{n-1} (\bigvee_i \kappa'_i(b')) \) because function suprema are pointwise.

- For interpretations we define ordering and the \( x \)-relation by components so it suffices to show chain-completeness for the component of an arbitrary \( \text{op} : A \rightarrow B \in \Sigma \). We denote the \( \text{op} \) component of \( H \) as \( H^{\text{op}} \), as opposed to the usual \( H_{\text{op}} \), to allow space for chain indices. We also drop the type in the skeleton-operation name, writing \( H'^{\text{op}} \) instead of \( H'^{\text{op},\lambda \rightarrow A^\ast} \). If \( (H_i)_i \) is a chain in \( \text{interp}_\Sigma([D]) \), then \( H'^{\text{op}}_i \) is a chain in \( [A] \times ([B] \rightarrow [D]) \rightarrow [D] \) and also \( \bigvee_i H'^{\text{op}}_i = (\bigvee_i H_i)^{\text{op}} \). Similarly, \( (H'^{\text{op}}_i)_i \) is a chain in \( ([A]_\ast)^{\ast} \times (([B]_\ast)_\ast \rightarrow ([D]_\ast)_\ast) \rightarrow ([D]_\ast)_\ast \). For any \( i \), the assumption \( H_i \sim H'_i \) gives us an assumption for the \( \text{op} \) component.

\[
(a \sim_A a' \wedge \kappa \sim_B \kappa') \Rightarrow H'^{\text{op}}_i(a,\kappa) \sim_D H'^{\text{op}}_i(a',\kappa')
\]

If we fix \( a \sim a' \) and \( \kappa \sim \kappa' \), we obtain two chains, where \( \forall i. (H'^{\text{op}}_i(a,\kappa) \sim H'^{\text{op}}_i(a',\kappa')) \).

By IH for \( D \) this also holds for the suprema.

\[
(a \sim a' \wedge \kappa \sim \kappa') \Rightarrow (\bigvee_i H'^{\text{op}}_i)(a,\kappa) \sim_D (\bigvee_i H'^{\text{op}}_i)(a',\kappa')
\]

This holds for any component, so \( \bigvee_i H_i \sim \bigvee_i H'_i \).

\( \square \)

### 6.4.2 Relating semantics

We now proceed to show that the denotation of an \( EEFF \) term is related to the denotation of its skeleton. Regardless of the typing derivation, denotations of \( [\Gamma \vdash v : A] \) will be related to a unique \( (\Gamma_\ast, v_\ast : A_\ast) \). In Subsection 6.4.3 we show that denotations related to the same skeleton have to be equal.

**Lemma 6.4.2.** Assume \( f \sim_A \xrightarrow{D} f' \) and \( H \sim_{\Sigma} \xrightarrow{D} H' \). Then for any \( c \sim_{\Sigma/E} c' \) (where \( E \) is arbitrary) it holds that

\[
\text{lift}_H f(c) \sim_D \text{lift}_H f'(c').
\]

**Proof.** Admissibility of the predicate used in the lemma is unclear, so we introduce an explicit construction of \( \text{lift} \) to obtain a different induction principle.

\[
\begin{align*}
(\text{lift}_H^n f)(x) &= \bot, \\
(\text{lift}_H^n f)(\bot) &= \bot, \\
(\text{lift}_H^n f)(\text{inval}(x)) &= f(x), \\
(\text{lift}_H^n f)(\text{inop}(x,\kappa)) &= H_{\text{op}}(x,\text{lift}_H^{(n-1)} f \circ \kappa).
\end{align*}
\]

Asserting that for any \( n \) we have

\[
\text{lift}_H^n f \le \text{lift}_H^{(n+1)} f,
\]

is done by straightforward induction on \( n \). This allows us to construct \( \bigvee_i \text{lift}_H^n f \), which satisfies the same conditions as the lift introduced in Subsection 6.2.1. By uniqueness of the recursion principle it follows that

\[
\bigvee_i \text{lift}_H^n f = \text{lift}_H f.
\]

We wish to show that for every \( n \) and every \( c \sim_{\Sigma/E} c' \) we have

\[
\text{lift}_H^n f(c) \sim_D \text{lift}_H^n f'(c').
\]

We proceed by induction on \( n \). For \( n = 0 \) it follows trivially. For the induction step, we assume it holds for \( n - 1 \) and try to show it for \( n \). We proceed with a case analysis on \( c \).

---

82
• From $c = \bot$ it follows that $c' = \bot$. Because lift is strict for any $n$, we end up with $\bot$ on both sides, and $\bot \Rightarrow \bot$.

• Assume $c = \text{in}_{\text{val}}(a)$, which implies $c' = \text{in}_{\text{val}}(a')$ and $a \approx_{A} a'$. Evaluating the lift on both sides results in $f(a)$ and $f'(a')$ respectively. Since $f \approx f'$ and $a \approx a'$, we have $f(a) \approx f'(a')$.

• Let $c = \text{in}_{\text{op}}(a, \kappa)$ for some $\text{op} : A_{\text{op}} \to B_{\text{op}}$. It follows that $c' = \text{in}_{\text{op}}(a', \kappa')$ where $a \approx_{A_{\text{op}}} a'$, and for all $b \approx_{B_{\text{op}}} b'$ we have $\kappa(b) \approx_{A_{\text{op}}} \kappa'(b')$. We drop the type annotations on operations for clarity.

By induction we know that for $b \approx b'$, we have $(\text{lift}_{H}^{(n-1)} f)(\kappa(b)) \approx_{D} (\text{lift}_{H}^{(n-1)} f')(\kappa'(b'))$.

Because $H \approx_{\Sigma_{ED}} \hat{H}'_{obs}$, it holds that $a \approx a'$ and $(\text{lift}_{H}^{(n-1)} f \circ \kappa) \approx_{B \to D} (\text{lift}_{H}^{(n-1)} f' \circ \kappa')$ imply

$H_{\text{op}}(a, \text{lift}_{H}^{(n-1)} f \circ \kappa) \approx_{D} H'_{\text{op}}(a', \text{lift}_{H}^{(n-1)} f' \circ \kappa')$.

This is precisely $(\text{lift}_{H}^{n} f)(\text{in}_{\text{op}}(a, \kappa)) \approx (\text{lift}_{H}^{n} f')(\text{in}_{\text{op}}(a', \kappa'))$.

Now assume $c \approx c'$. As we established at the beginning of the proof, $((\text{lift}_{H}^{n} f)(c))_{n}$ and $((\text{lift}_{H}^{n} f')(c'))_{n}$ are chains. For any $n$, we have $(\text{lift}_{H}^{n} f)(c) \approx (\text{lift}_{H}^{n} f')(c')$, and by Lemma 6.4.1 it follows that

$$\text{lift}_{H} f(c) = \bigvee_{n}((\text{lift}_{H}^{n} f)(c)) \approx_{D} \bigvee_{n}((\text{lift}_{H}^{n} f')(c')) = \text{lift}_{H} f'(c').$$


Lemma 6.4.3.

- If $a \approx_{A} a'$ and $A \leq A'$, then $[A \leq A']a \approx_{A'} a'$.
- If $c \approx_{C} c'$ and $C \leq C'$, then $[C \leq C']c \approx_{C'} c'$.

Proof. The proof proceeds by induction on derivation of subtyping.

• The proofs for unit, int, and empty are trivial because $[A \leq A']$ is the identity function.

• We do the proof for $A \text{ list} \leq A'$ list, with product and sum types having a similar proof. From $(a_{i})_{i=0}^{n} \approx_{A} (a'_{i})_{i=0}^{n}$ it follows that the sequences are of equal length and that for $i = 0 \ldots n$, the components are related: $a_{i} \approx_{A} a'_{i}$. By induction we have $\forall i = 0 \ldots n. [A \leq A']a_{i} \approx_{A'} a'_{i}$, which suffices for $\approx_{A \text{ list}}$.

• The proof for $A \to C \leq A' \to C'$ and the proof for handler types are similar, so we only write down the former. We need to show $[A \to C \leq A' \to C']f \approx_{A \to C} f'$ holds whenever $f \approx_{A \to C} f'$. Assume $a \approx_{A} a'$. By induction $[A' \leq A]a \approx_{A} a'$, and by definition $f([A' \leq A]a) \approx_{C} f'(a')$. We use the induction hypothesis for $[C \leq C']$ to obtain

$$[C \leq C'](f([A' \leq A]a)) \approx_{C'} f'(a').$$

Because this holds for arbitrary $a \approx a'$, it follows that

$$[A \to C \leq A' \to C']f = [C \leq C'] \circ f \circ [A' \leq A] \approx_{A \to C} f'.$$
• For $A!\Sigma/E \leq A'!\Sigma'/E'$ (shortened to $C \leq C'$), we need to prove $\llbracket C \leq C' \rrbracket c \triangleright_{C'} c'$, which by definition holds if $\forall n. \llbracket C \leq C' \rrbracket c \triangleright_{C'}^n c'$. Recall the semantics of subtyping judgements for computations.

$$(\llbracket \Sigma \leq \Sigma' \rrbracket X)_{op}(x, \kappa) = \text{in}_{op}(\llbracket A_{op} \leq A'_{op} \rrbracket x, \kappa \circ \llbracket B_{op} \leq B_{op} \rrbracket)$$

$$\llbracket A!\Sigma/E \leq A'!\Sigma'/E' \rrbracket = \text{lift}_{\Sigma \leq \Sigma'}(\lambda a \cdot \text{in}_{\text{val}}(\llbracket A \leq A' \rrbracket a))$$

We prove $\forall n. \llbracket C \leq C' \rrbracket c \triangleright_{C'}^n c'$ by a second induction on $n$. The case for $n = 0$ is trivial. We now assume it holds for $(n - 1)$ and try to prove it for $n$. We do a case analysis for $c$ and infer the shape of $c'$ by definition of $c \triangleright_{C'} c'$.

○ $c = \perp$, which implies $c' = \perp$. Since lift is strict, we need to prove $\perp \triangleright_{C'}^n \perp$, which holds for any $n$.

○ Assume $c = \text{in}_{\text{val}}(a)$, and by extension $c' = \text{in}_{\text{val}}(a')$ with $a \vdash_A a'$. By induction we have $\llbracket A \leq A' \rrbracket a \triangleright_{A'} a'$.

$$\llbracket C \leq C' \rrbracket \text{in}_{\text{val}}(a) = \text{in}_{\text{val}}(\llbracket A \leq A' \rrbracket a) \triangleright_{C'}^n \text{in}_{\text{val}}(a')$$

○ If $c = \text{in}_{\text{op}}(a, \kappa)$ for $op : A_{op} \rightarrow B_{op} \in \Sigma$, then $c' = \text{in}_{A_{op} \rightarrow B_{op}}(a', \kappa')$. We further know that $a \vdash_{A_{op}} a'$, and for all $b \vdash_{B_{op}} b'$ we have $\kappa(b) \vdash_{A!\Sigma/E} \kappa'(b')$. By definition of $\Sigma \leq \Sigma'$, we know that $op : A'_{op} \rightarrow B'_{op} \in \Sigma'$ for some $A_{op} \leq A'_{op}$ and $B_{op} \leq B_{op}$.

$$\llbracket C \leq C' \rrbracket (\text{in}_{\text{op}}(a, \kappa)) = (\llbracket \Sigma \leq \Sigma' \rrbracket (\llbracket A' \rrbracket a), \llbracket C \leq C' \rrbracket \circ \kappa \circ \llbracket B_{op} \leq B_{op} \rrbracket)$$

Through induction on subtype derivation, we get $\llbracket A_{op} \leq A'_{op} \rrbracket a \vdash_{A_{op}} a'$, and similarly for any $b \vdash_{B_{op}} b'$ we have $\llbracket B_{op} \leq B_{op} \rrbracket b \vdash_{B_{op}} b'$, and it follows that $\kappa(\llbracket B_{op} \leq B_{op} \rrbracket b) \vdash_{C'} \kappa'(b')$. We now use the IH for $n - 1$ to arrive at

$$b \vdash_{B_{op}} b' \implies \llbracket C \leq C' \rrbracket (\kappa(\llbracket B_{op} \leq B_{op} \rrbracket b)) \vdash_{C'}^{(n - 1)} \kappa'(b').$$

By definition of $\triangleright_{C'}^n$, it follows that

$$\text{in}_{\text{op}}(\llbracket A_{op} \leq A'_{op} \rrbracket a, \llbracket C \leq C' \rrbracket \circ \kappa \circ \llbracket B_{op} \leq B_{op} \rrbracket) \vdash_{C'}^n \text{in}_{A_{op} \rightarrow B_{op}}(a', \kappa').$$

Since skeletons ignore subtyping, the type annotations of $\text{in}_{A_{op} \rightarrow B_{op}}(a', \kappa')$ are the same as those of $c' = \text{in}_{A_{op} \rightarrow B_{op}}(a', \kappa')$.

\[\square\]

**Lemma 6.4.4.** For any $\eta \vdash_{\Gamma} \eta'$ it holds that

○ $\llbracket \Gamma \vdash v : A \rrbracket \eta \vdash (\llbracket \Gamma \vdash t_s : v^s : A^s \rrbracket) \eta'$

○ $\llbracket \Gamma \vdash c : C \rrbracket \eta \vdash (\llbracket \Gamma \vdash t_s : c^s : C^s \rrbracket) \eta'$

○ $\llbracket \Gamma \vdash h : \Sigma \Rightarrow D \rrbracket \eta \vdash (\llbracket \Gamma \vdash t_s : h^s : * \Rightarrow D^s \rrbracket) \eta'$

**Proof.** The proof proceeds by induction on typing derivations. In all cases apart from subsumption rules, the last typing rules used in the regular and skeleton type derivations need to match. If the last rule for typing $v$ is TYPEFUN then the last rule for typing $v^s$ has to be SKELTYPEFUN. When clear, we abbreviate $\llbracket \Gamma \vdash v : A \rrbracket$ to $\llbracket v \rrbracket$ and $(\llbracket \Gamma \vdash t_s : v^s : A^s \rrbracket)$ to $\llbracket v^s \rrbracket$ for shorter proofs. All the proofs follow the same steps, so some of the easier cases have been omitted for brevity.

84
• TYPEVAR: From $\eta \Rightarrow \eta'$ it follows that for all suitable $i$, we have $\eta_i \Rightarrow \eta_i'$.

$$\left\{ \begin{array}{l}
\Gamma \vdash x_i : A_i \Rightarrow \eta_i \Rightarrow \eta_i' = (\Gamma^\varphi \triangleright x_i : A_i)\eta'
\end{array} \right.$$ 

• TYPEINT:

$$\langle n \rangle \eta = n \Rightarrow n' = (\eta)\eta'$$

• TYPEPAIR: By induction we have $\langle v_1 \rangle \eta \Rightarrow \eta'$ and $\langle v_2 \rangle \eta \Rightarrow \eta'$. Then:

$$\langle (v_1, v_2) \rangle \eta = (\langle v_1 \rangle \eta, \langle v_2 \rangle \eta) \Rightarrow A \times B \Rightarrow (\langle v_1 \rangle \eta', \langle v_2 \rangle \eta') = ((v_1^s, v_2^s), \eta')$$

• TYPELEFT: By induction we have $\langle v \rangle \eta \Rightarrow \eta'$.

$$\langle \text{Left}_{A + B} \rangle \eta = \text{left}(\langle v \rangle \eta) \Rightarrow \eta'$$

• TYPECONS: By induction we have $\langle v_1 \rangle \eta \Rightarrow \eta'$ and $\langle v_2 \rangle \eta \Rightarrow \eta'$. If we add $\langle v_1 \rangle \eta$ and $\langle v_1 \rangle \eta'$ to the beginning of $\langle v_2 \rangle \eta$ and $\langle v_2 \rangle \eta'$, respectively, the resulting sequences are still of equal length with related components.

$$\langle v_1 :: v_2 \rangle \eta = \langle v_1 \rangle \eta :: \langle v_2 \rangle \eta \Rightarrow \Gamma \times A \Rightarrow (\langle v_1 \rangle \eta', v_2^s) \Rightarrow (\langle v_1 \rangle :: v_2 \rangle \eta')$$

• TYPEFUN: For any $\eta \Rightarrow \eta'$, we have $\langle \eta, a \rangle \Rightarrow \Gamma : A$, $\eta' \Rightarrow \eta$, and by induction we obtain $\langle c \rangle (\eta, a) \Rightarrow (\langle c \rangle \eta)\eta' \Rightarrow (\langle c \rangle \eta)\eta'$. This satisfies the conditions for $\Rightarrow A \rightarrow C$.

$$\text{fun}(x : A) \mapsto c \eta = \lambda a \cdot (\langle c \rangle \eta)\eta'$$

• TYPEHANDLER: For a $\Rightarrow A$, we have $\langle \eta, a \rangle \Rightarrow \Gamma : A$, $\eta' \Rightarrow \eta$, so by induction it holds that $\langle c \rangle (\eta, a) \Rightarrow (\langle c \rangle \eta)\eta'$. Another IH states that $\langle h \rangle \Rightarrow \Rightarrow A \Rightarrow (\langle c \rangle \eta)\eta'$.

• TYPEVSubsume: The premises of the rule are $A \preceq A'$ and $A' \preceq A$. By induction we obtain $\Gamma \vdash v : A \Rightarrow A$, $\preceq (\Gamma^\varphi \triangleright v \Rightarrow : A\alpha)$ and use Lemma 6.4.3 to transform it into $\Gamma \vdash v : A \Rightarrow (\Gamma^\varphi \triangleright v \Rightarrow : A\alpha)\eta'$. According to Lemma 6.3.1 subtyping has no effect on skeletal types, so $\Gamma^\varphi \triangleright v \Rightarrow : A\alpha$ is the same as $\Gamma^\varphi \triangleright v \Rightarrow : A\alpha$.

$$\left\{ \begin{array}{l}
\Gamma \vdash v : A \Rightarrow (\Gamma^\varphi \triangleright v \Rightarrow : A\alpha)\eta' = (\Gamma^\varphi \triangleright v \Rightarrow : A\alpha)\eta'
\end{array} \right.$$ 

• TYPEPRODMatch: We have $\langle v \rangle \Rightarrow A \times B$ by induction. It follows that for $\langle v \rangle = (a, b)$ and $\langle v' \rangle = (a', b')$, it holds that $a \Rightarrow A$ and $b \Rightarrow B$. Therefore, it also holds that $c \Rightarrow A$ and $b \Rightarrow B$ and by induction $\langle c \rangle (\eta, a) \Rightarrow (\langle c \rangle \eta)\eta'$.

$$\langle \text{match } v \text{ with } (x, y) \mapsto c \rangle \eta = \langle c \rangle (\eta, a) \Rightarrow (\langle c \rangle \eta)\eta'$$
• TYPELISTMATCH: By induction \( [[v]] \vdash_{A \text{list}} (v^\ell) \). We have two possibilities:
  
  \[ \text{match } v \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 \eta = [c_1 \eta] \]
  
  for \( \eta' \). Then
  
  \[ (\eta, a_0, (a_i)_{i=0}^n) \vdash_{T, x:A, xS:A \text{list}} (\eta', a'_0, (a'_i)_{i=1}^n). \]
  
  and by induction \( [[c_2][\eta, a_0, (a_i)_{i=1}^n] \vdash_C (c_2^\ell)(\eta', a'_0, (a'_i)_{i=1}^n). \)
  
  \[ \text{match } v \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 \eta = [c_2[\eta, a_0, (a_i)_{i=1}^n] \]
  
  for \( \eta' \).

• TYPEABSURD: By induction we have \( [[v]] \vdash_{\text{empty}} (v^\ell) \eta' \). Since \( \vdash_{\text{empty}} \) is an empty relation, we have a faulty assumption, and the statement is vacuously true.

• TYPEOP: Assume \( op : A' \rightarrow B' \in \Sigma \) and let \( A \rightarrow B \) be the type annotations of the operation call. We combine the IH \( [[v]] \vdash_{A} (v^\ell) \eta' \) with Lemma 6.4.3 to get
  
  \[ [[A \leq A']]][[v]] \eta \vdash_{A'} (v^\ell) \eta'. \]
  
  We also get that for any \( b \vdash_{B} b' \), we have
  
  \[ [[c][\eta, B' \leq B]] \vdash_C (c^\ell)(\eta', b'). \]
  
  These are the exact requirements for
  
  \[ \vdash_{\text{op}_{A \rightarrow B}(v; y.c)} \eta = \text{in}_{\text{op}}([[A \leq A'][[v]] \eta); \lambda b. [[c][\eta, B' \leq B]] b)) \]
  
  \[ \vdash_C \text{in}_{\text{op}_{A \rightarrow B}(v; y.c)}([[v]] \eta; \lambda b'. [[c][\eta', b']]) = [[\text{op}_{A \rightarrow B}(v; y.c)] \eta'. \]

• TYPEDO: For any \( a \vdash a' \), the IH states \( [[c_2][\eta, a] \vdash_{B \Sigma/E} (c_2^\ell)(\eta', a'). \) Checking that \( F_{B \Sigma/E} \vdash_{\Sigma \rightarrow A \Sigma/E} F_{c_2^\ell} \eta \) is a simple analysis of the components of the free interpretation. By Lemma 6.4.2, we obtain
  
  \[ \text{lift}_{F_{B \Sigma/E}}(\lambda a \cdot [c_2[\eta, a]])(\eta) = \text{lift}_{F_{c_2^\ell}}(\lambda a'. \langle c_2^\ell \rangle(\eta', a')). \]

  Since \( [[c_1][\eta] \vdash_{A \Sigma/E} (c_1^\ell) \) holds by induction, it follows that
  
  \[ [[x \leftarrow c_1 \text{ in } c_2]] \eta = \text{lift}_{F_{B \Sigma/E}}(\lambda a \cdot [c_2[\eta, a]])(\eta) \]
  
  \[ \vdash_{B \Sigma/E} \]
  
  \[ \text{lift}_{F_{c_2^\ell}}(\lambda a'. \langle c_2^\ell \rangle(\eta', a'))(\eta) = [[x \leftarrow c_1^\ell \text{ in } c_2^\ell]] \eta'. \]

• TYPEAPP: By induction we have \( [[v_1][\eta] \vdash_{A \rightarrow C} (v_1^\ell) \eta' \) and \( [[v_2][\eta] \vdash_{A} (v_2^\ell) \eta' \). The rest follows by definition of \( \vdash_{A \rightarrow C} \).
  
  \[ [[v_1] v_2] \eta = ([[v_1][\eta]) ([v_2][\eta]) \vdash_C (v_1^\ell)[v_2^\ell]) = (v_1^\ell v_2^\ell) \eta'. \]
• **TYPELETREC**: Assume \( f \succ_A C \) \( f' \) and \( a \succ_A a' \). By induction it follows that 
\[
\llbracket c_1 \rrbracket(\eta, a, f) \succ_C (c_1')(\eta', a', f').
\]
This means that for any \( f \succ f' \), we have
\[
\lambda a \cdot \llbracket c_1 \rrbracket(\eta, a, f) \succ_C \lambda a' . (\llbracket c_1 \rrbracket(\eta, a', f')).
\]
We use the explicit construction of the least fixed point. The bottom element of a function domain is \( \lambda x . \bot \), and it is straightforward to show \( \lambda a . \bot \succ_A C \lambda a' . \bot \).
We denote \( f_0 = \lambda a . \bot \) and \( f_{k+1} = \lambda a \cdot \llbracket c_1 \rrbracket(\eta, a, f_k) \), and similarly \( f'_0 = \lambda a' . \bot \) and \( f'_{k+1} = \lambda a' . (\llbracket c_1 \rrbracket(\eta', a', f'_k)). \)
We have \( f_0 \sim f'_0 \), and we know that if \( f_i \sim f'_i \), then by IH for \( c_1 \) it holds that \( f_{i+1} \sim f'_{i+1} \), so \( f_i \) and \( f'_i \) are related for all \( i \). By Lemma 6.4.1 it follows that \( \bigvee_i f_i \sim \bigvee_i f'_i \), which relates the fixpoints.
The other IH states that for \( f \succ_A C f' \), we have \( \llbracket c_2 \rrbracket(\eta, f) \succ_D (c_2')(\eta', f') \).
\[
\begin{align*}
\llbracket \text{let } f \ x : A \to C = c_1 \text{ in } c_2 \rrbracket &= \llbracket \text{let } f \ x : A \to C = c_1 \text{ in } c_2 \rrbracket \\
&\succ_D (c_2')(\eta', \bigvee_i f'_i)
\end{align*}
\]

• **TYPECSUBSUME**: We proceed in the same way as for **TYPEVSUBSUME**.
\[
\begin{align*}
\llbracket \Gamma \vdash c : C \rrbracket \eta &= \llbracket C' \rrbracket \leq \llbracket (\Gamma \vdash c : C') \eta \rrbracket \\
&\succ_C (\llbracket \Gamma^s \vdash s : C'' \rrbracket \eta') \succ (\llbracket \Gamma^s \vdash s \leq : C' \rrbracket \eta')
\end{align*}
\]

• **TYPECASES\( \{ \} \)**: In the definition of \( \succ \) \( \text{inter}_{\Sigma}(D) \) the requirements are bound to every \( o_p \in \Sigma \). The signature is empty, so the denotations are related by default.

• **TYPECASES\( U \)**: All components for \( \Sigma \) are already related by the induction hypothesis \( \llbracket h \rrbracket \eta \succ_{\Sigma=\Sigma} (h') \eta' \). All that is left is to check for the newly added \( o_p : A \to B \). For any \( a \succ_A a' \) and \( \kappa \succ_{B \to D} \kappa' \) it holds that \( \llbracket c_{o_p} \rrbracket(\eta, a, \kappa) \succ_D (c_{o_p}')(\eta', a', \kappa') \) (by induction). This is precisely the requirement for the \( o_p \) component of the interpretation. There are no requirements for operations that are not present in \( \Sigma \).

\[\square\]

### 6.4.3 Embedding-retraction functions and coherence

A direct map from term denotations to skeleton denotations greatly aids us in the proof of coherence. To that end, we provide pairs of functions that act as embedding-retraction pairs.

\[
\begin{align*}
F_A : \llbracket A \rrbracket \to \llbracket A^s \rrbracket_v \\
F_C : \llbracket C \rrbracket \to \llbracket C^s \rrbracket_c \\
F_{\Sigma=\Sigma} : \text{inter}_{\Sigma}(\llbracket D \rrbracket) \to \text{inter}_{\Sigma}(\llbracket D^s \rrbracket)
\end{align*}
\]

We know that not all skeleton denotations have a matching term denotation, but we manage to make functions \( G \) total by making use of \( \bot \). We are only interested in x-ray related elements, and in those cases there are no such issues.

The functions \( F_{\text{empty}}, G_{\text{empty}} \) are empty functions. Functions \( F_{\text{unit}}, G_{\text{unit}}, F_{\text{int}}, \) and \( G_{\text{int}} \) are identities, since the underlying predomains match.

\[
\begin{align*}
F_{A \times B}((a, b)) &= (F_A(a), F_B(b)) \\
F_{A + B}(x) &= \begin{cases} 
1_1(F_A(a)) & ; x = t_1(a) \\
1_2(F_B(b)) & ; x = t_2(b)
\end{cases} \\
F_{A \times \text{list}}(\{(a_i)_{i=0}^n\}) &= (F_A(a_i))_{i=0}^n \\
G_{A \times B}((a', b')) &= (G_A(a'), G_B(b')) \\
G_{A + B}(x') &= \begin{cases} 
1_1(G_A(a')) & ; x' = t_1(a') \\
1_2(G_B(b')) & ; x' = t_2(b')
\end{cases} \\
G_{A \times \text{list}}((a_i')_{i=0}^n) &= (G_A(a_i'))_{i=0}^n
\end{align*}
\]

87
When it comes to functions and handlers, we need to be careful, as the results need to be continuous functions. We show in Lemma 6.4.6 that $F$ and $G$ are continuous and strict for computation types, which is necessary for handlers.

$$
F_{A \to C}(f) = F_C \circ f \circ G_A \\
F_{C \Rightarrow D}(g) = F_D \circ g \circ G_C \\
G_{A \to C}(f') = G_C \circ f' \circ F_A \\
G_{C \Rightarrow D}(g') = G_D \circ g' \circ F_C
$$

For computation types, we could use the recursion principle of computation domains, but this turns out to be very restrictive for proofs. We therefore make an explicit least fixpoint construction. We construct $F^n_C$, which “maps $n$ levels deep”, and then construct $F$ as the supremum of such maps. For $C = A!\Sigma/E$ we define

$$
F^n_C(\bot) = \bot \\
F^n_C(\top) = \bot \\
F^n_C(\text{in}_{\text{val}}(a)) = \text{in}_{\text{val}}(F_A(a)) \\
F^n_C(\text{in}_{\text{op}}(a, \kappa)) = \text{in}_{\text{op}}(F_{A_{\text{op}}}(a), \lambda b \cdot F^{(n-1)}_C(\kappa(G_{B_{\text{op}}}(b')))) \quad (\text{op} : A_{\text{op}} \to B_{\text{op}} \in \Sigma)
$$

$$
G^n_C(\bot) = \bot \\
G^n_C(\top) = \bot \\
G^n_C(\text{in}_{\text{val}}(a')) = \text{in}_{\text{val}}(G_A(a')) \\
G^n_C(\text{in}_{\text{op}}(a', \kappa')) = \begin{cases} 
\text{in}_{\text{op}}(G_{A_{\text{op}}}(a'), \lambda b \cdot G^{(n-1)}_C(\kappa'(G_{B_{\text{op}}}(b')))) ; \text{op} : A_{\text{op}} \to B_{\text{op}} \in \Sigma \land \text{op} \neq S_1 \land \text{op} \neq S_2 \\
\bot ; \text{otherwise}
\end{cases}
$$

We need not worry about multiple options for $F^n_C(\text{in}_{\text{op}}(a, \kappa))$ and $G^n_C(\text{in}_{\text{op}}(a', \kappa'))$. The signature $\Sigma$ is well formed because we only work with well-formed types in denotational semantics. Therefore, it follows that there is at most one occurrence of the name op in $\Sigma$.

Before we can construct $F_C$ and $G_C$ through suprema, we need to first ensure that this is possible.

Lemma 6.4.5.

- For every $c \in \llbracket \downbracket$, the elements $(F^n_C(c))_n$ form a chain.
- For every $c' \in \llbracket \upbracket_c$, the elements $(G^n_C(c'))_n$ form a chain.

Proof. The proofs for $F^n_C$ and $G^n_C$ are similar, so we only do it for $F^n_C$. Assume an arbitrary $c \in \llbracket \downbracket$ where $C = A!\Sigma/E$. To show that elements $F^n_C(c)$ form a chain, we need to show $\forall n. F^n_C(c) \leq F^{(n+1)}_C(c)$. The proof proceeds by induction on $n$. For the base case of $n = 0$, we have $F^0_C(c) = \bot \leq F^1_C(c)$, since $\bot$ is the least element of the domain. For the induction step, we proceed by case analysis on $c$.

- If $c = \bot$, then $F^0_C(\bot) = \bot = F^{(n+1)}_C(\bot)$.
- If $c = \text{in}_{\text{val}}(a) \in \llbracket A \rrbracket$, then $F^n_C(\text{in}_{\text{val}}(a)) = \text{in}_{\text{val}}(F_A(a)) = F^{(n+1)}_C(\text{in}_{\text{val}}(a))$.
- Assume $c = \text{in}_{\text{op}}(a, \kappa)$ for some $\text{op} : A_{\text{op}} \to B_{\text{op}} \in \Sigma$. We omit the type annotations of the skeletal in_{opt} constructors for brevity.

$$
F^n_C(\text{in}_{\text{op}}(a, \kappa)) = \text{in}_{\text{op}}(F_{A_{\text{op}}}(a), \lambda b \cdot F^{(n-1)}_C(\kappa(G_{B_{\text{op}}}(b'))))
$$

88
By induction we know that $\forall c. F_{\mathbb{C}}^{(n-1)}(c) \leq F_n(c)$, so it follows that for all $b'$ we have $F_{\mathbb{C}}^{(n-1)}(\kappa(G(b'))) \leq F_n^{(n-1)}(\kappa(G(b'))).$ Ordering on functions is pointwise, so we have

$$\lambda b'. F_{\mathbb{C}}^{(n-1)}(\kappa(G(b'))) \leq \lambda b'. F_n^{(n-1)}(\kappa(G(b')))$$

For $\text{in}_{\mathbb{C}}$ it also holds that $\leq$ acts component-wise.

$$F_{\mathbb{C}}(\text{in}_{\mathbb{C}}(a,\kappa)) = \text{in}_{\mathbb{C}}(F(a), \lambda b'. F^{(n-1)}(\kappa(G(b'))))$$

$$\leq$$

$$\text{in}_{\mathbb{C}}(F(a), \lambda b'. F_n^{(n-1)}(\kappa(G(b')))) = F_{\mathbb{C}}^{(n+1)}(\text{in}_{\mathbb{C}}(a,\kappa))$$

Lemma 6.4.5 allows us to define $F_{\mathbb{C}}$ and $G_{\mathbb{C}}$ through suprema.

$$F_{\mathbb{C}}(c) = \bigvee_n F^p_{\mathbb{C}}(c) \quad G_{\mathbb{C}}(c) = \bigvee_n G^p_{\mathbb{C}}(c)$$

Finally, we define functions for embedding and retracting interpretations by describing the components of the function families.

$$(F_{\Sigma=\mathcal{D}}(H))_{op, S_1 \rightarrow S_2}(a', \kappa') = \begin{cases} F_D(H_{op}(G_A(a'), G_D \circ \kappa' \circ F_B)) & \text{op : } A \rightarrow B \in \Sigma \land S_1 = A^s \land S_2 = B^s \\ \bot & \text{otherwise} \end{cases}$$

$$(G_{\Sigma=\mathcal{D}}(H'))_{op}(a, \kappa) = G_D(H'_{op, A^s \rightarrow B^s}(F_A(a), F_D \circ \kappa \circ G_B)) \quad (\text{op : } A \rightarrow B \in \Sigma)$$

Most of these constructions only work if $F$ and $G$ are continuous, which is also an important property in later proofs.

**Lemma 6.4.6.**

- For every $\vdash A : \text{type}$, the functions $F_A$ and $G_A$ are continuous.
- For every $\vdash C : \text{type}$, the functions $F_C$ and $G_C$ are continuous.
- For every $\vdash \Sigma : \text{sig}$ and $\vdash D : \text{type}$, the functions $F_{\Sigma=\mathcal{D}}$ and $G_{\Sigma=\mathcal{D}}$ are continuous.

**Proof.** We proceed by induction on the well-formedness judgement.

- For $\text{unit}$ and $\text{int}$, the functions are identities, which are continuous. The empty functions for $\text{empty}$ are continuous by default.

- We show it for $A \times B$ with proofs for $A + B$ and $A \text{ list}$ being similar. Since $F$ and $G$ have nearly identical behaviour for these types, the proof is stated only for $F_{A \times B}$.

  We first confirm that $F$ is monotone. Assume $(a_1, b_1) \leq_{[A \times B]} (a_2, b_2)$, and it follows that $a_1 \leq_{[A]} a_2$ and $b_1 \leq_{[B]} b_2$. By induction we have $F_A(a_1) \leq_{[A]} F_A(a_2)$ and $F_B(b_1) \leq_{[B]} F_B(b_2)$, which gives us

  $$F((a_1, b_1)) = (F(a_1), F(b_1)) \leq (F(a_2), F(b_2)) = F((a_2, b_2)).$$

  For chain-completeness, assume we have a chain $(a_0, b_0) \leq (a_1, b_1) \leq \ldots$ with the supremum $(a, b) = \bigvee_i(a_i, b_i)$. It follows that $a_0 \leq a_1 \leq \ldots$ and $b_0 \leq b_1 \leq \ldots$ are chains with suprema $a$ and $b$ respectively. By induction, $\bigvee_i F(a_i) = F(a)$ and $\bigvee_i F(b_i) = F(b)$, so we conclude

  $$\bigvee_i F((a_i, b_i)) = \bigvee_i (F(a_i), F(b_i)) = (\bigvee_i F(a_i), \bigvee_i F(b_i)) = (F(a), F(b))$$

  $$= F((a, b)) = F(\bigvee_i(a_i, b_i)).$$
• The proof for $A \to C$ and $C \Rightarrow D$ relies on the fact that $F$ and $G$ are composites of continuous functions (thanks to induction hypotheses). We write it out for $F_{A \to C}$, with the other proofs proceeding similarly.

If we assume $f_1 \leq f_2$, it follows that for any $a'$ we have $f_1(G_A(a')) \leq f_2(G_A(a'))$. And since $F_C$ is continuous, it is also monotone, and we have

$$(F_{A \to C}(f_1))(a') = F_C(f_1(G_A(a')) \leq F_C(f_2(G_A(a'))) = (F_{A \to C}(f_2))(a').$$

The above holds for an arbitrary $a'$, so $F(f_1) \leq F(f_2)$.

Let $f_0 \leq f_1 \leq \ldots$ be a chain with the supremum $f$ and let $a' \in \langle A_n \rangle$. For functions it holds that $(\bigvee_i f_i)(x) = \bigvee_i f_i(x)$. By induction we know that $F_C$ is continuous, and therefore $\bigvee_i F_C(f_i) = F_C(\bigvee_i f_i)$.

$$\left(\bigvee_i F_{A \to C}(f_i)\right)(a') = \bigvee_i F_C(f_i(G_A(a'))) = \bigvee_i F_C(f_i(G_A(a'))) = \left(F_{A \to C}(\bigvee_i f_i)\right)(a').$$

• For $C = A^\Sigma/E$, we focus on the proof for $G$, since it is a tad more difficult. By definition $G_C = \bigvee_n G^n_C$, so we start by showing that for all $n$, the function $G^n_C$ is continuous. We proceed by induction on $n$.

For $n = 0$, the function $G^0$ maps everything to $\bot$, so it is both monotone and distributes over suprema. For the induction step we assume $G^{n-1}$ is continuous and try to show that $G^n$ is as well.

To show monotonicity, assume we have $c'_1 \leq c'_2$. We do a case analysis on $c'_1$:

- Suppose $c'_1 = \bot$. Then we have $G^n(c'_1) = \bot \leq G^n(c'_2)$.
- If $c'_1 = \text{in}_{\text{val}}(a'_1)$ then it follows that $c'_2 = \text{in}_{\text{val}}(a'_2)$ and $a'_1 \leq a'_2$. We use the IH for $A$ to obtain $G_A(a'_1) \leq G_A(a'_2)$, and we conclude by

$$G_{A^\Sigma/E}(\text{in}_{\text{val}}(a'_1)) = \text{in}_{\text{val}}(G_A(a'_1)) \leq \text{in}_{\text{val}}(G_A(a'_2)) = G_{A^\Sigma/E}(\text{in}_{\text{val}}(a'_2)).$$

- If $c'_1 = \text{in}_{\text{op}s_1 \to s_2}(a'_1, \kappa'_1)$, then it follows that $c'_2 = \text{in}_{\text{op}s_1 \to s_2}(a'_2, \kappa'_2)$. We split into two further cases.
  - The operation $\text{op}$ in $\Sigma$ has a compatible signature, meaning $S_1 = A_{\text{op}}^*$ and $S_2 = B_{\text{op}}^*$. This means that $G^n$ will map $c'_1$ and $c'_2$ into appropriate $\text{in}_{\text{op}}$ elements. First, we need $G_{A_{\text{op}}}(a'_1) \leq G_{A_{\text{op}}}(a'_2)$, which is true by induction, since $A_{\text{op}}$ is part of $\Sigma$. Secondly, we need to show that for any $b \in [B_{\text{op}}]$, we have

$$\lambda b \cdot G_{\Sigma}(n^{-1})(\kappa'_1(F_{B_{\text{op}}}(b))) \leq \lambda b \cdot G_{\Sigma}(n^{-1})(\kappa'_2(F_{B_{\text{op}}}(b))).$$

From $c_1 \leq c_2$ we know that for every $b'$ we have $\kappa'_1(b') \leq \kappa'_2(b')$, which implies $\kappa'_1(F(b)) \leq \kappa'_2(F(b))$. Finally, $G^{n-1}$ is monotone by induction on $n$, so it holds that $G^{n-1}(\kappa'_1(F(b))) \leq G^{n-1}(\kappa'_2(F(b)))$.

- The operation $\text{op}$ is either not in $\Sigma$, or the types are not compatible. This means that $G^n$ maps to $\bot$ and the proof is complete since $\bot \leq \bot$.

We now show that $G^n$ distributes over suprema. Recall that for chains $(c'_i)_i$, there are only three possibilities:

- The chain is constantly $\bot$.

$$\bigvee_i G^n(c'_i) = \bigvee_i G^n(\bot) = \bigvee_i \bot = \bot = G^n(\bot) = G^n(\bigvee_i \bot) = G^n(\bigvee_i c'_i)$$
The chain may start as a chain of $\perp$, but from some point onwards, elements $c'_i$ are of shape in $\nu a_1(a'_i)$ and $(a'_i)'_h$ is a chain in $(A)'_h$. We know $G_A$ is continuous by induction and therefore distributes over suprema.

$$\forall x, G_{\mathcal{A}/\mathcal{E}}(\nu x a_1(a_i)) = \nu x G_A(a_i) = G_{\mathcal{A}/\mathcal{E}}(\nu x G_A(a_i))$$

$\diamondsuit$ The chain may start as a chain of $\perp$, but from some point onwards elements $c'_i$ are of shape in $\nu_{S_1 \rightarrow S_2}(a'_i, \kappa'_i)$. Elements $(a'_i)'_h$ is a chain in $(S)_v$, and for any $b' \in (S)_v$, elements $(\kappa'_i(b'))_i$ form a chain in $(A)/\mathcal{E}$. We again separate two options.

$\diamondsuit$ There is an operation $op : A \rightarrow B$ in $\Sigma$ with a compatible signature. We keep in mind that $\forall x, y \in op(x, y) = \nu x, y$. This time we first simplify both $\forall x G^n(c)$ and $G^n(x)$.

$$\forall x G^n_{\mathcal{A}/\mathcal{E}}(\nu_{S_1 \rightarrow S_2}(a'_i, \kappa'_i)) = \nu x op(G_{\mathcal{A}/\mathcal{E}}(a'_i), gb, G^{(n-1)}_{\mathcal{A}/\mathcal{E}}(\kappa'_i(F_{B}(b))))$$

For $a'_i$, we use the IH for $A$, which states that $G_A$ is continuous. For the lambdas we use the IH for $n - 1$, which states that $G^{(n-1)}_{\mathcal{A}/\mathcal{E}}$ is continuous, so the suprema distributes over it.

$\diamondsuit$ The operation $op$ is either not in $\Sigma$, or the types are not compatible. Then $G^n$ maps the supremum and every element of the chain to $\perp$, and we have $\perp \leq \perp$. We now translate the results to $G_{\mathcal{C}}$. For chains with two indices, it holds that $\forall x, y \in op(x, y) = \nu x, y$.

$$\forall x G^n_{\mathcal{C}}(c_i) = \forall x G^n_{\mathcal{C}}(c_i) = \forall x G^n_{\mathcal{C}}(c_i) = G_{\mathcal{C}}(\nu x c_i)$$

For interpretations, we prove it for $F$, the harder of the two proofs.

We first check that $F_{\mathcal{A} = \mathcal{D}}$ is monotone. Assume $H \leq \text{interp}_{\mathcal{D}} H'$. We must prove that $(F(H))_{op_{S_1 \rightarrow S_2}} \leq (F(H'))_{op_{S_1 \rightarrow S_2}}$ for all $op : S_1 \rightarrow S_2 \in \Omega$. We have two options depending on $op$ and $\Sigma$.

$\diamondsuit$ Assume $op : A \rightarrow B \in \Sigma$ with $A \leq S_1$ and $B \leq S_2$. Then we need to show that for any $a'$ and $\kappa'$, it holds that

$$F_{\mathcal{D}}(H_{op}(G_{\mathcal{A}}(a'), G_{\mathcal{D}} \circ \kappa' \circ F_B)) \leq F_{\mathcal{D}}(H'_{op}(G_{\mathcal{A}}(a'), G_{\mathcal{D}} \circ \kappa' \circ F_B)).$$

We apply the fact that $F_{\mathcal{D}}$ is continuous by IH, and if we denote $a = G_{\mathcal{A}}(a')$ and $\kappa = G_{\mathcal{D}} \circ \kappa' \circ F_B$, we are left with showing

$$H_{op}(a, \kappa) \leq H'_{op}(a, \kappa).$$

This follows directly from the fact that $H \leq H'$. 91
Assume \( op \) has no suitable type-assignment in \( \Sigma \). Then the components for \( op_{S_1 \rightarrow S_2} \) of both \( F(H) \) and \( F(H') \) are the constant \( \bot \) function.

The proof for chain-completeness is similar. We write \( H^{op} \) instead of \( H_{op_{S_1 \rightarrow S_2}} \) for the components of interpretations to allow for chain indices. For every \( op : S_1 \rightarrow S_2 \in \Omega \), we need to show

\[
(F_{\Sigma = D}(\forall_i H_i))^{op} = \forall_i (F_{\Sigma = D}(H_i))^{op}.
\]

We separate two cases.

1. Assume \( op : A \rightarrow B \in \Sigma \) with \( A^s = S_1 \) and \( B^s = S_2 \). Then we need to show that for any \( a' \) and \( \kappa' \), it holds that

\[
F_{D}(\forall_i H_i)^{op}(G_A(a'), G_D \circ \kappa' \circ F_B)) = \forall_i F_D(H_i)^{op}(G_A(a'), G_D \circ \kappa' \circ F_B)).
\]

The function \( F_D \) is continuous by IH for \( D \), so it commutes with suprema. If we denote \( a = G_A(a') \) and \( \kappa = G_D \circ \kappa' \circ F_B \), we need to show

\[
(\forall_i H_i)^{op}(a, \kappa) = \forall_i H_i^{op}(a, \kappa).
\]

This follows from the fact that suprema are pointwise for functions.

2. Assume \( op \) has no suitable case in \( \Sigma \). Then the maps of all \( H_i \) as well as \( \forall_i H_i \) have the component \( op_{S_1 \rightarrow S_2} \) set to a constant \( \bot \) function.

\[\Box\]

Before we start the proof of the fact that \( F \) and \( G \) are indeed an embedding-retraction pair, we require some additional tools. The induction principle for elements of the computation domains requires admissibility, which is too big of a restriction. As an alternate approach, we define order "up to \( n \) levels deep": \((\leq^n) \subseteq [\mathbb{C}] \times [\mathbb{C}]\). We say that \( c_1 \leq^n c_2 \) if and only if any of the following holds:

\begin{itemize}
  \item \( n = 0 \),
  \item \( c_1 \) is \( \bot \),
  \item \( c_1 = \text{inv}_{a1}(a_1) \) and \( c_2 = \text{inv}_{a1}(a_2) \), and \( a_1 \leq a_2 \),
  \item \( c_1 = \text{in}_{op}(a_1, k_1) \) and \( c_2 = \text{in}_{op}(a_2, k_2) \), where \( a_1 \leq a_2 \) and \( \forall b. k_1(b) \leq^{(n-1)} k_2(b) \).
\end{itemize}

Transitivity of \( \leq^n \) can be shown with a straightforward induction on \( n \). We use the notation \( c_1 =^n c_2 \) to mean \( c_1 \leq^n c_2 \) and \( c_2 \leq^n c_1 \).

The relation \( \leq^n \) allows induction on \( n \), but we also need a way to transfer results for \( \leq^n \) to results for \( \leq \). More importantly, this allows us to use \( =^n \) as a tool to prove equality on two possibly infinite effect trees.

**Lemma 6.4.7.** For any \( c_1, c_2 \in [\mathbb{C}] \), we have \( (\forall n. \ c_1 \leq^n c_2) \iff c_1 \leq c_2 \)

**Proof.** The proof is done in two stages. We first prove \( (\forall n. \ c_1 \leq^n c_2) \iff c_1 \leq c_2 \), because we need to use this property in the proof of \( (\forall n. \ c_1 \leq^n c_2) \Rightarrow c_1 \leq c_2 \).

\((\Leftarrow):\) We restate the lemma to the equivalent \( (\forall c_1, c_2. \ c_1 \leq c_2 \Rightarrow c_1 \leq^n c_2) \), which we prove by induction on \( n \). For \( n = 0 \) this follows from the definition of \( \leq^0 \). For the induction step, we assume \( (\forall c_1, c_2. \ c_1 \leq c_2 \Rightarrow c_1 \leq^{(n-1)} c_2) \), and prove it for \( n \). Assume \( c_1 \leq c_2 \) and proceed by case analysis on shape of \( c_1 \).

\begin{itemize}
  \item If \( c_1 = \bot \), then \( \bot \leq^n c_2 \) by definition.
  \item If \( c_1 = \text{inv}_{a1}(a_1) \), then from \( c_1 \leq c_2 \) it follows that \( c_2 = \text{inv}_{a1}(a_2) \) and \( a_1 \leq a_2 \). This satisfies the conditions for \( \text{inv}_{a1}(a_1) \leq^n \text{inv}_{a1}(a_2) \).
  \item If \( c_1 = \text{in}_{op}(a_1, k_1) \), it follows from \( c_1 \leq c_2 \) that \( c_2 = \text{in}_{op}(a_2, k_2) \), with \( a_1 \leq a_2 \) and \( \forall b. k_1(b) \leq k_2(b) \). The only property left to show is \( \forall b. k_1(b) \leq^{(n-1)} k_2(b) \), which follows by induction.
\end{itemize}
Lemma 6.4.8: We show that the predicate $\varphi(c) = \forall d. (\forall n. c \leq^n d) \Rightarrow c \leq d$ holds for all $c$ by the induction principle of \(\mathbb{I}\). This requires us to first show that $\varphi$ is an admissible predicate. Admissibility: The proof for $\varphi(\bot)$ is trivial, because $\bot \leq d$. To show that $\varphi$ is chain complete, assume a chain $(c_i)_i$ with $\varphi(c_i)$ for all $i$. We need to show $\varphi(\bigvee_i c_i)$, which states

$$\forall d. (\forall n. \bigvee_i c_i \leq^n d) \Rightarrow \bigvee_i c_i \leq d.$$ 

We choose $d$ as an arbitrary element of $\mathbb{I}$ and assume $\forall n. \bigvee_i c_i \leq^n d$. The supremum is an upper bound, so $c_i \leq \bigvee_i c_i$ for any $i$. From the previously shown ($\Leftarrow$) of this lemma, we get $\forall n. c_i \leq^n \bigvee_i c_i$. We combine it with the assumption that $\forall n. \bigvee_i c_i \leq^n d$ to arrive at $\forall n. c_i \leq^n d$. The assumption is that $\forall i. \varphi(c_i)$, and since for every $i$, we have $\forall n. c_i \leq^n d$, this implies $\forall i. c_i \leq d$. It follows that $\bigvee_i c_i \leq d$, since a supremum is the least upper bound.

Induction: We follow the induction principle from Section 6.1 to show $\forall c \in A!\Sigma/\mathcal{E}. \varphi(c)$. 

- Assume $a \in \llbracket A \rrbracket$. We need to show $\forall d. (\forall n. \in_{\text{val}}(a) \leq^n d) \Rightarrow \in_{\text{val}}(a) \leq d$.

Even in_{\text{val}}(a) \leq^1 d suffices to show $d = \in_{\text{val}}(a')$ for some $a'$ and $a \leq a'$. It follows that in_{\text{val}}(a) \leq in_{\text{val}}(a') = d.

- Assume $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma$. We need to show $\forall d. (\forall n. \in_{\text{op}}(a, \kappa) \leq^n d) \Rightarrow \in_{\text{op}}(a, \kappa) \leq d$ under the hypothesis $\forall b. \forall d. (\forall n. \kappa(b) \leq^n d) \Rightarrow \kappa(b) \leq d$. We choose an arbitrary $d$ and assume $\forall n. \in_{\text{op}}(a, \kappa) \leq^n d$.

For any non-zero $n$, the assumption $\in_{\text{op}}(a, \kappa) \leq^n d$ implies that $d = \in_{\text{op}}(a', \kappa')$ for some $a', \kappa'$. It also holds that $a \leq a'$ and $\forall b. \kappa(b) \leq^{(n-1)} \kappa'(b)$. Repeating the process for all non-zero $n$, we obtain $\forall b. \forall n \geq 0 \Rightarrow \kappa(b) \leq^{(n-1)} \kappa'(b)$, which is equivalent to $\forall b. \forall n. \kappa(b) \leq^n \kappa'(b)$. By the induction hypothesis, we arrive at $\forall b. \kappa(b) \leq \kappa'(b)$. We therefore have $a \leq a'$ and $\kappa(b) \leq \kappa'(b)$ for any $b$, so $\in_{\text{op}}(a, \kappa) \leq \in_{\text{op}}(a', \kappa')$ follows.

\[\square\]

Lemma 6.4.8.

(a) For all $a \in \llbracket A \rrbracket$ we have $a \vdash_A F_A(a)$.

(b) If $a \vdash_A a'$, then $G_A(a') = a$.

(c) For all $c \in \llbracket C \rrbracket$ we have $c \vdash_C F_C(c)$.

(d) If $c \vdash_C c'$, then $G_C(c') = c$.

(e) For all $H \in \text{interp}_\Sigma(\mathcal{D})$ we have $H \vdash_{\Sigma \Rightarrow \mathcal{D}} F_{\Sigma \Rightarrow \mathcal{D}}(H)$.

(f) If $H \vdash_{\Sigma \Rightarrow \mathcal{D}} H'$, then $G_{\Sigma \Rightarrow \mathcal{D}}(H') = H$.

Proof. We need to prove all of the parts at the same time. The proof proceeds by induction on the structure of the type.

Proofs of (a) and (b)

- Proofs for empty, unit, and int are trivial.
• We state the proof for $A \times B$ and proofs for $A + B$ and $A \text{list}$ proceed similarly.

To show part (a), assume we have an element $(a, b) \in \llbracket A \times B \rrbracket$. By induction we know that $a \triangleright F(a)$ and $b \triangleright F(b)$. We obtain $(a, b) \triangleright F((a, b))$ by definition of $\triangleright$ and $F$.

For part (b), assume we have $(a, b) \simeq (a', b')$. This implies $a \simeq a'$ and $b \simeq b'$, so $a = G(a')$ and $b = G(b')$ by induction; therefore $G((a', b')) = (G(a'), G(b')) = (a, b)$.

• Proofs for $A \rightarrow C$ and $C \Rightarrow D$ are similar due to similar definitions for $\triangleright$, $F$, and $G$.

To show $f \triangleright_{A \rightarrow C} F_{A \rightarrow C}(f)$, assume $f \in \llbracket A \rightarrow C \rrbracket$ and $a \triangleright_{A} a'$. We need to show $f(a) \triangleright_{C} (F_{A \rightarrow C}(f))(a')$. The induction hypothesis for $A$ gives us $G(a') = a$ because $a \triangleright_{A} a'$, and the IH for $C$ gives us $f(a) \triangleright_{C} F_{C}(f(a))$.

This gives us the desired $f(a) \triangleright (F(f))(a')$ from which we conclude $f \triangleright F(f)$.

For part (b), assume $f \triangleright_{A \rightarrow C} f'$. To show function equality, we need to show that $f$ and $G_{A \rightarrow C}(f')$ match for all $a \in \llbracket A \rrbracket$. The induction hypothesis for $A$ gives us $a \triangleright F(a)$, and because of the assumption $f \triangleright f'$, we know $f(a) \triangleright_{C} F_{C}(f(a))$. Using IH of (d) for $C$, we show

$$f(a) = G_{C}(f'(F(a))) = (G_{A \rightarrow C}(f'))(a).$$

**Proof of (c)**

Note that since $\Sigma$ is part of the type, we have induction hypotheses for all the types occurring in the signature. Our goal is to show $c \triangleright_{A!\Sigma/E} F(c)$, which we do by a second induction on the structure of $c$. We must first show that $\varphi(c) = (c \triangleright_{A!\Sigma/E} F(c))$ is an admissible predicate.

For $\varphi(\bot) = (\bot \triangleright F(\bot))$, we know that $F$ is strict and that $\bot \triangleright \bot$. Now we assume we have a chain $(c_i)$ and $\forall i. \varphi(c_i)$ holds. We know that $(F(c_i))_i$ is a chain due to continuity of $F$ (Lemma 6.4.6). We have two chains with $\forall i. c_i \triangleright F(c_i)$, and since the $\triangleright$ relation is chain complete (Lemma 6.4.1), we conclude $\bigvee_i c_i \triangleright \bigvee_i F(c_i)$. By continuity suprema distribute over $F$, and we arrive at $(\bigvee_i c_i \triangleright F(\bigvee_i c_i)) = \varphi(\bigvee_i c_i)$.

We now proceed with induction on structure of $c \in \llbracket A!\Sigma/E \rrbracket$.

• Let $c = \text{in}_{\text{val}}(a)$. From induction on types, we have $a \triangleright F(a)$, and it follows that $\text{in}_{\text{val}}(a) \triangleright \text{in}_{\text{val}}(F(a))$, which is equal to $\text{in}_{\text{val}}(a) \triangleright F(\text{in}_{\text{val}}(a))$.

• Let $c = \text{in}_{\text{op}}(a, \kappa)$ for $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma$. We need to show

$$\text{in}_{\text{op}}(a, \kappa) \triangleright_{A!\Sigma/E} \text{in}_{\text{op}}_{A_{\text{op}} \rightarrow B_{\text{op}}}(F_{A_{\text{op}}}(a), A b') . F_{A!\Sigma/E}( \kappa(G_{B_{\text{op}}}(b'))).$$

This is reduced to showing that $a \triangleright F(a)$, which is true by IH on $A_{\text{op}}$, and that for $b \triangleright b'$, we have $\kappa(b) \triangleright F(\kappa(G(b')))$. The hypothesis of the computation induction is $\forall b. \kappa(b) \triangleright F(\kappa(b))$.

If we assume $b \triangleright b'$ the IH on $B_{\text{op}}$ gives us $b = G(b')$, which we combine with the IH for $\kappa$ to conclude $\kappa(b) \triangleright F(\kappa(G(b')))$. This shows $\kappa(b) \triangleright F(\kappa(G(b'))) = c$.

**Proof of (d)**

We have to show that $c \triangleright_{A!\Sigma/E} c'$ implies $G_{A!\Sigma/E}(c') = c$. We start by showing a weaker version, in which $c \triangleright^n c'$ implies $G(c') =^n c$. We proceed by induction on $n$. The case for $n = 0$ is trivial, because the right side always holds. For the inductive step we assume it holds for $n − 1$ and analyse the cases of $c \triangleright^n c'$.
• Both $c$ and $c'$ are $\bot$. Then $G(c') = \bot$ and $\bot =^n \bot$.

• $c = \text{in}_{\text{val}}(a)$ and $c' = \text{in}_{\text{val}}(a')$ where $a \prec_A a'$. The IH for $A$ states that $G(a') = a$ and it follows that $G(\text{in}_{\text{val}}(a')) = \text{in}_{\text{val}}(G(a')) = \text{in}_{\text{val}}(a)$. We use Lemma 6.4.7 to weaken equality to $=^n$ by weakening to $\leq^n$.

• $c = \text{in}_{\text{op}}(a, k)$ and $c' = \text{in}_{\text{op}_{S_1 \rightarrow S_2}}(a', k')$ where $a \prec_{A_{\text{op}}} a'$, and if $b \prec_{B_{\text{op}}} b'$, then $k(b)^{(n-1)} \sim_{A_{\Sigma} / \varepsilon} k'(b')$, with the side condition that $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma$ and $A_{\text{op}}^s = S_1$, $B_{\text{op}}^s = S_2$. Thanks to this side condition, we know that

$$G_{A_{\Sigma} / \varepsilon}(\text{in}_{\text{op}_{S_1 \rightarrow S_2}}(a', k')) = \text{in}_{\text{op}}(G_{A_{\text{op}}}(a'), \lambda b . G_{A_{\Sigma} / \varepsilon}(k'(F_{B_{\text{op}}}(b))))).$$

To check $c =^n G(c')$, we need $G(a') = a$ and $\forall b . k(b)^{(n-1)} = G(k'(F(b)))$. The first requirement is precisely the IH for $A_{\text{op}}$, since $a \prec a'$. For the second requirement we chose an arbitrary $b$, and by IH on $B_{\text{op}}$ we have $b \prec F(b)$. It follows that $k(b)^{(n-1)} = G(k'(F(b)))$, which enables us to use the IH for induction on $n$ to arrive at $G(k(b)) = G(k'(F(b)))$.

All that is left to do is to simplify the result to $\sim$. If $c \sim c'$, then $\forall n . c \sim^n c'$ by definition. This means that we have $\forall n . G(c') =^n c$, so by Lemma 6.4.7 it holds that $G(c') = c$.

**Proof of (e)**

To show $H \times_{\Sigma \rightarrow D} F_{\Sigma \rightarrow D}(H)$ we need to prove that for every $\text{op} : A \rightarrow B$, we have

$$a \sim_A a' \land k \sim_{B \rightarrow D} k' \Rightarrow H_{\text{op}}(a, k) \sim (F(H))_{\text{op}_{A \rightarrow B_1}}(a', k').$$

Assume some $a \sim a'$ and $k \sim k'$. We simplify the right side of the x-ray.

$$(F_{\Sigma \rightarrow D}(H))_{\text{op}}(a', k') = F_D(H_{\text{op}}(G_A(a'), G_D \circ k' \circ F_B))$$

We know that $a \sim a'$, so by IH we have $G_A(a') = a$. For any $b$, we know by induction that $b \sim_B F_B(b)$, and since $k \sim_{B \rightarrow D} k'$, it also holds that $k(b) \sim_D k'(F(b))$. The IH for $D$ gets us $G(k'(F(b))) = k(b)$ for any $b$, so $k = G_D \circ k' \circ F_B$. We now simplify our goal with the two new equalities.

$$(F_{\Sigma \rightarrow D}(H))_{\text{op}}(a', k') = F_D(H_{\text{op}}(a, k))$$

By IH on $D$ we have $H_{\text{op}}(a, k) \sim F_D(H_{\text{op}}(a, k)) = (F_{\Sigma \rightarrow D}(H))_{\text{op}}(a', k')$. Since this holds for any $\text{op}$, $a \sim a'$, and $k \sim k'$, we obtain $H \sim F(H)$.

**Proof of (f)**

Assume $H \times_{\Sigma \rightarrow D} H'$. We wish to show $G_{\Sigma \rightarrow D}(H') = H$, which we do by showing equality on every component. Assume $\text{op} : A \rightarrow B \in \Sigma$, $a \in [A]$ and $k \in [B] \rightarrow [D]$.

$$(G_{\Sigma \rightarrow D}(H'))_{\text{op}}(a, k) = G_D(H'_{\text{op}_{A \rightarrow B_1}}(F_A(a), F_D \circ k \circ G_B))$$

By induction we know that $a \sim F(a)$. To show $k \sim F_D \circ k \circ G_B$, we take an arbitrary related pair $b \sim b'$. By IH on $B$ we have $G_B(b') = b$, which is combined with the IH for $D$ to get $k(b) = k(G(b')) \sim_D F_D(k(G(b')))$, so $k \sim F \circ k \circ G$.

Since $H \sim H'$, we know

$$a \sim a' \land k \sim k' \Rightarrow H_{\text{op}}(a, k) \sim H'_{\text{op}_{A \rightarrow B_1}}(a', k').$$

By setting $a'$ to $F(a)$ and $k'$ to $F \circ k \circ G$ in the above implication, we obtain

$$H_{\text{op}}(a, k) \sim H'_{\text{op}_{A \rightarrow B_1}}(F(a), F \circ k \circ G).$$
Now we use of induction hypothesis for $G_D$ to get
\[(G_{=D}(H'))_{op}(a, \kappa) = G_{D}(H'_{op, A^* \to A^*}(F(a), F \circ \kappa \circ G)) = H_{op}(a, \kappa).\]
Since this holds for all $op$, $a$, and $\kappa$, we conclude that $G(H') = H$.

\[\square\]

**Proposition 6.4.9.** The definition of denotational semantics from Section 6.2 is coherent.

**Proof.** Assume we have two derivations of $\Gamma \vdash v : A$, denoted with $p_1$ and $p_2$, together with an arbitrary $\eta \in [\Gamma]$. By Lemma 6.3.2 we know that $\Gamma^x \vdash v^x : A^x$ has a unique derivation $p_x$ (Lemma 6.3.3). From Lemma 6.4.8 it follows that $\eta \cong_{\Gamma} F_{\Gamma}(\eta)$. By Lemma 6.4.4 it follows that $\Gamma^x \vdash v^x : A^x$ has a unique derivation $p_x$ (Lemma 6.3.3). From Lemma 6.4.8 it follows that $\eta \cong_{\Gamma} F_{\Gamma}(\eta)$. By Lemma 6.4.4 it follows that $\eta \cong_{\Gamma} F_{\Gamma}(\eta)$. Therefore, by Lemma 6.4.8 we have $\Gamma^x \vdash v^x : A^x$. The same proof can be applied to computations and operation cases.

\[\square\]

### 6.5 Properties

**Lemma 6.5.1 (Context weakening).** Let $\eta_1 \in [\Gamma_1]$, $\eta_2 \in [\Gamma_2]$ and $b' \in [B']$. Assume $x'$ is a fresh variable in the following terms.

- If $\Gamma_1, \Gamma_2 \vdash v : A$, then
  \[[\Gamma_1, \Gamma_2 \vdash v : A](\eta_1, \eta_2) = [\Gamma_1, x' : B', \Gamma_2 \vdash v : A](\eta_1, b', \eta_2).

- If $\Gamma_1, \Gamma_2 \vdash c : C$, then
  \[[\Gamma_1, \Gamma_2 \vdash c : C](\eta_1, \eta_2) = [\Gamma_1, x' : B', \Gamma_2 \vdash c : C](\eta_1, b', \eta_2).

- If $\Gamma_1, \Gamma_2 \vdash h : \Sigma \Rightarrow D$, then
  \[[\Gamma_1, \Gamma_2 \vdash h : \Sigma \Rightarrow D](\eta_1, \eta_2) = [\Gamma_1, x' : B', \Gamma_2 \vdash h : \Sigma \Rightarrow D](\eta_1, b', \eta_2).

**Proof.** The prerequisite of the proof consists of a context-weakening lemma for typing derivations, which states that $\Gamma_1, \Gamma_2 \vdash v : A$ implies $\Gamma_1, x' : B', \Gamma_2 \vdash v : A$. This is done in the formalisation of EEF as part of the substitution lemma, and it shows that the structure of derivation tree does not change. The denotational part is done by induction on the typing derivation of $\Gamma_1, \Gamma_2 \vdash v : A$. For brevity, we shorten $[\Gamma_1, \Gamma_2 \vdash v : A](\eta_1, \eta_2)$ to $[v](\eta_1, \eta_2)$ and $[\Gamma_1, x' : B', \Gamma_2 \vdash v : A](\eta_1, b', \eta_2)$ to $[v](\eta_1, b', \eta_2)$. We only list a few cases because they are very straightforward.

- **TYPEVAR:** Assume $\Gamma_1, \Gamma_2 \vdash x : A$. By assumption of freshness, $x \neq x'$. The shift from $\Gamma_1, \Gamma_2$ to $\Gamma_1, x' : B', \Gamma_2$ changes the projection, so that the value projected from $(\eta_1, \eta_2)$ matches the one projected from $(\eta_1, b', \eta_2)$.

- **TYPECONS:** The induction hypotheses are $[v_1](\eta_1, \eta_2) = [v_1](\eta_1, b', \eta_2)$ and also $[v_2](\eta_1, \eta_2) = [v_2](\eta_1, b', \eta_2)$.

\[ [v_1 :: v_2](\eta_1, \eta_2) = [v_1](\eta_1, \eta_2) :: [v_2](\eta_1, \eta_2) = [v_1](\eta_1, b', \eta_2) :: [v_2](\eta_1, b', \eta_2) = [v_1 :: v_2](\eta_1, b', \eta_2) \]
• **TYPEFUN**: For any \( a \in \llbracket A \rrbracket \), we have \( \llbracket c \rrbracket(\eta_1, \eta_2, a) = \llbracket c \rrbracket(\eta_1, b, \eta_2, a) \) by induction.

\[
\llbracket \text{fun } (x : A) \mapsto c \rrbracket(\eta_1, \eta_2) = \lambda a . \llbracket c \rrbracket(\eta_1, \eta_2, a) = \lambda a . \llbracket c \rrbracket(\eta_1, b', \eta_2, a) = \llbracket \text{fun } (x : A) \mapsto c \rrbracket(\eta_1, b', \eta_2)
\]

• **TYPEPRODMATCH**: The induction hypotheses are \( \llbracket v \rrbracket(\eta_1, \eta_2) = \llbracket v \rrbracket(\eta_1, b', \eta_2) = (a, b) \) and \( \llbracket c \rrbracket(\eta_1, \eta_2) = \llbracket c \rrbracket(\eta_1, b', \eta_2, a, b) \). The rest follows.

• **TYPEDO**: By induction we have \( \llbracket c_1 \rrbracket(\eta_1, \eta_2) = \llbracket c_1 \rrbracket(\eta_1, b', \eta_2) \), and also that for any \( a \in \llbracket A \rrbracket \) it holds that \( \llbracket c_2 \rrbracket(\eta_1, \eta_2, a) = \llbracket c_2 \rrbracket(\eta_1, b', \eta_2, a) \).

\[
\llbracket \text{do } x \leftarrow c_1 \text{ in } c_2 \rrbracket(\eta_1, \eta_2) = \text{lift}_{\Gamma_1}^{\Sigma}((\lambda a . \llbracket c_2 \rrbracket(\eta_1, \eta_2, a))(\llbracket c_1 \rrbracket(\eta_1, \eta_2)))
\]

\[
= \text{lift}_{\Gamma_1}^{\Sigma}((\lambda a . \llbracket c_2 \rrbracket(\eta_1, b', \eta_2, a))(\llbracket c_1 \rrbracket(\eta_1, b', \eta_2)))
\]

\[
= \llbracket \text{do } x \leftarrow c_1 \text{ in } c_2 \rrbracket(\eta_1, b', \eta_2)
\]

\[\square\]

**Lemma 6.5.2**. Assume \( \Gamma_1, \Gamma_2 \vdash u : B \) and let \( \eta_1 \in \llbracket \Gamma_1 \rrbracket \) and \( \eta_2 \in \llbracket \Gamma_2 \rrbracket \).

\[
\llbracket \Gamma_1, x' : B, \Gamma_2 \vdash v : A \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \llbracket \Gamma_1, \Gamma_2 \vdash v[x' \mapsto u] : A \rrbracket(\eta_1, \eta_2)
\]

\[
\llbracket \Gamma_1, x' : B, \Gamma_2 \vdash c : C \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \llbracket \Gamma_1, \Gamma_2 \vdash c[x' \mapsto u] : \llbracket C \rrbracket(\eta_1, \eta_2)
\]

\[
\llbracket \Gamma_1, x' : B, \Gamma_2 \vdash h : \Sigma \Rightarrow D \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \llbracket \Gamma_1, \Gamma_2 \vdash h[x' \mapsto u] : \Sigma \Rightarrow D \rrbracket(\eta_1, \eta_2)
\]

**Proof.** The proof proceeds by induction on the typing derivation of the left term. We only cover a few cases to convey the general idea of the proof. We always ensure that bound variables have a name different from \( x' \) (otherwise, we \( \alpha \)-rename the variable). We sometimes shorten \( \llbracket \Gamma_1, x' : B, \Gamma_2 \vdash v : A \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) \) to \( \llbracket v \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) \) and we shorten \( \llbracket \Gamma_1, \Gamma_2 \vdash v[x' \mapsto u] : A \rrbracket(\eta_1, \eta_2) \) to \( \llbracket v[x' \mapsto u] \rrbracket(\eta_1, \eta_2) \).

• **TYPEVAR**: The case of \( (\Gamma_1, x : B, \Gamma_2 \vdash x' : A) \) is the most relevant case, as it directly deals with values from the context. There are two options.

  □ Suppose \( x \neq x' \); then substitution has no effect on \( x' \). In both cases, we end up with the appropriate projection from either \( \eta_1 \) or \( \eta_2 \), depending on the position of \( x' \) in the context.

  □ Suppose \( x = x' \). On the left we have

\[
\llbracket \Gamma_1, \Gamma_2 \vdash x : A \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \llbracket u \rrbracket(\eta_1, \eta_2),
\]

and the right side simplifies to

\[
\llbracket \Gamma_1, \Gamma_2 \vdash x[x' \mapsto u] : A \rrbracket(\eta_1, \eta_2) = \llbracket \Gamma_1, \Gamma_2 \vdash u : B \rrbracket(\eta_1, \eta_2).
\]

• **TYPERIGHT**: By induction we have \( \llbracket v \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \llbracket v[x' \mapsto u] \rrbracket(\eta_1, \eta_2) \).

\[
\llbracket \text{Right}_{A+B} v \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2) = \nu_2(\llbracket v \rrbracket(\eta_1, \llbracket u \rrbracket(\eta_1, \eta_2), \eta_2))
\]

\[
= \nu_2(\llbracket v[x' \mapsto u] \rrbracket(\eta_1, \eta_2))
\]

\[
= \llbracket \text{Right}_{A+B} v \rrbracket[x' \mapsto u] \rrbracket(\eta_1, \eta_2)
\]

97
• **TYPEHANDLER**: The first IH states \([h](\eta_1, [u](\eta_1, \eta_2), \eta_2) = [h[x \mapsto u]](\eta_1, \eta_2)\). The induction hypothesis for \(c_r\) is more interesting, as it features an extended context.

\[ [c_r](\eta_1, [u](\eta_1, \eta_2, \eta_2, \eta_2), \eta_2, \eta_2) = [c_r[x \mapsto u]](\eta_1, \eta_2, \eta_2) \]

This is not directly applicable, which becomes clear when we simplify the denotation of the handler.

\[ [\text{handler} \,(\text{ret} \,(x : A) \mapsto c_r ; h)](\eta_1, [u](\eta_1, \eta_2), \eta_2) \]

\[ = \text{lift}\,[\eta_1, [u](\eta_1, \eta_2), \eta_2](\lambda a \cdot [A][c_r \,(x : A) \mapsto u](\eta_1, \eta_2, \eta_2, \eta_2), \eta_2, \eta_2) \]

The structure does not exactly match our induction hypothesis, due to the discrepancy between \([u](\eta_1, \eta_2)\) and \([u](\eta_1, \eta_2, \eta_2)\). We fix it by applying Lemma 6.5.1.

\[
\ldots = \text{lift}\,[\eta_1, [u](\eta_1, \eta_2), \eta_2](\lambda a \cdot [c_r \,(x : A) \mapsto u](\eta_1, \eta_2, \eta_2, \eta_2), \eta_2, \eta_2) \\
 = \text{lift}\,[\eta_1 \,(x : A) \mapsto c_r ; h[x \mapsto u]](\eta_1, \eta_2, \eta_2) \\
\]

• **TYPESUMMATCH**: The IH gives us \([v](\eta_1, [u](\eta_1, \eta_2), \eta_2) = [v[x \mapsto u]](\eta_1, \eta_2)\). We separate two cases based on the shape of \([v]\).

  - Assume \([v](\eta_1, [u](\eta_1, \eta_2), \eta_2) = \iota_1(a)\). From the IH for \(c_1\) it follows that

    \[
    \text{match } v \text{ with Left } x \mapsto c_1 | \text{Right } y \mapsto c_2(\eta_1, [u](\eta_1, \eta_2), \eta_2) \\
    = [c_1](\eta_1, [u](\eta_1, \eta_2), \eta_2, \eta_2) \\
    = [c_1](\eta_1, [u](\eta_1, \eta_2, \eta_2), \eta_2, \eta_2) \\
    = [c_1[x \mapsto u]](\eta_1, \eta_2, \eta_2) \\
    = [[\text{match } v \text{ with Left } x \mapsto c_1 | \text{Right } y \mapsto c_2][x \mapsto u]](\eta_1, \eta_2, \eta_2) 
    \]

  - The other option is \([v](\eta_1, [u](\eta_1, \eta_2), \eta_2) = \iota_2(b)\) with a similar proof.

• **TYPEHANDLE**: By induction we have \([v](\eta_1, [u](\eta_1, \eta_2), \eta_2) = [v[x \mapsto u]](\eta_1, \eta_2)\) and \([c](\eta_1, [u](\eta_1, \eta_2), \eta_2) = [c[x \mapsto u]](\eta_1, \eta_2)\).

\[
\text{with v handle c}(\eta_1, [u](\eta_1, \eta_2), \eta_2) \\
= ([v](\eta_1, [u](\eta_1, \eta_2), \eta_2))(\,([c](\eta_1, [u](\eta_1, \eta_2), \eta_2)) \\
= ([v[x \mapsto u]](\eta_1, \eta_2))(\,([c[x \mapsto u]](\eta_1, \eta_2)) \\
= ([[\text{with v handle c}][x \mapsto u]](\eta_1, \eta_2, \eta_2) 
\]

• **TYPECASES**: The denotation of well-typed operation cases

\[
\Gamma_1, x' : B, \Gamma_2 \vdash h \cup \{ \text{op}_{A \rightarrow B} (x; k) \mapsto c_{op} \} : (\Sigma \cup \{ \text{op} : A \rightarrow B \}) \Rightarrow \square 
\]

is a family of functions. By induction we know that the lemma holds for components in \(h\) that cover \(\Sigma\). We need to check that this also holds for \(\text{op}\). We use the induction hypothesis for \(c_{op}\) together with the weakening Lemma 6.5.1.

\[
[\lambda a \cdot \lambda k . c_{op} \,[\eta_1, [u](\eta_1, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2), \eta_2, \eta_2, \eta_2, \eta_2)](\eta_1, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2) \\
\]

Therefore, the two evaluations agree on all components of \(\Sigma \cup \{ \text{op} : A \rightarrow B \}\).

\[
[\text{h} \cup \{ \text{op}_{A \rightarrow B} (x; k) \mapsto c_{op} \}][x \mapsto u](\eta_1, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2, \eta_2) \\
\]
Theorem 6.5.3 (Soundness). Assume a well-typed computation $\Gamma \vdash c : C$ and that $c \leadsto c'$. Then it holds that $\llbracket \Gamma \vdash c : C \rrbracket = \llbracket \Gamma \vdash c' : C \rrbracket$.

Proof. Substitution plays a major role in operational semantics, which makes Lemma 6.5.2 one of the key tools in this proof. The proof proceeds by induction on derivation of $c \leadsto c'$. The operational semantics of match statements is only shown for lists, with pairs and sums having a similar proof.

- **APPFUN**: The proof is a rather direct application of Lemma 6.5.2.

$$\llbracket (\lambda x : A . c) \eta \rrbracket = (\llbracket \lambda x : A . c \eta \rrbracket)(\llbracket \eta \rrbracket) = (\lambda a . \llbracket c \eta, a \rrbracket)(\llbracket \eta \rrbracket) = \llbracket c \eta \rrbracket(\llbracket \eta \rrbracket) = \llbracket c(x \mapsto \eta) \rrbracket(\eta)$$

- **LETRECSTEP**: The rule LETRECSTEP states

$$\begin{align*}
\llbracket \text{let rec } f \ x : A \rightarrow C = c_1 \text{ in } c_2 \leadsto c_2[f \mapsto (\text{fun } y : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])] \rrbracket
\end{align*}$$

The semantics of recursive functions include a fixpoint, and in this case there will be two—$\tilde{f}_1$ and $\tilde{f}_2$. We first simplify both sides.

$$\llbracket \text{let rec } f \ x : A \rightarrow C = c_1 \text{ in } c_2 \rrbracket \eta = \llbracket c_2 \rrbracket(\eta, \tilde{f}_1)$$

$$\begin{align*}
\llbracket c_2[f \mapsto (\text{fun } y : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])] \rrbracket \eta \\
= \llbracket c_2 \rrbracket(\eta, \llbracket \text{fun } y : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y] \rrbracket) \eta \\
= \llbracket c_2 \rrbracket(\eta, \lambda a . \llbracket \text{let rec } f \ x : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y] \rrbracket)(\eta, a)) \\
= \llbracket c_2 \rrbracket(\eta, \lambda a . \llbracket c_1[x \mapsto y] \rrbracket(\eta, a, \tilde{f}_2))
\end{align*}$$

The term $c_1[x \mapsto y]$ only rename the variable to reduce confusion with variable bindings. Because $\llbracket \Gamma, y : A, f : A \rightarrow C \vdash y : A \rrbracket(\eta, a, \tilde{f}_2) = a$, we can use Lemma 6.5.2.

$$\llbracket c_1[x \mapsto y](\eta, a, \tilde{f}_2) \rrbracket = \llbracket c_1 \rrbracket(\eta, \llbracket y \rrbracket(\eta, a, \tilde{f}_2), \tilde{f}_2) = \llbracket c_1 \rrbracket(\eta, a, \tilde{f}_2)$$

All that is left to show is that $\tilde{f}_1$ is equal to $\lambda a . \llbracket c_1 \rrbracket(\eta, a, \tilde{f}_2)$. In both cases the definition of the recursive function is $c_1$, so $\tilde{f}_1$ and $\tilde{f}_2$ are both constructed as

$$\mu f . \lambda a . \llbracket c_1 \rrbracket(\eta, a, f)$$

It follows that $\tilde{f}_1 = \tilde{f}_2$. By the fixpoint property, it also follows that

$$\tilde{f}_1 = \lambda a . \llbracket c_1 \rrbracket(\eta, a, \tilde{f}_1) = \lambda a . \llbracket c_1 \rrbracket(\eta, a, \tilde{f}_2).$$

We now connect all the steps.

$$\begin{align*}
\llbracket \text{let rec } f \ x : A \rightarrow C = c_1 \text{ in } c_2 \rrbracket \eta \\
= \llbracket c_2 \rrbracket(\eta, \tilde{f}_1) \\
= \llbracket c_2 \rrbracket(\eta, \lambda a . \llbracket c_1 \rrbracket(\eta, a, \tilde{f}_1)) \\
= \llbracket c_2 \rrbracket(\eta, \lambda a . \llbracket c_1[x \mapsto y] \rrbracket(\eta, a, \tilde{f}_1)) \\
= \llbracket c_2 \rrbracket(\eta, \lambda a . \llbracket \text{let rec } f \ x : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y] \rrbracket(\eta, a, \tilde{f}_2)) \\
= \llbracket c_2[f \mapsto (\text{fun } y : A \rightarrow C = c_1 \text{ in } c_1[x \mapsto y])] \rrbracket \eta
\end{align*}$$
• **MatchNil**: The semantics of the match statement depends on the meaning of the argument. In this case $$\llbracket \lambda x \mapsto \epsilon \rrbracket = \epsilon$$, so the first branch is selected.

\[
\llbracket \text{match } \lambda \mapsto \epsilon \text{ with } \lambda x \mapsto x \rrbracket \mapsto c_1 | x :: xs \mapsto c_2 \eta = \llbracket c_1 \rrbracket \eta
\]

• **MatchCons**: In the case of a constructed list, the second branch is chosen. Here, we need to use Lemma 6.5.2 twice.

\[
\llbracket \text{match } (v :: vs) \text{ with } \lambda x \mapsto x \rrbracket \mapsto c_1 | x :: xs \mapsto c_2 \eta = \llbracket c_2 \rrbracket (\eta, \llbracket v \rrbracket \eta, \llbracket vs \rrbracket \eta) = \llbracket c_2 \rrbracket (\eta, \llbracket v \rrbracket \eta, \llbracket vs \rrbracket \eta) = \llbracket c_2 \rrbracket (\eta)
\]

• **DoStep**: If $$c \leadsto c'$$, then by induction $$\llbracket c \rrbracket \eta = \llbracket c' \rrbracket \eta$$.

\[
\llbracket \text{do } x \leftarrow c_1 \text{ in } c_2 \rrbracket \eta = \text{lift}_{\llbracket v \rrbracket \eta} (\lambda a . \llbracket c_2 \rrbracket (\eta, a)) (\llbracket c_1 \rrbracket \eta)
\]

• **DoRet**: Operations also carry type coercions. Assume $$\text{op} : A_{op} \rightarrow B_{op} \in \Sigma$$ and let $$A_{op} \rightarrow B_{op}$$ be the type annotation of the operation call. We denote $$\alpha = \llbracket A_{op} \leq A'_{op} \rrbracket$$ and $$\beta = \llbracket B'_{op} \leq B_{op} \rrbracket$$. In line 5 we use Lemma 6.5.1 for $$\llbracket c_2 \rrbracket (\eta, a) = \llbracket c_2 \rrbracket (\eta, \beta(b), a)$$.

\[
\llbracket \text{do } x \leftarrow \text{op}_{A_{op} \rightarrow B_{op}} (v; y . c_1) \text{ in } c_2 \rrbracket \eta
\]

• **HandleStep**: If $$c \leadsto c'$$, then by induction $$\llbracket c \rrbracket \eta = \llbracket c' \rrbracket \eta$$.

\[
\text{with } v \text{ handle } c \eta = (\llbracket v \rrbracket \eta)(\llbracket c \rrbracket \eta) = (\llbracket v \rrbracket \eta)(\llbracket c' \rrbracket \eta) = \text{with } v \text{ handle } c' \eta
\]

• **HandleRet**:}

\[
\text{with (handler (ret (x : A) \mapsto c_r ; h)) handle (ret v) \rrbracket \eta}
\]

\[
= \text{lift}_{\llbracket v \rrbracket \eta} (\lambda a . \llbracket c_r \rrbracket (\eta, a)) (\text{in}_{\text{val}_1} (\llbracket v \rrbracket \eta))
\]

\[
= \llbracket c_r \rrbracket (\eta, \llbracket v \rrbracket \eta)
\]

\[
= \llbracket c_r \rrbracket (\eta)
\]

100
• HANDLEOP: We shorten \((\text{handler} (\text{ret} (x : A) \mapsto c; h))\) to \(\mathcal{H}\). Assume that the operation case in \(\Gamma \vdash h : \Sigma \Rightarrow D\) for \(op\) is \(c_{op}\). We again need to deal with the type coercions of operation calls. Assume \(op : A'_{op} \rightarrow B'_{op} \in \Sigma\) and let \(A_{op} \rightarrow B_{op}\) be the type annotation of the operation call. We denote \(\alpha = [A_{op} \leq A'_{op}]\) and \(\beta = [B'_{op} \leq B_{op}].\)

\[
\begin{align*}
&\{\text{with } \mathcal{H} \text{ handle } (op_{A_{op}} \rightarrow B_{op} (v; y.c))\} \eta \\
&= (\text{lift}_{\mathcal{H}} \text{if} (\lambda a. \ [c_r (\eta, a)]) (\text{in}_{op} (\alpha (\eta, \eta) \alpha \beta)) (\lambda b. \ [c (\eta, \beta)])) \\
&= (\text{lift}_{\mathcal{H}} \eta) \alpha (\eta, \eta) \alpha \beta (\lambda b. \ [c (\eta, \beta)]) \\
&= \{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
&\{\text{with } \mathcal{H} \text{ handle } c\} \eta \\
\end{align*}
\]

We now simplify the right side of \(\sim\). By coherence of denotational semantics (Proposition 6.4.9), any typing derivation produces the same denotation. For \(v\) and \((\text{fun} (y : B'_{op}) \mapsto \ldots)\) we choose typing derivations that start with \(\text{TYPEVSUBSUME}\). This allows us to apply \(\alpha\) and \(\beta\). The coercion for \(D \leq D\) is the identity and can be dropped.

\[
\begin{align*}
&\{c_{op} (x \mapsto v, k \mapsto (\text{fun} (y : B_{op}) \mapsto \text{with } \mathcal{H} \text{ handle } c))\} \eta \\
&= \{c_{op} (\eta, \Gamma \vdash v : A'_{op}, \eta, \Gamma \vdash \text{fun} (y : B_{op}) \mapsto \text{with } \mathcal{H} \text{ handle } c : B'_{op} \rightarrow D (\eta))\} \eta \\
&= \{c_{op} (\eta, \alpha (\eta, \eta), \eta / D \leq D) \circ (\lambda b. \ [c (\eta, \beta)])\} \eta \\
&= \{c_{op} (\eta, \alpha (\eta, \eta), \eta / \text{with } \mathcal{H} \text{ handle } c (\eta, \beta))\} \eta \\
&\square
\]

101
Chapter 7

Effect theory semantics

Equations enrich the denotational semantics through partial equivalence relations (PERs). The equivalence relations are partial in order to focus solely on elements with suitable behaviour. We begin by providing denotational semantics of templates in Section 7.1. We then use template denotations to construct denotations of theories in Section 7.2, which are interpreted as PERs. This allows us to specify the requirements for a sound logic system in Section 7.3. When the type system is equipped with a sound logic, the denotations of terms are well formed with respect to the equivalence relation. The equational and propositional logics from Chapter 5 are both sound, establishing the validity of the proposed effect-theory system.

7.1 Semantics of templates

Template variables have types \( A \to *, \) which require * to be instantiated before use. We mirror that behaviour in denotational semantics by using functors. The denotations of template contexts are constructed in a way that \( \llbracket (z_i : A_i \to *)_i \rrbracket \llbracket C \rrbracket \) is the same as \( \llbracket (z_i : A_i \to C)_i \rrbracket \) if \( C \) is a constant functor, mapping to \( \llbracket \Gamma \rrbracket \) when convenient. Throughout this chapter we use \( Z_C \) for a context obtained from \( Z \) by replacing * with \( C \).

We consider template denotations with respect to an effect interpretation \( H \), similarly to how template instantiation \( T^H_\Lambda \) uses operation cases. In fact, we show that the notions are closely related (Lemma 7.2.5). Applying an interpretation \( H : \text{interp}_\Sigma(Y) \) to a well-typed template \( \Gamma; Z \vdash T : \Sigma \) is denoted as

\[
\llbracket \Gamma; Z \vdash T : \Sigma \rrbracket^H : (\llbracket \Gamma \rrbracket \times \llbracket Z \rrbracket)Y \to Y.
\]

The definition is recursive and closely mirrors the denotations of terms, except for
operations where we use the supplied interpretation \( H \).

\[
\frac{\{\Gamma ; Z \vdash z \colon \Sigma \}}{\|\Gamma ; Z \vdash z \colon \Sigma \|^{H}(\eta; \xi) = \xi(\|v\|\eta)}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{op}_{A \rightarrow B}(v; y.T) \colon \Sigma \|^{H}(\eta; \xi) = \ \\
H_{\text{op}}(\|A \leq A'\|\|v\|\eta), \lambda b \in \|B\| \cdot \|T\|^{H}(\eta, \|B' \leq B\|b; \xi))}{(\text{op} : A' \rightarrow B' \in \Sigma)}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{do pure } x \leftarrow c \text{ in } T ; \Sigma \|^{H}(\eta; \xi) = \ \\
\left\{ \begin{array}{ll}
\{\Gamma, x : A ; Z \vdash T ; \Sigma \|^{H}(\eta, a; \xi) ; \|c\|\eta = \text{in}_{\text{va}}a \\
\bot & ; \|c\|\eta = \bot
\end{array} \right.}{(\text{do pure})}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{absurd } v \colon \Sigma \|^{H}(\eta; \xi) = \text{emptyfun}(\|v\|\eta)\}}{(\text{absurd})}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{match } v \text{ with } (x, y) \mapsto T ; \Sigma \|^{H}(\eta; \xi) = \ \\
\{\Gamma, x : A, y : B ; Z \vdash T ; \Sigma(\eta, a, b; \xi)\|^{H}\}}{(\text{match})}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{match } v \text{ with Left } x \mapsto T_1 \mid \text{Right } y \mapsto T_2 ; \Sigma \|^{H}(\eta; \xi) = \ \\
\left\{ \begin{array}{ll}
\{\Gamma, x : A ; Z \vdash T_1 ; \Sigma \|^{H}(\eta, a; \xi) ; \|v\|\eta = \iota_1a \\
\{\Gamma, y : B ; Z \vdash T_2 ; \Sigma \|^{H}(\eta, b; \xi) ; \|v\|\eta = \iota_2b
\end{array}\right.}{(\text{match})}
\]

\[
\frac{\{\Gamma ; Z \vdash \text{match } v \text{ with } [ ] \mapsto T_1 \mid x :: xs \mapsto T_2 ; \Sigma \|^{H}(\eta; \xi) = \ \\
\left\{ \begin{array}{ll}
\{\Gamma ; Z \vdash T_1 ; \Sigma \|^{H}(\eta; \xi) ; \|v\|\eta = \epsilon \\
\{\Gamma, x : A, xs : A \text{ list} ; Z \vdash T_2 ; \Sigma \|^{H}(\eta, a_0, (a_i)_{i=1}^{n}; \xi) ; \|v\|\eta = a_0, \ldots, a_n
\end{array}\right.}{(\text{match})}
\]

As opposed to the semantics of \text{do}, we do not use lift for \text{do pure}, because it is not possible to specify the free interpretation for \( Y \) at this point. As before, we shorten \( \{\Gamma ; Z \vdash T ; \Sigma \|^{H} \) to \( \{\|T\|^{H} \) when \( \Gamma, Z, \) and \( \Sigma \) are easily inferred.

To recover the functionality of template instantiation, we use \( \{\|T\|^{FX.\Sigma} \). The free interpretation maps operation nodes to equal operation nodes, and we end up with the denotation of the computation that the template represents. Next we show that the definition of \( \{\|T\|^{H} \)
really mimics the behaviour of applying the interpretation \( H \).

**Lemma 7.1.1.** Take any predomain \( X \), domain \( Y \), interpretation \( H : \text{interp}_\Sigma(Y) \), and continuous function \( f : X \rightarrow Y \). Then the following diagram commutes

\[
\begin{array}{c}
(\|\Gamma\| \times \|\Sigma\|)X \leftarrow (\|\Gamma\| \times \|\Sigma\|)Y \\
\downarrow \text{lift}_{\Sigma}f \rightarrow \downarrow \text{lift}_{\Sigma}f
\end{array}
\]

\[
\begin{array}{c}
(\|\Gamma\| \times \|\Sigma\|)(\text{lift}_{\Sigma}f)(\|\eta\|; (\xi_j)) = ((\eta_i); (\text{lift}_{\Sigma}f \circ \xi_j)).
\end{array}
\]

**Proof.** The lift mapped by the functor \( \|\Gamma\| \times \|\Sigma\| \) only affects elements used for template variables.

Because it has no effect on \( \eta \), we use the notation \( (\eta_i)(\text{lift}_{\Sigma}f \circ \xi_j) \) for the transformed environment. To clarify the notation, the precedence is \( \text{lift}_{\Sigma}f \circ \xi_j = (\text{lift}_{\Sigma}f) \circ \xi_j \). The goal of
the proof is to show that for any \((\eta; \zeta) \in [\Gamma] \times [Z](\Sigma) X\) it holds that

\[
(lift_H f)([T]^F_{x,z}(\eta; \zeta)) = [T]^H(\eta; lift_H f \circ \zeta).
\]
The proof proceeds by induction on the derivation \(\Gamma; Z \vdash T : \Sigma\).

- **WfTApp:**

\[
(lift_H f)([\lambda \mathbf{v} \mathbf{v}]^F_{x,z}(\eta; \zeta)) = (lift_H f)([\lambda \mathbf{v} \mathbf{v}]) = [\lambda \mathbf{v} \mathbf{v}]^H(\eta; lift_H f \circ \zeta)
\]

- **WfTOP:** Assume \(op : A' \rightarrow B' \in \Sigma\) and denote \(\alpha = [A \leq A']\) and \(\beta = [B' \leq B]\).

  We start by simplifying both sides.

\[
\begin{align*}
(lift_H f)(\mathbf{op}_{A \rightarrow B}(v; y.T))^F_{x,z}(\eta; \zeta)) & = (lift_H f)((F_X_{1,1})_{op}(\alpha \mathbf{v} \mathbf{v} \eta), \lambda b . [T]^F_{x,z}(\eta, \beta(b); \zeta)) \\
& = (lift_H f)(\mathbf{in}_{op}(\alpha \mathbf{v} \mathbf{v} \eta), \lambda b . [T]^F_{x,z}(\eta, \beta(b); \zeta)) \\
& = H_{op}(\alpha \mathbf{v} \mathbf{v} \eta), \lambda b . (lift_H f)([T]^F_{x,z}(\eta, \beta(b); \zeta))
\end{align*}
\]

\[
\begin{align*}
& \mathbf{op}_{A \rightarrow B}(v; y.T)^H(\eta; lift_H f \circ \zeta) \\
& = H_{op}(\alpha \mathbf{v} \mathbf{v} \eta), \lambda b . [T]^H(\eta, \beta(b); lift_H f \circ \zeta)
\end{align*}
\]

The induction hypothesis states

\[
(lift_H f)([T]^F_{x,z}(\eta, \beta(b); \zeta)) = [T]^H(\eta, \beta(b); lift_H f \circ \zeta),
\]

which proves that the sides are equal.

- **WfTDO:** By the restrictions of \texttt{do} \texttt{pure}, we have \(\Gamma \vdash \mathbf{c} : A'!\{\}\{\}\{\}. It follows that 
\([\mathbf{c}]_{\eta}\) is either \(\perp\) or \(\text{inva}_1(a)\) for some \(a \in [A]\), so we do a case analysis.

  - If \([\mathbf{c}]_{\eta} = \perp\), then we have \((lift_H f)(\perp)\) on the left and \(\perp\) on the right. Since lifts are strict functions, the two are equal.
  - If \([\mathbf{c}]_{\eta} = \text{inva}_1(a)\), then we have \((lift_H f)([T]^F_{x,z}(\eta, a; \zeta))\) on the left side and \([T]^H(\eta, a; lift_H f \circ \zeta)\) on the right side. These two expressions are equal by induction.

- **WfTABSURD:** After simplification we arrive at

\[
(lift_H f)(\text{emptyfun}_{\Sigma' X}([\mathbf{v}]_{\eta})) = \text{emptyfun}_{\Sigma' Y}([\mathbf{v}]_{\eta}).
\]

Empty functions are determined by their codomain, and since \(lift_H f \circ \text{emptyfun}_{\Sigma' X}\) maps from \(\emptyset\) to \([\Sigma']_Y\), it is equal to \(\text{emptyfun}_{\Sigma' Y}\).

- **WfTPRODMatch, WfTSumMatch, WfTListMatch:** The proof for all three cases follows the same pattern, so we only do it for WfTListMatch. The induction hypotheses state that for any \(a \in [A]\) and any \(as \in [A \text{ list}]\), we have

\[
\begin{align*}
(lift_H f)([T_1]^F_{x,z}(\eta; \zeta)) & = [T_1]^H(\eta; lift_H f \circ \zeta) \\
(lift_H f)([T_2]^F_{x,z}(\eta, a, as; \zeta)) & = [T_2]^H(\eta, a, as; lift_H f \circ \zeta).
\end{align*}
\]

105
The remainder of the proof is straightforward:

\[(\text{lift}_H f)(\mathbb{match} \; v \; \text{with} \; [ \; \rightarrow \; T_1 \; ] \; \rightarrow \; x :: xs \rightarrow \; T_2)^{\mathbb{F}_x,y}(\eta, \zeta))\]

\[
\begin{align*}
&= \left(\left(\text{lift}_H f\right)(T_1)^{\mathbb{F}_x,y}(\eta, \zeta)\right) \quad ; \; \eta = \epsilon, \\
&\quad \left(\left(\text{lift}_H f\right)(T_2)^{\mathbb{F}_x,y}(\eta, a_0, (a_i)_{i=1}^n \circ \zeta)\right) \quad ; \; \eta = a_0, \ldots, a_n \\
&= \left[\left[ T_1 \right]^H(\eta; \text{lift}_H f \circ \zeta) \quad ; \; \eta = \epsilon, \\
&\quad \left[\left[ T_2 \right]^H(\eta, a_0, (a_i)_{i=1}^n; \text{lift}_H f \circ \zeta)\right] \quad ; \; \eta = a_0, \ldots, a_n \\
&= \mathbb{match} \; v \; \text{with} \; [ \; \rightarrow \; T_1 \; ] \; \rightarrow \; x :: xs \rightarrow \; T_2]^H(\eta; \text{lift}_H f \circ \zeta)
\end{align*}
\]

\[\square\]

### 7.2 Semantics of theories

The equations are interpreted as partial equivalence relations that relate only those elements we consider well-formed; for instance, functions need to map related elements to related results. The PER must also relate elements that are considered equal according to equations of a theory. This way, handlers that are not correct (and therefore do not map related elements to related results) are ignored by the relation. We construct a PER for each type, built recursively on the type structure.

For value types, the partial equivalence relations on \([A]\) are defined by the the following:

- \(\sim_{\text{empty}}\) is the empty relation.
- \(\sim_{\text{unit}}\) and \(\sim_{\text{int}}\) are identity relations.
- \((a, b) \sim_{A \times B} (a', b') \iff a \sim_A a' \land b \sim_B b'\)
- \(t_1(a) \sim_{A+B} t_1(a') \iff a \sim_A a'\) and \(t_2(b) \sim_{A+B} t_2(b') \iff b \sim_B b'\) and elements from different components are never related.
- \((a_i)_{i=0}^n \sim_{A \text{ list}} (a_i')_{i=0}^n \iff \forall i = 0, \ldots, n. a_i \sim_A a_i'\) and lists of different lengths are never related.
- \(f \sim_{A \rightarrow C} f' \iff (\forall a, a' \in [A]. a \sim_A a' \implies f(a) \sim_C f'(a'))\)
- \(h \sim_{C \rightarrow D} h' \iff (\forall c, c' \in [C]. c \sim_C c' \implies h(c) \sim_D h'(c'))\)

For \(\Gamma = x_1 : A_1, \ldots, x_n : A_n\), we define the relation \(\sim_\Gamma\) on \([\Gamma]\) by

\[(a_1, \ldots, a_n) \sim_\Gamma (a_1', \ldots, a_n') \iff a_1 \sim_{A_1} a_1' \land \ldots \land a_n \sim_{A_n} a_n'.\]

A relation for operation cases relates elements of \(\text{interp}_{\Sigma}(\llbracket D \rrbracket)\). For \(H, H' : \text{interp}_{\Sigma}(\llbracket D \rrbracket)\) we define

\[H \sim_{\Sigma \rightarrow D} H' \iff \left(\forall \text{op} : A \rightarrow B \in \Sigma. \forall a, a' \in [A]. \forall \kappa, \kappa' \in [B \rightarrow D]. a \sim_A a' \land \kappa \sim_B \kappa' \implies H_{\text{op}}(a, \kappa) \sim_D H'_{\text{op}}(a', \kappa')\right)\]

For computation types, we need to take into account the effect-theory equations. We also ensure that the PER is symmetric, transitive, and closed under chains. For the computation type \(C = A\Sigma/E\), we define the relation \(\sim_C\) on the domain \([A\Sigma/E]\) to be the smallest relation, closed under the following rules:
1. We have $\bot \sim_\mathcal{C} \bot$.

2. If $a \sim_A a'$, then $\text{in}_{\text{val}_1}(a) \sim_{A\Sigma/E} \text{in}_{\text{val}_1}(a')$.

3. For every operation $\text{op}: A_{\text{op}} \to B_{\text{op}} \in \Sigma$, if $a \sim_{A_{\text{op}}} a'$, and if $b \sim_{B_{\text{op}}} b'$ implies $\kappa(b) \sim \kappa(b')$, then $\text{in}_{\text{op}(a, \kappa)} \sim_\mathcal{C} \text{in}_{\text{op}(a', \kappa')}$.

4. Let $\Gamma: Z \vdash T_1 \sim T_2$ be an equation in $\mathcal{E}$, where $\Gamma = (x_i : A_i)_{i}$ and $Z = (c_j : B_j \to *)_j$. If we have $a_i \sim_{A_i} a'_i$ for all $i$, and if $b \sim_{B_j} b'$ implies $f_j(b) \sim \kappa f_j'(b')$ for all $j$, then

$$\llbracket \Gamma; Z \vdash T_1 : \Sigma \rrbracket^{\mathcal{F}_{\text{A}_{\text{L}}}}((a_i)_i, (f_j)_j) \sim_\mathcal{C} \llbracket \Gamma; Z \vdash T_2 : \Sigma \rrbracket^{\mathcal{F}_{\text{A}_{\text{L}}}}((a'_i)_i, (f'_j)_j)$$

5. If $c_1 \sim_\mathcal{C} c_2$, then $c_2 \sim_\mathcal{C} c_1$.

6. If $c_1 \sim_\mathcal{C} c_2$ and $c_2 \sim_\mathcal{C} c_3$, then $c_1 \sim_\mathcal{C} c_3$.

7. For chains $(c_i)_i$ and $(c'_i)_i$, if $\forall i. c_i \sim_\mathcal{C} c'_i$, then $\bigvee_i c_i \sim_\mathcal{C} \bigvee_i c'_i$.

**Lemma 7.2.1.** There exists a smallest relation, closed under rules for $\sim_\mathcal{C}$.

*Proof.* There always exists at least one such relation—the full relation—which relates any pair of elements. We now show that there exists a smallest one. Assume we have a family of relations $\{\mathcal{R}_i\}_{i \in I}$. closed under the rules. We now show that $\mathcal{R}_\cap = \cap_{i \in I} \mathcal{R}_i$ is also a relation closed under the same rules.

1. We have $\bot \mathcal{R}_i \bot$ for all $i \in I$, so $\bot \mathcal{R}_\cap \bot$.

2. From $a \sim_A a'$ it follows that $\text{in}_{\text{val}_1}(a) \mathcal{R}_i \text{in}_{\text{val}_1}(a')$ for all $i \in I$, which means that $\text{in}_{\text{val}_1}(a) \mathcal{R}_\cap \text{in}_{\text{val}_1}(a')$.

3. Assume $a \sim_{A_{\text{op}}} a'$ and that $b \sim_{B_{\text{op}}} b'$ implies $\kappa(b) \mathcal{R}_\cap \kappa(b')$. By construction of $\mathcal{R}_\cap$, it follows that $b \sim_{B_{\text{op}}} b'$ implies $\forall i \in I. \kappa(b) \mathcal{R}_i \kappa(b')$. Since all $\mathcal{R}_i$ are closed under rule 3, we have $\forall i \in I. (\text{in}_{\text{op}(a, \kappa)}) \mathcal{R}_i (\text{in}_{\text{op}(a', \kappa')})$, and in turn $(\text{in}_{\text{op}(a, \kappa)}) \mathcal{R}_\cap (\text{in}_{\text{op}(a', \kappa')})$.

4. Let $(x_i : A_i) : (c_j : B_j \to *)_j \vdash T_1 \sim T_2$ be an equation in $\mathcal{E}$ and assume $a_k \sim_{A_k} a'_k$ and that $b \sim_{B_j} b'$ implies $f_j(b) \mathcal{R}_j f_j'(b')$. For $b \sim_{B_j} b'$ it follows that $\forall i \in I. f_j(b) \mathcal{R}_i f_j'(b')$ by construction of $\mathcal{R}_\cap$, so for all $i \in I$, we have

$$(\llbracket T_1 \rrbracket^{\mathcal{F}_{\text{A}_{\text{L}}}}((a_k)_k; (f_j)_j)) \mathcal{R}_i (\llbracket T_2 \rrbracket^{\mathcal{F}_{\text{A}_{\text{L}}}}((a'_k)_k; (f'_j)_j)).$$

By definition, the above also holds for $\mathcal{R}_\cap$.

5. From $c_1 \mathcal{R}_\cap c_2$ it follows that $\forall i \in I. c_1 \mathcal{R}_i c_2$, and by symmetry of $\mathcal{R}_i$ it follows that $\forall i \in I. c_2 \mathcal{R}_i c_1$. This in turn means that $c_2 \mathcal{R}_\cap c_1$.

6. Similar to 5.

7. Assume that for chains $(c_j)_j$ and $(c'_j)_j$, we have $c_j \mathcal{R}_\cap c'_j$ for all $j$. Then we also have $c_i \mathcal{R}_i c_i'$ for all $i$ and all $j$. Relations $\mathcal{R}_i$ are chain complete, so it holds that $\forall i \in I. \bigvee_i c_i \mathcal{R}_i \bigvee_i c'_i$, and therefore $\bigvee_i c_j \mathcal{R}_\cap \bigvee_i c'_j$.

We obtain $\sim_\mathcal{C}$ as the intersection of all relations that are closed under the rules 1-7. □
Lifts play an important role in the denotational semantics of EEFF, and we need to determine the interaction between lift and ~. If we assume two related functions and two related interpretations, it would be sensible to expect that the respective lifts are related as well. It turns out that an additional requirement must be placed on interpretations in order to ensure correct behaviour with regard to equations.

**Lemma 7.2.2.** Let $C = A!\Sigma/E$, $g \sim_A \circ \neg \circ \neg \circ B$, and $H \sim_{\Sigma=E} H^\prime$. Assume that for any $((x_i : A_i), (z_j : B_j \rightarrow *), T_1 \sim T_2) \in E$ and any $(\eta, \xi) \sim_{(x_i : A_i), (z_j : B_j \rightarrow \Sigma)} (\eta', \xi')$ we have

$$
[T_1]^H(\eta, \xi) \sim_{\Sigma=E} [T_2]^H(\eta', \xi').
$$

Then for every $c \sim C c'$ it holds that

$$(\text{lift}_H g)(c) \sim_{\Sigma=E} (\text{lift}_H g')(c').$$

**Proof.** The proof takes into account the structure of the relation $\sim_C$, which is defined as the smallest relation, closed under a set of rules. We define a relation $\mathcal{R} \subseteq \parallel C \parallel \times \parallel C \parallel$ by

$$c_1 \mathcal{R} c_2 \Leftrightarrow (\text{lift}_H g)(c_1) \sim_{\Sigma=E} (\text{lift}_H g')(c_2)$$

and show that it is closed under the rules 1-7 of the definition of $\sim_C$.

1. Since lift is strict and $\bot \sim_{\Sigma=E} \bot$, we have $\bot \mathcal{R} \bot$.

2. Assume $a \sim_A a'$. We simplify $(\text{lift}_H \circ \text{val}_\Sigma)(a) = g(a)$ and $(\text{lift}_H \circ g')(\text{val}_\Sigma)(a') = g'(a')$. From $g \sim_A \circ \neg \circ \neg \circ B$ it follows that $g(a) \sim_{\Sigma=E} g'(a')$. Therefore, $(\text{val}_\Sigma)(a) \mathcal{R} (\text{val}_\Sigma)(a')$.

3. Let $\circ \circ = A \circ \circ B \in \Sigma$. Assume $a \sim_A a'$ and that $b \sim_{B \circ \circ} b'$ implies $\kappa(b) \mathcal{R} \kappa'(b')$. By definition of $\mathcal{R}$ it follows that

$$b \sim b' \Rightarrow (\text{lift}_H g)(\kappa(b)) \sim_{\Sigma=E} (\text{lift}_H g')(\kappa'(b')),$$

which can be stated as $(\text{lift}_H g) \circ \kappa \sim_{B \circ \circ \rightarrow \Sigma} (\text{lift}_H g') \circ \kappa'$. Because $H \sim_{\Sigma=E} H'$, we have

$$H_{\circ \circ}(a, \text{lift}_H g \circ \kappa) \sim_{\Sigma=E} H'_{\circ \circ}(a', \text{lift}_H g' \circ \kappa').$$

This results in $(\text{in}_{\circ \circ}(a, \kappa)) \mathcal{R} (\text{in}_{\circ \circ}(a', \kappa'))$.

4. Let $((x_i : A_i), (z_j : B_j \rightarrow *), T_1 \sim T_2)$ be an equation in $E$. Assume $a_i \sim_{A_i} a'_i$ for all $i$, and that $b \sim_{B_j} b'$ implies $f_j(b) \mathcal{R} f'_j(b')$ for all $j$. By Lemma 7.1.1 it holds that

$$(\text{lift}_H g)(\parallel T_k \parallel)^{1 \rightarrow 1}(a_i ; (f_j)_{j}) = \parallel T_k \parallel^H(a_i ; (\text{lift}_H g \circ f_j)_{j}) \quad (k = 1, 2).$$

By definition of $\mathcal{R}$ we have $\text{lift}_H g \circ f_j \sim_{B_j \circ \circ} \circ \circ \circ \circ B \circ \circ g' \circ f'_j$ for all $j$, so it follows by assumption of the Lemma that

$$[T_1]^H(a_i ; (\text{lift}_H g \circ f_j)) \sim_{\Sigma=E} [T_2]^H(a_i ; (\text{lift}_H g' \circ f'_j)).$$

We conclude that $(\parallel T_1 \parallel^{1 \rightarrow 1}(a_i ; (f_j))) \mathcal{R} (\parallel T_2 \parallel^{1 \rightarrow 1}(a_i ; (f'_j))).$

5. Follows from symmetry of $\sim_{\Sigma=E}$.

6. Follows from transitivity of $\sim_{\Sigma=E}$. 

108
7. Let \((c_i)_i\) and \((c'_i)_i\) be chains with \(c_i \leq c'_i\). This means that for every \(i\), we have 
\((\text{lift}_g \circ \text{lift}_h)(c_i) \sim \text{D} (\text{lift}_h \circ \text{lift}_g)(c'_i)\), and because lifts are continuous, we obtain two chains in 
\([\text{D}]\). The relation \(\sim \text{D}\) is chain complete so \(\bigvee_i (\text{lift}_h \circ \text{lift}_g)(c_i) \sim \text{D} \bigvee_i (\text{lift}_h \circ \text{lift}_g)(c'_i)\). Suprema distribute over continuous functions, and thus \((\text{lift}_h \circ \text{lift}_g)(\bigvee_i c_i) \sim \text{D} (\text{lift}_h \circ \text{lift}_g)(\bigvee_i c'_i)\), which means \((\bigvee_i c_i) \leq c'\).

By definition \(c\) is the smallest relation closed under the above rules, so \(c \leq c'\) implies \(c \leq c'\). This means that for any \(c \leq c'\), we have \((\text{lift}_h \circ \text{lift}_g)(c) \sim \text{D} (\text{lift}_h \circ \text{lift}_g)(c')\).

**Corollary 7.2.3.** If \(g \sim_A \Sigma/E\) \(g'\) then for every \(c \sim_A \Sigma/E\) \(c'\) it holds that 
\((\text{lift}_f \circ \text{lift}_g)(c) \sim \Sigma/E (\text{lift}_f \circ \text{lift}_g')(c')\).

**Proof.** Checking that \(\text{lift}_f \circ \text{lift}_g \sim_B \Sigma/E \text{ lift}_f \circ \text{lift}_g\) is straightforward. The requirement 
\(\lceil T_1 \rceil \circ \text{lift}_f \sim_B \Sigma/E \lceil T_2 \rceil \circ \text{lift}_f\) for all equations in \(E\) (and suitable \(\eta\) and \(\zeta\)), follows directly from rule 4 of \(\sim_B \Sigma/E\). □

**Lemma 7.2.4.**

\(\circ\) If \(a \sim_A a'\), then also \(\lceil A = A' \rceil a \sim_A \lceil A = A' \rceil a'\).

\(\circ\) If \(c \leq c'\), then also \(\lceil C \leq C' \rceil c \sim \lceil C \leq C' \rceil c'\).

**Proof.** The proof proceeds by induction on the derivation of \(A \leq A'\) and \(C \leq C'\).

- The proofs for base types, sums, products, and lists are straightforward.

- Assume \(f \sim_A \Sigma f'\) and that we have \(A \leq C \leq A' \rightarrow C'\). If \(a \sim_A a'\), then by induction \(\lceil A' \leq A \rceil a \sim_A \lceil A' \leq A \rceil a'\), and from the definition of \(\sim_A \Sigma\) it follows that 
\(f(\lceil A' \leq A \rceil a) \sim_C \lceil A' \leq A \rceil a'\). The induction hypothesis for \(C \leq C'\) then gives us 
\(C \leq C' \lceil f(\lceil A' \leq A \rceil a) \sim_C C' \lceil f(\lceil A' \leq A \rceil a')\).

This holds for arbitrary \(a \sim_A a'\), so it follows that 
\(A \rightarrow C \leq A' \rightarrow C' \lceil f \sim_A \Sigma f' \lceil A \rightarrow C \leq A' \rightarrow C' \rceil f'\).

The same proof holds for handler types.

- The case for \(A \Sigma/E \leq A' \Sigma'/E'\) is a bit trickier. By construction of \(\Sigma \leq \Sigma'\), it holds that every \(op : A_{op} \rightarrow B_{op} \in \Sigma\) has the type \(op : A'_{op} \rightarrow B'_{op}\) in \(\Sigma'\), where \(A_{op} \leq A'_{op}\) and \(B_{op} \leq B'_{op}\). Recall the definitions: 
\(\lceil A \Sigma/E \leq A' \Sigma'/E' \rceil = \text{lift}_E \Sigma \leq \Sigma' \Sigma F (\lambda a. \text{in}_{\text{eval}}(\lceil A \leq A' \rceil a))\)
\(\lceil \Sigma \leq \Sigma' \rceil \circ \text{op}(x, k) = \text{in}_{\text{op}}(\lceil A_{op} \leq A'_{op} \rceil x, k \circ (\lceil B_{op} \leq B'_{op} \rceil))\)

To utilize Lemma 7.2.2 for lift, we need to satisfy three requirements.

- The functions being lifted need to be related. If \(a \sim_A a'\), then by induction 
\(\lceil A \leq A' \rceil a \sim_A \lceil A \leq A' \rceil a'\), and it follows that 
\(\lambda a. \text{in}_{\text{eval}}(\lceil A \leq A' \rceil a) \sim_A \lceil A \leq A' \rceil \lambda a. \text{in}_{\text{eval}}(\lceil A \leq A' \rceil a)\).
Using Lemma 7.2.2, it follows that for \( a \sim_{A_{op}} a' \) and \( \kappa \sim_{B_{op} \rightarrow A'\Sigma'/E'} \kappa' \) then

\[
\begin{align*}
\text{in}_{op}(\|A_{op} \leq A'_\kappa \|a, \kappa \circ \|B'_{op} \leq B_{op}\|) \\
\sim_{A'\Sigma'/E'} \text{in}_{op}(\|A_{op} \leq A'_\kappa \|a', \kappa' \circ \|B'_{op} \leq B_{op}\|).
\end{align*}
\]

This satisfies the conditions of \( \sim_{\Sigma \rightarrow A'\Sigma'/E'} \), and it follows that

\[
\|\Sigma \leq \Sigma'\|\|A'\| \sim_{\Sigma \rightarrow A'\Sigma'/E'} \|\Sigma \leq \Sigma'\|\|A'\|.
\]

The remaining requirement is that

\[
\|\Gamma ; Z \vdash T_1 : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta) \sim_{A'\Sigma'/E'} \|\Gamma ; Z \vdash T_2 : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta', \zeta')
\]

for every equation \( T_1 \sim T_2 \) in \( \mathcal{E} \). To that end we first show

\[
\|\Gamma ; Z \vdash T : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta) = \|\Gamma ; Z \vdash T : \Sigma' \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta).
\]

This is done by induction on the derivation of \( (\Gamma ; Z \vdash T : \Sigma) \). The only non-trivial case is \( \text{WFTOP} \). To avoid confusion, assume that \( A_1 \rightarrow B_1 \) is the type annotation of the operation \( op \), which has the type \( A_2 \rightarrow B_2 \) in \( \Sigma \) and the type \( A_3 \rightarrow B_3 \) in \( \Sigma' \). In line 4 we use 6.2.1 to combine subtype denotations, and line 5 follows by induction.

\[
\begin{align*}
\|\Gamma ; Z \vdash op_{A_1 \rightarrow B_1}(v; y.T) : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta) \\
&= ([\Sigma \leq \Sigma']^{\|A'\|})_{op}(\|A_1 \leq A_2\|([v], \eta), \lambda b. \|T\|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \|B_2 \leq B_1\|b; \zeta)) \\
&= \text{in}_{op}(\|A_2 \leq A_3\|([A_1 \leq A_2\|([v], \eta), \lambda b. \|T\|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \|B_2 \leq B_1\|b; \zeta)) \\
&= \text{in}_{op}(\|A_1 \leq A_3\|([v], \eta), \lambda b. \|T\|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \|B_2 \leq B_1\|b; \zeta)) \\
&= \|\Gamma ; Z \vdash op_{A_1 \rightarrow B_1}(v; y.T) : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta)
\end{align*}
\]

If \( T_1 \sim T_2 \) is in \( \mathcal{E} \) then by definition of \( \Sigma \leq \Sigma' \) it is also present in \( \mathcal{E}' \). Therefore, by rule 4 in the definition of \( \sim_{A'\Sigma'/E'} \), it must hold that

\[
\|T_1\|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta) \sim_{A'\Sigma'/E'} \|T_2\|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta', \zeta').
\]

and we have shown that this is equal to

\[
\|\Gamma ; Z \vdash T_1 : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta, \zeta) \sim_{A'\Sigma'/E'} \|\Gamma ; Z \vdash T_2 : \Sigma \|^{A \Sigma \subseteq \Sigma' \|A'\|}(\eta', \zeta').
\]

Using Lemma 7.2.2, it follows that for \( c \sim_{A'\Sigma/E} c' \) we have

\[
\|A'\Sigma/E \leq A'\Sigma'/E'\|c = ([\text{lift}_{A \Sigma \subseteq \Sigma' \|A'\|}(\lambda a. \text{inv}_{val}(\|A \leq A'\|a)))(c) \\
\sim_{A'\Sigma/E} ([\text{lift}_{A \Sigma \subseteq \Sigma' \|A'\|}(\lambda a. \text{inv}_{val}(\|A \leq A'\|a)))(c') = \|A'\Sigma/E \leq A'\Sigma'/E'\|c'.
\]

\( \square \)
Lemma 7.2.5. Assume well-typed operation cases \( \Gamma_1 \vdash h : \Sigma \Rightarrow D \) and a well-typed template \( \Gamma_2 : Z \vdash T : \Sigma \). Let \( \eta_1 \in \llbracket \Gamma_1 \rrbracket \), \( \eta_2 \in \llbracket \Gamma_2 \rrbracket \) and \( \zeta \in \llbracket Z \rfloor D \rrbracket \).

\[
\llbracket \Gamma_1,\Gamma_2,Z_D \vdash T_D^h : D \rrbracket (\eta_1,\eta_2,\zeta) = \llbracket \Gamma_2 : Z \vdash T : \Sigma \rrbracket \downarrow h \downarrow (\eta_2,\zeta)
\]

Proof. Under the given assumptions, Lemma 5.4.4 guarantees that the type derivation \( \Gamma_1,\Gamma_2,Z_D \vdash T_D^h : D \) holds. In the following proof we run into the issue of context reordering. We can keep variable names unique through \( a \) renaming, so reordering the context has no effect. Safety of context reordering can be shown by a straightforward induction, similar to proof of Lemma 6.5.1.

The proof proceeds by induction on derivation of \( \Gamma_2 : Z \vdash T : \Sigma \). The pattern for match templates is shown in the proof for \( \text{WFTSUMMATCH} \), and proofs for other match statements are omitted.

- \( \text{WFTSUMMATCH} \): We begin by simplifying both sides.

\[
\llbracket (\text{match} \; v \; \text{with} \; \text{Left} \; x \mapsto T_1 \; | \; \text{Right} \; y \mapsto T_2)^h_D \rrbracket (\eta_1,\eta_2,\zeta)
= \llbracket (\text{match} \; v \; \text{with} \; \text{Left} \; x \mapsto T_1^h_D \; | \; \text{Right} \; y \mapsto T_2^h_D) \rrbracket (\eta_1,\eta_2,\zeta)
= \begin{cases} 
\llbracket T_1^h_D \rrbracket (\eta_1,\eta_2,\zeta,a) & ; \llbracket v \rrbracket (\eta_1,\eta_2,\zeta) = \iota_1 a \\
\llbracket T_2^h_D \rrbracket (\eta_1,\eta_2,\zeta,b) & ; \llbracket v \rrbracket (\eta_1,\eta_2,\zeta) = \iota_2 b 
\end{cases}
\]

By Lemma 6.5.1 we know \( \llbracket v \rrbracket \eta_2 = \llbracket v \rrbracket (\eta_1,\eta_2,\zeta) \) by weakening the context. We also reorder contexts so that \((\eta_1,\eta_2,\zeta,a)\) becomes \((\eta_1,\eta_2,\zeta,\zeta)\); the rest follows by induction.

- \( \text{WFTABSURD} \): We use Lemma 6.5.1 on \( \Gamma_2 \vdash v : \text{empty} \) to get \( \llbracket v \rrbracket (\eta_1,\eta_2,\zeta) = \llbracket v \rrbracket (\eta_2) \).

\[
\llbracket \text{absurd} \; v \rrbracket (\eta_1,\eta_2,\zeta) = \text{emptyfun}_{\downarrow D} (\llbracket v \rrbracket (\eta_1,\eta_2,\zeta)) \\
= \text{emptyfun}_{\downarrow D} (\llbracket v \rrbracket (\eta_2)) \\
= \llbracket \text{absurd} \; v \rrbracket \downarrow h \downarrow (\eta_2,\zeta)
\]

- \( \text{WFTDO} \): This case is interesting because the denotations are not directly mirrored. By Lemma 6.5.1 we have \( \llbracket c \rrbracket (\eta_1,\eta_2,\zeta) = \llbracket c \rrbracket (\eta_2) \), where \( c \) is the pure computation. We now proceed by case analysis of \( \llbracket c \rrbracket \), which is either \( \bot \) or a value node due to purity.

\( \cdot \) If \( \llbracket c \rrbracket (\eta_1,\eta_2,\zeta) = \llbracket c \rrbracket (\eta_2) = \bot \), then the right side is \( \bot \) by definition. We simplify the left side to see that both sides match.

\[
\llbracket (\text{do pure} \; x \leftarrow c \; \text{in} \; T_D^h) \rrbracket (\eta_1,\eta_2,\zeta)
= \llbracket \text{do} \; x \leftarrow c \; \text{in} \; T_D^h \rrbracket (\eta_1,\eta_2,\zeta)
= (\text{lift}_{F_X}(\Lambda a \cdot \llbracket T_D^h \rrbracket (\eta_1,\eta_2,\zeta,a))) (\llbracket c \rrbracket (\eta_1,\eta_2,\zeta)) \\
= (\text{lift}_{F_X}(\Lambda a \cdot \llbracket T_D^h \rrbracket (\eta_1,\eta_2,\zeta,a))) (\bot) \\
= \bot
\]
\( \textbf{WfTOp} \): If \( \llbracket c \rrbracket (\eta_1, \eta_2, \zeta) = \llbracket c \rrbracket (\eta_2) = \text{in}_{va2}(a) \), we also use the IH that (after context reordering) states \( \llbracket T^h_D \rrbracket (\eta_1, \eta_2, \zeta, a) = \llbracket T \rrbracket^{h|n_1}(\eta_2, a; \zeta) \).

\[
\begin{align*}
\llbracket \text{do pure } x \leftarrow c \text{ in } T^h_D \rrbracket (\eta_1, \eta_2, \zeta) &= (\text{lift}_{F_2, D}(\lambda a \cdot \llbracket T^h_D \rrbracket (\eta_1, \eta_2, \zeta, a)))(\llbracket c \rrbracket (\eta_1, \eta_2, \zeta)) \\
&= (\text{lift}_{F_2, D}(\lambda a \cdot \llbracket T^h_D \rrbracket (\eta_1, \eta_2, \zeta, a)))(\text{in}_{va2}(a)) \\
&= \llbracket T^h_D \rrbracket (\eta_1, \eta_2, \zeta, a) \\
&= \llbracket T \rrbracket^{h|n_1}(\eta_2, a; \zeta) \\
&= \llbracket \text{do pure } x \leftarrow c \text{ in } T^h_D \rrbracket^{h|n_1}(\eta_2; \zeta)
\end{align*}
\]

- \textbf{WfTAPP}: We simplify both sides.

\[
\begin{align*}
\llbracket (z_i \mapsto v)^\eta_2 \rrbracket (\eta_1, \eta_2, \zeta) &= \llbracket (z_i \mapsto v) \rrbracket (\eta_1, \eta_2, \zeta) = \zeta(\llbracket v \rrbracket (\eta_1, \eta_2, \zeta)) \\
\llbracket (z_i \mapsto v)^{h|n_1} \rrbracket (\eta_2; \zeta) &= \zeta(\llbracket v \rrbracket (\eta_2))
\end{align*}
\]

Using the context-weakening Lemma 6.5.1, it follows that the two are equal.

- \textbf{WfTOP}: Let \( A \rightarrow B \) be the type annotations of \( op \) and \( A' \rightarrow B' \) its type in \( \Sigma \). Assume that \( c_{op} \) is the computation used to handle \( op \). We need to show that the following elements are equal:

\[
\begin{align*}
\llbracket c_{op}[x \mapsto v, k \mapsto (\text{fun } (y : B) \mapsto T^h_D)] \rrbracket (\eta_1, \eta_2, \zeta) \\
&= (h|n_1)(A \leq A')(\llbracket v \rrbracket (\eta_2), \lambda b \cdot \llbracket T \rrbracket^{h|n_1}(\eta_2, \llbracket B' \leq B \rrbracket b; \zeta))
\end{align*}
\]

We know that \((h|n_1)_{op} = \lambda a, \kappa \cdot \llbracket c_{op} \rrbracket (\eta_1, a, \kappa)\) by definition of \( h \). By induction we have

\[
\llbracket T^h_D \rrbracket (\eta_1, \eta_2, \llbracket B' \leq B \rrbracket b, \zeta) = \llbracket T \rrbracket^{h|n_1}(\eta_2, \llbracket B' \leq B \rrbracket b; \zeta).
\]

Taking into account type annotations, which require the use of \text{TYPENVSOLVE}, we arrive at

\[
\begin{align*}
\llbracket \Gamma_1, \Gamma_2, Z_D + (\text{fun } (y : B) \mapsto T^h_D) : B' \rightarrow D \rrbracket (\eta_1, \eta_2, \zeta) &= \lambda b \cdot \llbracket T \rrbracket^{h|n_1}(\eta_2, \llbracket B' \leq B \rrbracket b; \zeta).
\end{align*}
\]

We use Lemma 6.5.1 to weaken the contexts and Lemma 6.5.2 to switch to substitution.

\[
\begin{align*}
&= (\llbracket h \rrbracket_{\eta_1})(A \leq A')(\llbracket v \rrbracket (\eta_2), \lambda b \cdot \llbracket T \rrbracket^{h|n_1}(\eta_2, \llbracket B' \leq B \rrbracket b; \zeta)) \\
&= \llbracket c_{op} \rrbracket (\eta_1, A \leq A')(\llbracket v \rrbracket (\eta_2), \llbracket \text{fun } (y : B) \mapsto T^h_D \rrbracket (\eta_1, \eta_2, \zeta)) \\
&= \llbracket c_{op} \rrbracket (\eta_1, A \leq A')(\llbracket v \rrbracket (\eta_1, \eta_2, \zeta), \llbracket \text{fun } (y : B) \mapsto T^h_D \rrbracket (\eta_1, \eta_2, \zeta)) \\
&= \llbracket c_{op}[x \mapsto v, k \mapsto (\text{fun } (y : B) \mapsto T^h_D)] \rrbracket (\eta_1, \eta_2, \zeta)
\end{align*}
\]

\( \square \)
7.3 Soundness of a logic

The logics we are interested in are the ones where respects guarantees correct behaviour of handler denotations.

**Definition 7.3.1.** A logic is sound if \((\Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } E)\) implies that for any

\[(x_i : A_i)_{\eta}; (z_j : B_j \rightarrow *)_{\eta}) \vdash T_1 \sim T_2 \in E,\]

any \(\eta \sim \eta'\), any \(a_i \sim A_i\), \(a'_i\), and any \(f_j \sim B_j \rightarrow D f'_j\), we have

\[\llbracket T_1 \rrbracket^{\llbracket h \rrbracket}_{\eta}((a_i); (f_j)) \sim D \llbracket T_2 \rrbracket^{\llbracket h \rrbracket}_{\eta'}((a'_i); (f'_j)).\]

This is strongly linked to the requirement of Lemma 7.2.2. If operation cases respect equations, then using them with lift is safe with regard to the PER.

We now show that logic soundness is sufficient for a natural interaction between denotations of terms and denotations of effect theories. Proofs of logic soundness can then be done separately.

**Theorem 7.3.2.** If the logic used by the type system is sound, then for any \(\eta \sim \Gamma \eta'\) we have the following:

- \(\Gamma \vdash v : A\) implies \(\llbracket \Gamma \vdash v : A \rrbracket \eta \sim_A \llbracket \Gamma \vdash v : A \rrbracket \eta'\).
- \(\Gamma \vdash c : C\) implies \(\llbracket \Gamma \vdash c : C \rrbracket \eta \sim_C \llbracket \Gamma \vdash c : C \rrbracket \eta'\).
- \(\Gamma \vdash h : \Sigma \Rightarrow D\) implies \(\llbracket \Gamma \vdash h : \Sigma \Rightarrow D \rrbracket \eta \Rightarrow_{\Sigma \Rightarrow D} \llbracket \Gamma \vdash h : \Sigma \Rightarrow D \rrbracket \eta'\).

**Proof.** The proof proceeds by induction on the typing derivation. We present proofs for enough typing judgements to convey the general approach. All interesting cases, such as TYPEHANDLER or TYPELETREC, are covered.

- **TYPEINT:** The \(\sim_{\text{int}}\) is an identity relation on \(\mathbb{N}\), and thus \(\llbracket n \rrbracket \eta = n \sim_{\text{int}} n = \llbracket n \rrbracket \eta'\).
- **TYPELEFT:** For \(\Gamma \vdash \text{Left}_{A+B} v : A + B\), it follows by induction that \(\llbracket v \rrbracket \eta \sim_A \llbracket v \rrbracket \eta'\).

By definition of \(\sim_{A+B}\) we conclude that \(\llbracket v \rrbracket \eta \sim_{A+B} \llbracket v \rrbracket \eta'\).
- **TYPEFUN:** For any \(a \sim_A a'\) we have \(\eta(a) \sim_{\Gamma, x:A} (\eta', a')\), so by induction it holds that \(\llbracket \eta(a) \rrbracket \sim_{\text{C}} \llbracket \eta' \rrbracket (a')\), and it follows that \(\lambda a \cdot \llbracket \eta(a) \rrbracket \sim_{\text{C}} \lambda a' \cdot \llbracket \eta' \rrbracket (a')\).
- **TYPEHANDLER:** Similarly to the function case, we use the induction hypothesis for \(c_r\) to get \(\lambda a \cdot \llbracket c_r \rrbracket (\eta(a)) \sim_{\text{C}} \lambda a' \cdot \llbracket c_r \rrbracket (\eta'(a'))\). The other induction hypothesis states that \(\llbracket h \rrbracket \eta \sim_{\Sigma \Rightarrow D} \llbracket h \rrbracket \eta'\).

The assumption of logic soundness perfectly satisfies the requirement of Lemma 7.2.2.

\[\text{lift}_{\llbracket h \rrbracket \eta}(\lambda a \in \llbracket A \rrbracket, \llbracket c_r \rrbracket (\eta(a))) \sim_{\llbracket A \rrbracket \Sigma \Rightarrow D} \text{lift}_{\llbracket h \rrbracket \eta'}(\lambda a' \in \llbracket A \rrbracket, \llbracket c_r \rrbracket (\eta'(a')))\]

- **TYPESUBSUME:** Assume \(\Gamma \vdash v : A'\) was obtained from the premises \(A \leq A'\) and \(\Gamma \vdash v : A\) via subsumption. Then by induction we have \(\llbracket \Gamma \vdash v : A \rrbracket \eta \sim_A \llbracket \Gamma \vdash v : A \rrbracket \eta'\).

Lemma 7.2.4 completes the proof.

\[\llbracket A \leq A' \rrbracket (\llbracket \Gamma \vdash v : A \rrbracket \eta) \sim_{A'} \llbracket A \leq A' \rrbracket (\llbracket \Gamma \vdash v : A \rrbracket \eta')\]
• **TYPEABSURD:** Through induction we have \([v]_\emptyset \sim \emptyset [v]_\emptyset'\), with the statement 
\(\emptyset \text{fun}_\emptyset \sim \emptyset \text{fun}_\emptyset\) being vacuously true. By definition it follows that 
\(\emptyset \text{fun}_\emptyset (\emptyset [v]_\emptyset) \sim \emptyset \text{fun}_\emptyset (\emptyset [v]_\emptyset')\).

• **TYPEMATCHPAIR:** We know by induction that \([v]_\emptyset \sim \text{A}_\emptyset [v]_\emptyset'\), so if \((a, b) = [v]_\emptyset\) and \((a', b') = [v]_\emptyset'\), we have \(a \sim \text{A}_\emptyset a'\) and \(b \sim \text{B}_\emptyset b'\). By the IH for \([c]_\emptyset\), we arrive at 
\([c]_\emptyset (\eta, a, b) \sim [c]_\emptyset (\eta', a', b')\).

\[\text{match } v \text{ with } (x, y) \mapsto c\eta = [c]_\emptyset (\eta, a, b) \sim [c]_\emptyset (\eta', a', b')\]

• **TYPEOP:** Let \(\Sigma = \text{A}!\Sigma / \emptyset\). Suppose that \(\text{op} : \text{A}_\text{op} \to \text{B}_\text{op} \in \Sigma\) and that the function call has type annotations \(\text{A}_\text{op} \to \text{B}_\text{op}\) with appropriate subtypes.

\(\text{op}_{\text{A}_\text{op} \to \text{B}_\text{op}} (\text{v}; y, c)\eta = \text{in}_\text{op} ([\text{A}_\text{op} \leq \text{A}'_\text{op}][[\text{v}]\eta] ; \lambda b . [c]_\emptyset (\eta, \text{B}_\text{op} \leq \text{B}_\text{op} \circ b))\)

By induction it holds that \(([v]_\emptyset \sim [v]'_\emptyset\), and by Lemma 7.2.4 we have 
\([\text{A}_\text{op} \leq \text{A}'_\text{op}][[\text{v}]\eta] \sim [\text{A}'_\text{op} \leq \text{A}_\text{op}][[\text{v}]\eta']\).

If we assume \(b \sim \text{B}_\text{op} b'\), then by combining Lemma 7.2.4 with the IH for \([c]_\emptyset\) we get 
\([c]_\emptyset (\eta, \text{B}_\text{op} \leq \text{B}_\text{op} \circ b) \sim [c]_\emptyset (\eta', \text{B}_\text{op} \leq \text{B}_\text{op} \circ b')\).

The above fits rule 3 in the definition of \(\sim_{\text{A}!\Sigma / \emptyset}\).

\(\text{in}_\text{op} ([\text{A}_\text{op} \leq \text{A}'_\text{op}][[\text{v}]\eta] ; \lambda b . [c]_\emptyset (\eta, \text{B}_\text{op} \leq \text{B}_\text{op} \circ b))\)

\(\sim_{\text{A}!\Sigma / \emptyset} \text{in}_\text{op} ([\text{A}_\text{op} \leq \text{A}'_\text{op}][[\text{v}]\eta'] ; \lambda b'. [c]_\emptyset (\eta', \text{B}_\text{op} \leq \text{B}_\text{op} \circ b'))\)

• **TYPELETREC:** The denotation of \((\Gamma \vdash \text{let rec } f \ x : A \to \Sigma = \text{c}_1 \text{ in } c_2 : D)\) contains a least fixed point \(\tilde{f}\). We use the notation \(\tilde{f}(\eta)\) to denote that \(\eta\) is used in the least-fixed-point calculation.

\(\tilde{f}(\eta) = \mu f . \lambda a . [\Gamma, x : A, f : A \to \Sigma + c_1 : C] \eta, a, f)\)

We want to show that \(\tilde{f}(\eta) \sim_{\Sigma \to C} \tilde{f}(\eta')\) to obtain 
\([c_2]_\emptyset (\eta, \tilde{f}(\eta)) \sim_{\Sigma \to D} [c_2]_\emptyset (\eta, \tilde{f}(\eta'))\) by induction. The explicit construction of the least fixpoint is as follows:

\(\tilde{f}_0(\eta) = \lambda a . \bot\)

\(\tilde{f}_{n+1}(\eta) = \lambda a . [c_1]_\emptyset (\eta, \tilde{f}_n(\eta), a)\)

\(\tilde{f}(\eta) = \lor_n \tilde{f}_n(\eta)\)

We show that \(\forall n. \tilde{f}_n(\eta) \sim \tilde{f}(\eta')\) holds by induction on \(n\). It is clear that \(\tilde{f}_0(\eta) \sim \tilde{f}_0(\eta')\). And if it holds for \(n\), then it holds for \(n + 1\) directly from the IH for \(c_1\).

We have chains \((\tilde{f}_n(\eta))_n\) and \((\tilde{f}_n(\eta'))_n\), with \(\forall n. \tilde{f}_n(\eta) \sim \tilde{f}_n(\eta')\). Because \(\sim\) is chain complete, we have \(\tilde{f}(\eta) \sim \tilde{f}(\eta')\).
• TYPE DO: The semantics of sequencing is

\[\Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 : B!\Sigma/E \eta = \text{lift}_{\text{FB}1.2}(\lambda a \cdot c_2)(\eta, a)(\eta)\].

Using induction hypotheses, we can show \(c_1\eta \sim A!\Sigma/E \Rightarrow c_1\eta'\) and that \(a \sim a'\) implies \(c_2\eta, a) \sim A \rightarrow B!\Sigma/E \Rightarrow c_2(\eta', a')\). With this we can apply Corollary 7.2.3.

\[\text{lift}_{\text{FB}1.2}(\lambda a \cdot c_2)(\eta, a)(\eta) \sim B!\Sigma/E \Rightarrow \text{lift}_{\text{FB}1.2}(\lambda a' \cdot c_2)(\eta', a')(\eta')\]

• TYPE CASES ∪: The rule extends \(\Gamma \vdash h : \Sigma\) with the case \(\text{op}_{A \rightarrow B}(x; k) \mapsto c_{op}\). Interpretations are related component-wise, and by induction we already know that \(\llbracket h \eta \rrbracket \sim_{\Sigma=D} \llbracket h \eta' \rrbracket\), so components match for all operations in \(\Sigma\). The only thing left is to ensure that the components for \(\text{op}\) match.

\[\llbracket h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \eta \rrbracket_{op} = \lambda a \cdot \lambda \kappa . c_{op}(\eta, a, \kappa)\]

The IH states that \(a \sim a'\) and \(\kappa \sim B \rightarrow D \Rightarrow \kappa'\) imply \(c_{op}(\eta, a, \kappa) \sim D \Rightarrow c_{op}(\eta', a', \kappa')\), which satisfies the requirements for the component \(\text{op}\). It follows that

\[\llbracket h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \eta \rrbracket \sim_{\Sigma \cup \{ \text{op}_{A \rightarrow B} \}} \llbracket h \cup \{ \text{op}_{A \rightarrow B}(x; k) \mapsto c_{op} \} \eta' \rrbracket\]

as they are related for all operations in \(\Sigma \cup \{ \text{op} : A \rightarrow B \}\).

\[\square\]

**Proposition 7.3.3.** The full logic from Section 5.3 is not sound.

**Proof.** The full-logic rule for \(\text{respects}\) has no premises, so operation cases respect any equations. It is easy to find a counterexample even without using operations.

\[E_{CE} = \cdot : z_1 : \text{unit} \rightarrow *, z_2 : \text{unit} \rightarrow * \vdash z_1(\ ) \sim z_2(\ )\]

Since no operations are present, we can use empty cases \{\} for the \(\text{respects}\) judgement. We map to a type that has multiple distinct values—for instance, \(\text{int!}()\{\}\).

\[\cdot \vdash \{\} : \{\} \Rightarrow \text{int!}()\{\}\ \text{respects} E_{CE}\]

If the logic was sound, it would hold that

\[f_1 \sim_{\text{unit} \rightarrow \text{int!}()\{\}} f_1' \land f_2 \sim_{\text{unit} \rightarrow \text{int!}()\{\}} f_2' \Rightarrow \llbracket z_1(\ )\rrbracket_{\{\}}(f_1, f_2) \sim \text{int!}()\{\} \llbracket z_2(\ )\rrbracket_{\{\}}(f_1', f_2').\]

However, if we choose \(f_1 = f_1' = \lambda\_\_ . 1\) and \(f_2 = f_2' = \lambda\_\_ . 2\), this results in

\[\text{in}_{\text{val!}}(1) \sim_{\text{int!}()\{\}} \text{in}_{\text{val!}}(2).\]

Since the type contains no equations, the above is not true. \[\square\]
7.3.1 Soundness of equational logic

In Theorem 7.3.2 we are able to avoid the inner constructions of the logic system by assuming a sound logic. But equational logic heavily relies on the typing derivations. To that end, we need to include Theorem 7.3.2 as part of the soundness theorem and construct a mutually inductive proof.

**Theorem 7.3.4.** The equational logic introduced in Section 5.4 is sound, and for \( \eta \sim_\Gamma \eta' \) we have:

- \( \Gamma \vdash v : A \) implies \( \Gamma \vdash v : A \| \eta \sim_A \| \Gamma \vdash v : A \| \eta' \).
- \( \Gamma \vdash c : C \) implies \( \Gamma \vdash c : C \| \eta \sim_C \| \Gamma \vdash c : C \| \eta' \).
- \( \Gamma \vdash h : \Sigma \Rightarrow D \) implies \( \Gamma \vdash h : \Sigma \Rightarrow D \| \eta \sim_{\Sigma \Rightarrow D} \| \Gamma \vdash h : \Sigma \Rightarrow D \| \eta' \).
- For \( \Gamma : Z \vdash T : \Sigma \) and \( \zeta \sim_{D_1 D} \zeta' \), if \( H \sim_{\Sigma \Rightarrow D} H' \), then \( \| T \|_H(\eta ; \zeta) \sim_D \| T \|_H(\eta' ; \zeta') \).
- \( \Gamma \vdash v \equiv_A v' \), then \( \Gamma \vdash v : A \| \eta \sim_A \| \Gamma \vdash v' : A \| \eta' \).
- \( \Gamma \vdash c \equiv_C c' \), then \( \Gamma \vdash c : C \| \eta \sim_C \| \Gamma \vdash c' : C \| \eta' \).

\( \eta \sim_\Gamma \eta' \) if and only if \( \eta \sim_\Gamma \eta' \) and \( \eta' \sim_\Gamma \eta \).

\( \eta \sim_\Gamma \eta' \) due to the sub-derivation \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( \mathcal{E} \) implies

\[
\| T_1 \|^{h,v_i(\eta(a_i)):f_j)} \sim_D \| T_2 \|^{h,v_i(\eta(a_i)):f_j)}
\]

for any equation \( (x_i : A_i)_i ; (z_j : B_j \rightarrow *)_j \vdash T_1 \sim T_2 \) in \( \mathcal{E} \) whenever \( \forall i. a_i \sim_{A_i} a'_i \) and \( \forall j. f_j \sim_{B_j \rightarrow D} f'_j \). In equational logic, proving \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( \mathcal{E} \) is done by providing proofs for

\( \Gamma, (x_i : A_i)_i, (z_j : B_j \rightarrow D)_j \vdash T_1^h_{D_1} \equiv_D T_2^b_{D_2} \)

for all equations of \( \mathcal{E} \). The equation proofs are sub-derivations of \( \mathcal{E} \); therefore, by induction it follows that:

\( \| T_1^h_{D_1} \parallel (\eta, (\eta(a_i)_i), (f_j)_j) \sim_D \| T_2^{b D_2} \parallel (\eta', (\eta(a'_i)_i), (f'_j)_j) \). \]

We use Lemma 7.2.5 to transform the above into the shape required by soundness.

**Typing judgements:** The proofs directly coincide with those of Theorem 7.3.2.

**Well-formed templates:** The role of \( H \) in \( \| T \|_H \) is only visible in \( \text{WfTop} \). We state proofs for some other cases to show the general approach.

- \( \text{WfMatchPair} \): By induction we have \( \| v \|_H \) due to the sub-derivation of \( \Gamma \vdash v : A \times B \). For \( \| v \|_H = (a, b) \) and \( \| v \|_H = (a', b') \) it follows by definition of \( \sim_{AXB} \) that \( a \sim_B a' \) and \( b \sim_B b' \). The rest follows by the induction hypothesis for well-formed templates.

\[
\| \text{match } v \text{ with } (x, y) \mapsto T \|_H(\eta ; \zeta) = \| T \|_H(\eta, a, b; \zeta)
\]

\( \sim_D \)

\[
\| T \|_H(\eta', a', b'; \zeta') = \| \text{match } v \text{ with } (x, y) \mapsto T \|_H(\eta' ; \zeta')
\]
Transitivity and Symmetry of logic: The proofs for VeqTRANS, CeqTRANS, VenqSYM, and CeqSYM follow directly from $\sim$ being a partial equivalence relation.

Structural logic rules: The proofs are very close to those of Theorem 7.3.2, so we omit some. However, it is important to touch upon the problem of type annotations. For instance in VeqLEFT we have $\Gamma \vdash \text{Left}_{A_1 + B_1} \; v \equiv_{A + B} \text{Left}_{A_2 + B_2} \; v'$. The side conditions state that both sides need to type at $A + B$. There is no direct insight into the structure of $\Gamma \vdash \text{Left}_{A_1 + B_1} \; v : A + B$. In Proposition 6.4.9, where we established coherence of denotational semantics, we used the translation to skeletons. Following the same idea, we show that $\Gamma \vdash \text{Left}_{A_1 + B_1} \; v : A + B) = \Gamma \vdash \text{Left}_{A_2 + B_2} \; v : A + B)$ because skeletons of the terms match. This gives us a way to avoid annotation problems in the following proofs.

- VeqVar: The terms $v$ and $v'$ must be the same variable $x$. Because $\eta \sim \eta'$ their components are related, and it holds that $\llbracket x \rrbracket \eta \sim \llbracket x \rrbracket \eta'$.

- VeqLeft: The induction hypothesis states that $\llbracket v \rrbracket \eta \sim_A \llbracket v \rrbracket \eta'$, and by definition of $\sim_{A+B}$ it follows that $t_1(\llbracket v \rrbracket \eta) \sim_{A+B} t_1(\llbracket v' \rrbracket \eta')$. We simplify both sides of $\equiv$, where we switch type annotations as described previously.

- VeqFun: We first simplify both sides.

- VeqHandler: In equational logic, VeqHandler includes subtyping $\Sigma \leq \Sigma'$ and $D' \leq D$, which we have to take into account. Handlers are modelled through lifts, and we plan to use Lemma 7.2.2, with the $\lambda a . \llbracket c \rrbracket (\eta, a) \sim_C \llbracket c' \rrbracket (\eta', a')$ and $\llbracket h \rrbracket \eta \sim_{\Sigma = D'} h' \llbracket \eta'$ already having been satisfied through induction hypotheses. To show the remaining requirement of Lemma 7.2.2, assume $\Gamma_2 ; Z \vdash T_1 \sim T_2 \in \mathcal{E}$ and
let $\eta_2 \sim_{\Gamma_2} \eta_2'$ and $\zeta \sim_{\Sigma'} \zeta'$. We do this in two steps. First we use the induction hypothesis for $\Gamma \vdash h : \Sigma' \Rightarrow D' \text{ respects } \mathcal{E}$ (a premise of \textsc{Vehandler}) where we use $\eta \sim \eta$, $\eta_2 \sim_{\Gamma_2} \eta_2'$, and $\zeta \sim_{\Sigma'} \zeta'$. This is possible because $\sim$ is a PER, and if $x \sim x'$, then $x \sim x$.

$$\llbracket T_1 \rrbracket^{l h \eta}(\eta_2; \ldots) \sim_{D'} \llbracket T_2 \rrbracket^{l h \eta}(\eta_2; \ldots)$$

In the second step we use induction for $\Gamma_2 : \Sigma' \Rightarrow T_2 : \Sigma'$ to switch one side from $\llbracket h \rrbracket$ to $\llbracket T_2 \rrbracket$. The well-formedness judgement is a sub-derivation of $\vdash \mathcal{E} : \Sigma'$, which is itself a sub-derivation of $\vdash h : \Sigma' \Rightarrow D' \text{ respects } \mathcal{E}$, so we can use induction.

$$\llbracket T_2 \rrbracket^{l h \eta}(\eta_2; \ldots) \sim_{D'} \llbracket T_2 \rrbracket^{l h \eta}(\eta_2'; \ldots)$$

Finally, we use transitivity to arrive at

$$\llbracket T_1 \rrbracket^{l h \eta}(\eta_2; \ldots) \sim_{D'} \llbracket T_2 \rrbracket^{l h \eta}(\eta_2'; \ldots).$$

This satisfies the second requirement of Lemma 7.2.2, which gives us that the lifts are related at $A!\Sigma'/\mathcal{E} \Rightarrow D'$. We now correct the type to $A!\Sigma'/\mathcal{E} \Rightarrow D$, following Lemma 7.2.4.

- \textbf{CEQRET}: From $\Gamma \vdash v \equiv_A v'$ it follows by induction that $\llbracket v \rrbracket \eta \sim_A \llbracket v' \rrbracket \eta'$. This satisfies requirements of rule 2 in the definition of $\sim_{A!\Sigma'/\mathcal{E}}$.

$$\llbracket \text{ret } v \rrbracket \eta = \text{in}_{\text{val}}(\llbracket v \rrbracket \eta) \sim_{A!\Sigma'/\mathcal{E}} \text{in}_{\text{val}}(\llbracket v' \rrbracket \eta') = \llbracket \text{ret } v' \rrbracket \eta'$$

- \textbf{CEQMATCHSUM}: We have $\llbracket v \rrbracket \eta \sim_{A+B} \llbracket v' \rrbracket \eta'$ by induction, and we consider two possible cases:

  $\circ \llbracket v \rrbracket \eta = \iota_1(a)$ and $\llbracket v' \rrbracket \eta' = \iota_1(a')$, with $a \sim_A a'$, and by induction it follows that $\llbracket c_1 \rrbracket(\eta, a) \sim_C \llbracket c_1' \rrbracket(\eta', a')$.

  $\circ \llbracket v \rrbracket \eta = \iota_2(b)$ and $\llbracket v' \rrbracket \eta' = \iota_2(b')$, with $b \sim_A b'$, and by induction it follows that $\llbracket c_2 \rrbracket(\eta, b) \sim_C \llbracket c_2' \rrbracket(\eta', b')$.

This suffices for $\llbracket \text{match } v \text{ with } \ldots \rrbracket \eta \sim_C \llbracket \text{match } v' \text{ with } \ldots \rrbracket \eta'$.

- \textbf{CEQDO}: Through induction hypotheses we know that $\llbracket c_1 \rrbracket \eta \sim_{A!\Sigma/E} \llbracket c_1' \rrbracket \eta'$ and that $\lambda a . \llbracket c_2 \rrbracket(\eta, a) \sim_{A-B!\Sigma/E} \lambda a' . \llbracket c_2' \rrbracket(\eta', a')$. The result follows directly from Corollary 7.2.3.

$$\llbracket \text{do } x \leftarrow c_1 \text{ in } c_2 \rrbracket \eta = \text{lift}_{F_{\text{do}1.2}}(\lambda a . \llbracket c_2 \rrbracket(\eta, a))(\llbracket c_1 \rrbracket \eta) \sim_{B!\Sigma/E} \text{lift}_{F_{\text{do}1.2}}(\lambda a' . \llbracket c_2' \rrbracket(\eta', a'))(\llbracket c_1' \rrbracket \eta' \llbracket \text{do } x \leftarrow c_1' \text{ in } c_2' \rrbracket \eta'$$

- \textbf{CEQAPP}: The IH for $\Gamma \vdash v_1 \equiv_{A-C} v_1'$ gives us $\llbracket v_1 \rrbracket \eta \sim_{A-C} \llbracket v_1' \rrbracket \eta'$, and from $\Gamma \vdash v_2 \equiv_A v_2'$ we have $\llbracket v_2 \rrbracket \eta \sim_A \llbracket v_2' \rrbracket \eta'$. We use the definition of $\sim_{A-C}$ to conclude

$$\llbracket v_1 \odot v_2 \rrbracket \eta = \text{lift}_{F_{\text{app}1.2}}(\llbracket v_1 \rrbracket \eta)(\llbracket v_2 \rrbracket \eta) \sim_C (\llbracket v_1' \rrbracket \eta')(\llbracket v_2' \rrbracket \eta') = \llbracket v_1' \odot v_2' \rrbracket \eta'.$$
\textbf{\(\beta\)-laws:} Since \(\beta\)-laws directly mirror the operational semantics, we use Theorem 6.5.3, which states that \(c \sim c'\) implies \([c] = [c']\). The problem is thus reduced to \([c]_\eta \sim [c']_\eta'\), which is true by induction.

\textbf{\(\eta\)-laws:} The expansion rules for pairs, sums, and lists are similar, so we only prove the case of lists, which is the most complex.

\begin{itemize}
  \item \(\eta\)-\textsc{Unit}: If \(\Gamma \vdash v \equiv \text{\textsc{unit}}()\) then \([v]_\eta \in \text{\textsc{unit}}\) which is the singleton \(*\).

  \[\[v\]_\eta = *_{\text{\textsc{unit}}} * = \[\text{\textsc{unit}}\]_\eta'\]

  \item \(\eta\)-\textsc{Fun}: Keep in mind that \(f\) is a variable in \(\Gamma\). We use Lemma 6.5.1 to weaken the context to \(\Gamma, x : A\) when needed. By definition of variable denotations, we get \([\Gamma, x : A \vdash x : A][\eta', a'] = a'\).

  \[
  \begin{align*}
  \llbracket \Gamma \vdash f : A \rightarrow C \rrbracket_{\eta'} &= \lambda a'. (\llbracket \Gamma \vdash f : A \rightarrow C \rrbracket_{\eta'}(a')) \\
  &= \lambda a'. (\llbracket \Gamma, x : A \vdash f : A \rightarrow C \rrbracket_{\eta', a'}(a')) \\
  &= \lambda a'. (\llbracket \Gamma, x : A \vdash f : A \rightarrow C \rrbracket_{\eta', a'}(\llbracket \Gamma, x : A \vdash x : A \rrbracket_{\eta', a'})) \\
  &= \llbracket \Gamma \vdash \text{\textsc{fun}} (x : A) \mapsto f \ x : C \rrbracket_{\eta'}
  \end{align*}
  \]

  From \(\Gamma \vdash f : A \rightarrow C\) it follows that \(\llbracket f \rrbracket_{\eta} \sim_{A \rightarrow C} \llbracket f \rrbracket_{\eta'}\), and therefore by definition of \(\sim_{A \rightarrow C}\) it holds that \(\llbracket f \rrbracket_{\eta} \sim \llbracket \text{\textsc{fun}} (x : A) \mapsto f \ x \rrbracket_{\eta'}\).

  \item \(\eta\)-\textsc{Empty}: By Lemma 6.5.2 we can resolve the substitution as

  \[
  \llbracket \Gamma_1, \Gamma_2 \vdash c[e \mapsto v] : C \rrbracket(\eta_1, \eta_2) = \llbracket \Gamma_1, e : \text{\textsc{empty}}, \Gamma_2 \vdash c : C \rrbracket(\eta_1, \llbracket v \rrbracket(\eta_1, \eta_2), \eta_2).
  \]

  We know that \(\llbracket c \rrbracket : \llbracket \Gamma_1 \rrbracket\) is an element of \(\emptyset \times \llbracket \Gamma_2 \rrbracket \rightarrow \llbracket C \rrbracket\). Any Cartesian product that includes an empty set is itself an empty set, and therefore the statement \(\forall x, y \in \emptyset. \llbracket c \rrbracket(x) = \text{\textsc{emptyfun}}_{\llbracket C \rrbracket}(y)\) is vacuously true.

  \[
  \begin{align*}
  \llbracket c[e \mapsto v] \rrbracket(\eta_1, \eta_2) &= \llbracket c \rrbracket(\eta_1, \llbracket v \rrbracket(\eta_1, \eta_2), \eta_2) \\
  &= \text{\textsc{emptyfun}}_{\llbracket C \rrbracket}(\llbracket v \rrbracket(\eta_1, \eta_2)) \\
  &= \llbracket \text{\textsc{absurd}}_{\llbracket C \rrbracket} v \rrbracket(\eta_1, \eta_2) \\
  &= \sim_{\llbracket C \rrbracket} \llbracket \text{\textsc{absurd}}_{\llbracket C \rrbracket} v \rrbracket(\eta'_1, \eta'_2)
  \end{align*}
  \]

  The last step comes from the induction hypothesis for well-typed computations.

  \item \(\eta\)-\textsc{List}: We use Lemma 6.5.2 to switch back and forth between substitutions. By the typing derivation for \(c\) and Lemma 6.5.1 for context weakening it follows that \(\llbracket c \rrbracket(\eta_1, x, \eta_2) = \llbracket c \rrbracket(\eta_1, x, \eta_2, y, z)\) for any \(x, y, z\). By definition of semantics it also holds
Because both sides of the logic equality are well typed, we have the IH

\[ \llbracket c[l \mapsto v] \rrbracket(\eta_1, \eta_2) \sim_C \llbracket c[l \mapsto v] \rrbracket(\eta'_1, \eta'_2) \]

and by the previously proven equality it follows that

\[ \llbracket c[l \mapsto v] \rrbracket(\eta_1, \eta_2) \sim_C \llbracket \text{match } v \text{ with } [ ] \mapsto c[l \mapsto [ ]_A] \mapsto x :: xs \mapsto c[l \mapsto x :: xs] \rrbracket(\eta'_1, \eta'_2) \]

\* \eta-DO: We take a closer look at the lift in

\[ \llbracket \text{do } x \mapsto c \text{ in } \text{ret } x \rrbracket = (\text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{ret } x)(\eta, a)) (\llbracket c \rrbracket \eta) \]

\[ = (\text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{in}_{\text{val}}(a))) (\llbracket c \rrbracket \eta). \]

Because \((\text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{ret } x)(\eta, a))\) is defined through the recursion principle, it is the unique strict continuous function for which

\[ (\text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{in}_{\text{val}}(a)))(\text{in}_{\text{val}}(x)) = (\lambda a \cdot \text{in}_{\text{val}}(a))(x) \]

\[ = \text{in}_{\text{val}}(x) \]

\[ (\text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{in}_{\text{val}}(a)))(\text{in}_{\text{op}}(x, \kappa)) = (\text{F}_{\mathbb{I}})_0(\text{op}(x, \text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{in}_{\text{val}}(a)) \circ \kappa) \]

\[ = \text{in}_{\text{op}}(x, \text{lift}_{\Gamma_0\Sigma_1\mathbb{I}}(\lambda a \cdot \text{in}_{\text{val}}(a)) \circ \kappa) \]

If we denote the lift with \(\mathcal{F}\), we see that \(\mathcal{F}(\text{in}_{\text{val}}(x)) = \text{in}_{\text{val}}(x)\), and for all operations \(\text{op}\) we have \(\mathcal{F}(\text{in}_{\text{op}}(x, \kappa)) = \text{in}_{\text{op}}(x, \mathcal{F}_0 \circ \kappa)\). This is also satisfied by the identity function, and by uniqueness the lift is the identity function. Because \(c\) is well typed, we have \(\llbracket c \rrbracket \eta \sim \llbracket c \rrbracket \eta'\), and it follows that \(\llbracket c \rrbracket \eta \sim \llbracket \text{do } x \mapsto c \text{ in } \text{ret } x \rrbracket \eta'\).

**Inheriting equations:** Recall the definition of OOTB.

\[
\frac{(x_i : A_i)_i \mid (z_j : B_j \rightarrow \ast)_j \vdash T_1 \sim T_2) \in \mathcal{E} \quad A \Sigma / \mathcal{E} \leq C \quad (\Gamma \vdash \nu_i : A_i)_i \quad (\Gamma \vdash u_j : B_j \rightarrow A \Sigma / \mathcal{E})_j}{\Gamma \vdash T_1 \text{ in } \Sigma / \mathcal{E}(\nu_i \mapsto u_i)_i \equiv_C T_2 \text{ in } \Sigma / \mathcal{E}(\nu_i \mapsto u_i)_i} \text{ OOTB}
\]
By induction we have \( \llbracket v_i \rrbracket \eta \sim \llbracket v_i \rrbracket \eta' \) and \( \llbracket u_j \rrbracket \eta \sim \llbracket u_j \rrbracket \eta' \) for all \( i \) and \( j \). We remove substitution with Lemma 6.5.2, which gives us that
\[
\llbracket T^I_{A!\Sigma/E}(x_i \mapsto v_i), (z_j \mapsto u_j) \rrbracket \eta = \llbracket T^I_{A!\Sigma/E}(\llbracket v_i \rrbracket \eta), (\llbracket u_j \rrbracket \eta) \rrbracket \eta.
\]
The use of \( I \) in \( (_I)^\Sigma_{\Delta/E} \) is a substitute for “identity cases”, and when using Lemma 7.2.5, it is clear that the equivalent of \( \llbracket I \rrbracket \eta \) is \( F_Z^{\Delta A} \).
\[
\llbracket T^I_{A!\Sigma/E}(\llbracket v_i \rrbracket \eta), (\llbracket u_j \rrbracket \eta) \rrbracket = \llbracket T^I_{A!\Sigma/E}(\llbracket v_i \rrbracket \eta), (\llbracket u_j \rrbracket \eta) \rrbracket.
\]
Because \( T_1 \sim T_2 \in E \) it follows by definition of \( \sim_{A!\Sigma/E} \) that
\[
\llbracket T^I_{A!\Sigma/E}(\llbracket v_i \rrbracket \eta), (\llbracket u_j \rrbracket \eta) \rrbracket \sim_{A!\Sigma/E} \llbracket T^I_{A!\Sigma/E}(\llbracket v_i \rrbracket \eta'), (\llbracket u_j \rrbracket \eta') \rrbracket.
\]
The type is corrected to \( C \) with the subtype judgement \( A!\Sigma/E \leq C \) combined with Lemma 7.2.4. \( \square \)

### 7.3.2 Soundness of predicate logic

We first define the denotation of formulae. Each well-formed formula \( \Gamma \vdash \phi : \text{form} \) is interpreted as a subset of \( \llbracket \Gamma \rrbracket \). We shorten \( \llbracket \Gamma \vdash \phi : \text{form} \rrbracket \) to \( \llbracket \phi \rrbracket \) when no confusion can arise.

\[
\begin{align*}
\llbracket \Gamma \vdash T : \text{form} \rrbracket & = \llbracket \Gamma \rrbracket \\
\llbracket \Gamma \vdash \bot : \text{form} \rrbracket & = \emptyset \\
\llbracket \Gamma \vdash \phi_1 \land \phi_2 : \text{form} \rrbracket & = \llbracket \phi_1 \rrbracket \cap \llbracket \phi_2 \rrbracket \\
\llbracket \Gamma \vdash \phi_1 \lor \phi_2 : \text{form} \rrbracket & = \llbracket \phi_1 \rrbracket \cup \llbracket \phi_2 \rrbracket \\
\llbracket \Gamma \vdash \phi_1 \rightarrow \phi_2 : \text{form} \rrbracket & = \{ \eta \mid \eta \in \llbracket \phi_1 \rrbracket \Rightarrow \eta \in \llbracket \phi_2 \rrbracket \} \\
\llbracket \Gamma \vdash \forall x : A. \phi : \text{form} \rrbracket & = \{ \eta \mid \exists a \sim_A a. (\eta, a) \in \llbracket \Gamma, x : A \vdash \phi : \text{form} \rrbracket \} \\
\llbracket \Gamma \vdash \exists x : A. \phi : \text{form} \rrbracket & = \{ \eta \mid \forall a \sim_A a. (\eta, a) \in \llbracket \Gamma, x : A \vdash \phi : \text{form} \rrbracket \} \\
\llbracket \Gamma \vdash v_1 \equiv_a v_2 : \text{form} \rrbracket & = \{ \eta \mid \llbracket \Gamma \vdash v_1 : A \rrbracket \eta \sim_A \llbracket \Gamma \vdash v_2 : A \rrbracket \eta \} \\
\llbracket \Gamma \vdash c_1 \equiv_C c_2 : \text{form} \rrbracket & = \{ \eta \mid \llbracket \Gamma \vdash c_1 : C \rrbracket \eta \sim_C \llbracket \Gamma \vdash c_2 : C \rrbracket \eta \} \\
\llbracket \Gamma \vdash h_1 \equiv_{z\in D} h_2 : \text{form} \rrbracket & = \{ \eta \mid \llbracket \Gamma \vdash h_1 : \Sigma \Rightarrow D \rrbracket \eta \sim_{z\in D} \llbracket \Gamma \vdash h_2 : \Sigma \Rightarrow D \rrbracket \eta \}
\end{align*}
\]

Certain lemmas from Section 6.5 need to be extended to formulae.

**Lemma 7.3.5.** Let \( \eta_1 \in \llbracket \Gamma_1 \rrbracket , \eta_2 \in \llbracket \Gamma_2 \rrbracket \) and \( b \in \llbracket B' \rrbracket \). Assume \( x' \) is a fresh variable in the formula \( \phi \).

\[
(\eta_1, \eta_2) \in \llbracket \Gamma_1, \Gamma_2 \vdash \phi : \text{form} \rrbracket \iff (\eta_1, b, \eta_2) \in \llbracket \Gamma_1, x' : B, \Gamma_2 \vdash \phi : \text{form} \rrbracket
\]

**Proof.** This is the formula equivalent of Lemma 6.5.1. Safety of weakening the context in formula judgements is shown in the formalisation. The rest of the proof proceeds by induction on derivation of \( \Gamma_1, \Gamma_2 \vdash \phi : \text{form} \). The only nontrivial cases are judgements for \( \equiv \). We describe the proof outline for \( \text{WfVEQ} \) where \( \phi = v_1 \equiv_A v_2 \). By Lemma 6.5.1 it follows that \( \llbracket v_1 \rrbracket (\eta_1, \eta_2) \sim_A \llbracket v_2 \rrbracket (\eta_1, \eta_2) \) and therefore
\[
\llbracket v_1 \rrbracket (\eta_1, \eta_2) \sim_A \llbracket v_2 \rrbracket (\eta_1, \eta_2) \iff \llbracket v_1 \rrbracket (\eta_1, b, \eta_2) \sim_A \llbracket v_2 \rrbracket (\eta_1, b, \eta_2).
\]

\( \square \)
Lemma 7.3.6. Assume $\Gamma_1, \Gamma_2 \vdash u : B$ and let $\eta_1 \in [\Gamma_1]$ and $\eta_2 \in [\Gamma_2]$.

$$(\eta_1, [u](\eta_1, \eta_2), \eta_2) \in [\Gamma_1, x' : B, \Gamma_2 \vdash \phi : \text{form}] \iff (\eta_1, \eta_2) \in [\Gamma_1, \Gamma_2 \vdash \phi [x' \mapsto u] : \text{form}]$$

Proof. This is the formula equivalent of Lemma 6.5.2. Just like with Lemma 7.3.5 the proof proceeds by induction on derivation of well-formedness, where most parts of the proof are entirely structural, and in when faced with $\equiv$ we use Lemma 6.5.2. □

The addition of recursion also requires a judgement that asserts whether a formula is admissible, which is crucial for the soundness of the induction principle in the logic. We show that $x$ is admissible in $\phi$ ensures chain-completeness of $[\phi]$ for the component of the variable $x$.

Lemma 7.3.7. Let $\Gamma_1, x : A, \Gamma_2 \vdash \phi : \text{form}$ and $x$ is admissible in $\phi$. Assume we have $\eta_1 \in [\Gamma_1]$, $\eta_2 \in [\Gamma_2]$, and a chain $(a_i)_i$ in $[A]$.

$$(\forall i. (\eta_1, a_i, \eta_2) \in [\phi]) \Rightarrow (\eta_1, \lor, a_i, \eta_2) \in [\phi]$$

Proof. The proof proceeds by induction on the judgement $x$ is admissible in $\phi$.

- The variable $x$ does not appear in $\phi$. As a result we can use Lemma 7.3.5 on both sides to remove the component of $x$. The resulting statement is clearly true.

$$(\forall i. (\eta_1, \eta_2) \in [\phi]) \Rightarrow (\eta_1, \eta_2) \in [\phi]$$

- $\phi$ is of form $\top, \bot$. For $\top$, it holds because $[A]$ is a predomain and therefore chain complete. For $\bot$ it holds because $\forall i. (\eta_1, a_i, \eta_2) \in [\bot]$ cannot be satisfied.

- $\phi$ is of form $\varphi_1 \land \varphi_2$, where $x$ is admissible in $\varphi_1$ and $x$ is admissible in $\varphi_2$. From $\forall i. (\eta_1, a_i, \eta_2) \in [\varphi_1 \land \varphi_2]$, it follows by definition that $\forall i. (\eta_1, a_i, \eta_2) \in [\varphi_1]$ and $\forall i. (\eta_1, a_i, \eta_2) \in [\varphi_2]$. By induction $(\eta_1, \lor, a_i, \eta_2)$ is in $[\varphi_1]$ and $[\varphi_2]$, and therefore in $[\varphi_1 \land \varphi_2]$.

- $\phi$ is of form $\varphi_1 \lor \varphi_2$, where $x$ is admissible in $\varphi_1$ and $x$ is admissible in $\varphi_2$. If we assume $\forall i. (\eta_1, a_i, \eta_2) \in [\varphi_1 \lor \varphi_2]$, then for any $i$ the element $(\eta_1, a_i, \eta_2)$ is either in $[\varphi_1]$ or in $[\varphi_2]$. This induces two chains, one in $[\varphi_1]$ and one in $[\varphi_2]$ (with one of them possibly finite). We denote the suprema of the induced chains as $a^1$ and $a^2$ respectively. By induction we have $(\eta_1, a^1, \eta_2) \in [\varphi_1]$ and $(\eta_1, a^2, \eta_2) \in [\varphi_2]$. One (or both) of $a^1$ and $a^2$ must be equal to $\lor, a_i$, and it follows that $(\eta_1, \lor, a_i, \eta_2) \in [\varphi_1 \lor \varphi_2]$.

- $\phi$ is of form $\varphi_1 \Rightarrow \varphi_2$, where $x$ does not occur in $\varphi_1$ and $x$ is admissible in $\varphi_2$. By definition

$$[\varphi_1 \Rightarrow \varphi_2] = \{(\eta_1, a, \eta_2) \mid (\eta_1, a, \eta_2) \in [\varphi_1] \Rightarrow (\eta_1, a, \eta_2) \in [\varphi_2]\},$$

but because $x$ does not occur in $\varphi_1$, we can use Lemma 7.3.5 and simplify it to

$$[\varphi_1 \Rightarrow \varphi_2] = \{(\eta_1, a, \eta_2) \mid (\eta_1, a, \eta_2) \in [\varphi_1] \Rightarrow (\eta_1, a, \eta_2) \in [\varphi_2]\}.$$
\[ \varphi \text{ is of form } \forall y : B. \varphi', \text{ where } x \neq y \text{ and } x \text{ is admissible in } \varphi'. \text{ Assume that for all } i, \text{ we have } (\eta_i, a_i, \eta_2) \in [\forall x. \varphi'], \text{ and by induction } (\eta_1, \sqrt{a_i, \eta_2, b}) \in [\varphi']. \]

\[ \varphi \text{ is of form } v \equiv A' v', c \equiv \mathcal{C} c', \text{ or } h \equiv \mathcal{D} h'. \text{ The result follows because } \sim \text{ is chain complete by definition.} \]

\[ \square \]

Theorem 7.3.8. The predicate logic from Section 5.5 is sound, and we have the following:

- If \( \Gamma \vdash v : A \), then for \( \eta \sim \Gamma \eta' \) it follows that \( [\Gamma \vdash v : A] \eta \sim A [\Gamma \vdash v : A] \eta' \).
- If \( \Gamma \vdash c : C \), then for \( \eta \sim \Gamma \eta' \) it follows that \( [\Gamma \vdash c : C] \eta \sim C [\Gamma \vdash c : C] \eta' \).
- If \( \Gamma \vdash h : \Sigma \Rightarrow D \), then for \( \eta \sim \Gamma \eta' \) we have \( [\Gamma \vdash h : \Sigma \Rightarrow D] \eta \sim \Sigma \Rightarrow D [\Gamma \vdash h : \Sigma \Rightarrow D] \eta' \).
- If \( \Gamma ; Z : T : \Sigma, \eta \sim \Gamma \eta' \), then for \( (\eta, \xi, \iota) \sim \Gamma ; Z \Rightarrow D (\eta', \iota') \) and \( H \sim \Sigma \Rightarrow D H' \) it holds that \( [T]^H (\eta, \xi, \iota) \sim \Sigma \Rightarrow D [T]^H (\eta', \iota') \).
- If \( \Gamma \vdash \eta \vdash \psi_1, \ldots, \psi_n \vdash \varphi \), then for any \( \eta \sim \Gamma \eta \) where \( \eta \in [\psi_1 \land \cdots \land \psi_n] \), we have \( \eta \in [\varphi]. \)

Proof. Some of the proofs are close to those in Theorem 7.3.4, and we do not repeat them. The proof proceeds by induction on the mutually defined relations of well-formedness, typing, and logic proofs.

- Soundness: Let \( (x_i : A_i)_{j}, (z_j : B_j \rightarrow *)_{j} \vdash T_i \sim T_2 \) be an equation in \( \mathcal{E} \). Assume \( \forall i, a_i \sim A_i, a_i' \) and \( \forall j, f_j \sim B_j \rightarrow D, f_j' \). To construct \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( \mathcal{E} \) in predicate logic, one must provide proofs for
  \[ \Gamma, (x_i : A_i)_{i}, (z_j : B_j \rightarrow D)_{j} \vdash c. \Gamma^h \equiv \mathcal{E}^h \]
  for all equations of \( \mathcal{E} \). Note that hypotheses are empty, so no additional restrictions are required. The equation proofs are sub-derivations of respects, and therefore
  \[ [T^h_1] \eta \equiv \mathcal{E} [T^h_2] \eta_n. \]
  We use Lemma 7.2.5 to transform it into the shape required by soundness.
  \[ [T^h_1] \eta \equiv \mathcal{E} [T^h_2] \eta_n. \]
  We then use the IH for \( T_2 \) and the IH for \( \Gamma \vdash h : \Sigma \Rightarrow D \), which is a sub-derivation of respects, in order to switch to \( (\eta', a'_i, (f_j')) \).

- \( \equiv \) Judgements: The proofs for equality judgements are nearly identical to the ones in Theorem 7.3.4, seeing as they do not interact with hypotheses.

We repeat the proof for CeqDO to show that the same approach still works. By induction we have \( [c_1] \eta \sim A_{\Sigma \mathcal{E}} [c_1] h \). For any \( a \sim A a \) we know by Lemma 7.3.5 that \( (\eta, a) \in [\psi_1 \land \cdots \land \psi_n] \), and thus by induction \( [c_2] (\eta, a) \sim B_{\Sigma \mathcal{E}} [c_2] (\eta, a) \). To use Corollary 7.2.3, we need to state this in terms of \( A \rightarrow B_{\Sigma \mathcal{E}} \). If \( a \sim A a' \), then \( (\eta, a) \sim \Gamma, x : A (\eta, a') \), and by IH for \( \Gamma \vdash c' : B_{\Sigma \mathcal{E}}, \) we have \( [c'_2] (\eta, a) \sim B_{\Sigma \mathcal{E}} [c'_2] (\eta, a) \). Due to transitivity of \( \sim B_{\Sigma \mathcal{E}}, \) we get \( [c_2] (\eta, a) \sim B_{\Sigma \mathcal{E}} [c'_2] (\eta, a) \) so it follows that
  \[ \lambda a. [c_2] (\eta, a) \sim A \rightarrow B_{\Sigma \mathcal{E}} \lambda a. [c_2] (\eta, a). \]
The rest follows directly from Corollary 7.2.3.

\[
[\text{do } x \leftarrow c_1 \text{ in } c_2] \eta = (\text{lift}_{\text{B1,2}}(\lambda a \cdot [c_2]((\eta, a))))([c_1] \eta)
\]

Because of the formulation of the theorem for formulae, we end up with \([v] \eta \sim [v'] \eta\), as opposed to \([v] \eta \sim [v'] \eta'\) in Theorem 7.3.4. All terms in logic judgements need to be well typed, so it holds that \([v'] \eta \sim [v'] \eta'\). We can use transitivity of \(\sim\) to get \([v] \eta \sim [v'] \eta'\), as we did in the above proof for \text{CeqDO}. Therefore, the induction hypotheses at our disposal are equivalent to those in Theorem 7.3.4.

And as for the modified \(\equiv\) judgements, the exclusion of explicit subtyping only makes proofs of \text{VeqVAR}, \text{VeqHANDLER}, and \text{OOTB} simpler. The aforementioned rules included subtyping in order for Lemma 5.4.8 to hold, but this has been rendered obsolete due to inclusion of subsumption rules. The proofs for \text{VeqSUBSUME} and \text{CeqSUBSUME} follow directly from Lemma 7.2.4. Symmetry rules and transitivity rules for operation cases follow from \(\sim\) being a partial equivalence relation. The proof of \text{HeqEXTEND} is entirely structural. We briefly cover the rules \text{DoLOOP} and \text{HandleLoop} below.

- \text{IsHyp}: The assumption of the theorem states \(\eta \in [\psi_1 \land \cdots \land \psi_n]\), which is by definition equivalent to \(\eta \in [\psi_1] \cap \cdots \cap [\psi_n]\). \text{IsHyp} can only be used when \(\varphi\) is one of the hypotheses \(\psi_1, \ldots, \psi_n\), and therefore \(\eta \in [\varphi]\).

- \(\top\): We have \(\eta \sim \top \eta\) by assumption, so it follows that \(\eta \in [\top]\).

- \(\bot\): The premise of the rule is \(\Gamma \vdash \bot\), so by induction we have \(\eta \in \emptyset\). The statement is vacuously true.

- \(\land\): By induction we have \(\eta \in [\varphi_1]\) and \(\eta \in [\varphi_2]\), so \(\eta \in [\varphi_1 \land \varphi_2]\) by definition.

- \(\land\text{ELEFT}\) and \(\land\text{ELRIGHT}\): We have the IH \(\eta \in [\varphi_1 \land \varphi_2]\), so it follows that \(\eta \in [\varphi_1]\) and \(\eta \in [\varphi_2]\).

- \(\lor\text{LEFT}\) and \(\lor\text{ELEFT}\): In the first case induction gives us \(\eta \in [\varphi_1]\), and in the second case \(\eta \in [\varphi_2]\). In both cases it holds that \(\eta \in [\varphi_1] \cup [\varphi_2]\), which implies \(\eta \in [\varphi_1 \lor \varphi_2]\).

- \(\lor\): By induction we have \(\eta \in [\varphi_1] \Rightarrow \eta \in [\varphi]\) and \(\eta \in [\varphi_2] \Rightarrow \eta \in [\varphi]\), so it follows that \(\eta \in [\varphi_2] \cup [\varphi_2] \Rightarrow \eta \in [\varphi]\).

- \(\Rightarrow\): The IH states that if \(\eta \in [\psi_1 \land \cdots \land \psi_n \land \varphi_1]\), then \(\eta \in [\varphi_2]\). Therefore, from the assumption \(\eta \in [\psi_1 \land \cdots \land \psi_n]\) it follows that if \(\eta \in [\varphi_1]\), then also \(\eta \in [\varphi_2]\).

- \(\Rightarrow\): Since we have \(\eta \in [\varphi_1 \Rightarrow \varphi_2]\) and \(\eta \in [\varphi_1]\), we have \(\eta \in [\varphi_2]\) by definition of \([\varphi_1 \Rightarrow \varphi_2]\).

- \(\forall\): For \((\eta, a) \sim_{(\Gamma, x : A)} (\eta, a)\) we have by induction that if \((\eta, a) \in [\psi_1 \land \cdots \land \psi_n]\), then also \((\eta, a) \in [\varphi]\). By assumption \(\eta \in [\psi_1 \land \cdots \land \psi_n]\) and by Lemma 7.3.5, we can weaken the context to \((\eta, a) \in [\psi_1 \land \cdots \land \psi_n]\) for any \(a \in [A]\). Therefore, for any \(a \sim_A a\) we have \((\eta, a) \in [\varphi]\), so by definition \(\eta \in [\forall x : A. \varphi]\).
• ∀EL: The IH for \( \forall x : A. \varphi \) states that \((\eta, a) \in [\varphi]\) for any \( a \sim_A a \). From \( \Gamma \vdash v : A \) and \( \eta \sim \Gamma \eta \), it follows that \([v] \eta \sim_A [v] \eta \). Therefore, \((\eta, [v] \eta) \in [\varphi]\) and by Lemma 7.3.6 it follows that \( \eta \in [\varphi[x \mapsto v]] \).

• ∃IN: The induction hypotheses for \( \Gamma \vdash v : A \) and \( \Gamma \vdash \varphi[x \mapsto v] \), combined with Lemma 7.3.6, result in \([v] \eta \sim_A [v] \eta \) and \((\eta, [v] \eta) \in [\varphi]\). There exists \( a \sim_A a \)—namely \([v] \eta \sim_A [v] \eta \), for which \((\eta, a) \in [\varphi]\).

From the IH for the premise \((\Gamma \vdash \exists x : A. \psi)\) we know that there exists some \( a_\psi \sim_A a_\psi \) for which \((\eta, a_\psi) \in [\psi]\). From \((\eta, a_\psi) \in [\psi]\) and \( \eta \in [\psi_1 \land \cdots \land \psi_n]\), it follows that \((\eta, a_\psi) \in [\psi_1, \ldots, \psi_n, \psi] \) after weakening with Lemma 7.3.5. Using the IH for \((\Gamma, x : A \mid \psi_1, \ldots, \psi_n, \psi \vdash \varphi)\) results in \((\eta, a_\psi) \in [\varphi]\). By Lemma 7.3.5, that is equal to \( \eta \in [\varphi] \) because \( x \) is not present in \( \varphi \).

• DOLOOP and HANDLELOOP: It is straightforward to check that the denotation of the recursive function is \((\_ \_ \_ \_ 1)\). This also means that the denotation of the whole construct is \( \bot \). Since handlers and the lift used for do are strict continuous functions, both sides receive the denotation \( \bot \), which is self-related.

• INDUCTION: Recall the judgment for induction.

\[
\begin{align*}
\text{INDUCTION} & \quad f \text{ is admissible in } \varphi \quad \Gamma, x : A \vdash \varphi[f \mapsto \text{(fun } _ \_ \_ \mapsto \text{ret} x)]\[1cm]
\Gamma, x : A_\text{op}, k : B_\text{op} \rightarrow \vartriangle \mid \Psi, (\forall y : B_\text{op}. \varphi[f \mapsto \text{(fun } _ \_ \_ \mapsto k y)])[1cm]
\vdash \varphi[f \mapsto \text{(fun } _ \_ \_ \mapsto \text{op}_A \text{op}_B (x \cdot y \cdot k y))] \quad \text{op}_A : \text{op}_B \rightarrow \Sigma
\end{align*}
\]

By definition \( \eta \in [\forall f. \varphi] \) holds if for all \( f \sim_{\text{unit} \rightarrow A! \Sigma/\varepsilon} f \), we have \((\eta, f) \in [\varphi]\).

Functions of \([\text{unit} \rightarrow A! \Sigma/\varepsilon]\) are of shape \( \lambda \star. c \) because \( \star \) is the only value that can be passed as an argument, and therefore has no effect on \( c \).

\[
\lambda \star. c \sim_{\text{unit} \rightarrow A! \Sigma/\varepsilon} \lambda \star. c \iff c \sim_{A! \Sigma/\varepsilon} c
\]

It is straightforward to assert that \((\eta, \lambda \star. c) \in [\varphi]\) holds for all \( c \sim_{A! \Sigma/\varepsilon} c \).

It is straightforward to assert that

\[
\{ \text{let rec } g \_ : \text{unit} \rightarrow C \equiv g () \text{ in } g () \} \eta = \bot.
\]

For clarity, we simplify and name the induction hypotheses, applying Lemma 7.3.5 and Lemma 7.3.6 to weaken contexts and resolve substitutions. The hypotheses for value returns and nontermination are straightforward to simplify, and both assume that \( \eta \sim \eta \) and \( \eta \in [\psi_1 \land \cdots \land \psi_n]\).

\[
(\eta, \lambda \star. \bot) \in [\varphi] \quad \text{ (IH1)}
\]

\[
a \sim_A a \Rightarrow (\eta, \lambda \star. \text{inval}(a)) \in [\varphi] \quad \text{ (IH2)}
\]

The hypotheses for operation calls are denoted by IH3op for each \( \text{op}_A : A_\text{op} \rightarrow B_\text{op} \in \Sigma \).
Assume \( \eta \sim \Gamma \eta \), \( a \sim_A a \), and \( \kappa \sim \text{B}_\text{op}_\text{op} \rightarrow \text{C}_\kappa \), as well as \( \eta \in [\psi_1 \land \cdots \land \psi_n] \). The additional hypothesis is equivalently stated as part of the implication.

\[
(\forall b \sim_{B_\text{op}} b. (\eta, \lambda \star. \kappa(b)) \in [\varphi]) \Rightarrow (\eta, \lambda \star. \text{inval}_\text{op}(a, \kappa)) \in [\varphi] \quad \text{ (IH3op)}
\]

125
We are now faced with a subtle problem. We are unable to use the induction principle for $\sim AΣ/E$ because the hypotheses rely on the fact that elements will be self-related by $\sim$. The predicate contains an implication and is not necessarily admissible. We therefore construct

$$c \mathcal{R} c' \iff (\eta, \lambda \star \cdot c) \in \llbracket \varphi \rrbracket \land (\eta, \lambda \star \cdot c') \in \llbracket \varphi \rrbracket$$

and show that it is closed under the rules for $\sim AΣ/E$. Because $\sim AΣ/E$ is the smallest such relation, $c \sim c$ implies $c \mathcal{R} c$, and it follows that $(\eta, \lambda \star \cdot c) \in \llbracket \varphi \rrbracket$, which is precisely what we need. The part of $\mathcal{R}$ pertaining to $c'$ is included to satisfy symmetry and transitivity.

1. By IH1 we have $(\eta, \lambda \star \cdot \bot) \in \llbracket \varphi \rrbracket$, so $\bot \mathcal{R} \bot$.
2. For $a \sim_A a'$ we have $a \sim_A a$ by symmetry and transitivity of $\sim$. By IH2 it follows that $(\eta, \lambda \star \cdot \text{in}_\text{val}(a)) \in \llbracket \varphi \rrbracket$. The same works for $a'$.
3. Assume $a \sim_{A_{\text{op}}} a'$ and that for all $b \sim_{B_{\text{op}}} b'$ we have $\kappa(b) \mathcal{R} \kappa'(b')$. Then for all $b \sim_{B_{\text{op}}} b$ we have $\kappa(b) \mathcal{R} \kappa'(b)$, and therefore $(\eta, \lambda \star \cdot \kappa(b)) \in \llbracket \varphi \rrbracket$. By IH3$_{\text{op}}$ it follows that $(\eta, \lambda \star \cdot \text{in}_{\text{op}}(a, \kappa))$. The same can be done for $a'$ and $\kappa'$, so we conclude that $(\text{in}_{\text{op}}(a, \kappa)) \mathcal{R} (\text{in}_{\text{op}}(a', \kappa'))$.
4. Requires a longer proof (done below).
5. Follows by symmetry of $\land$.
6. From $c_1 \mathcal{R} c_2$ it follows that $(\eta, \lambda \star \cdot c_1) \in \llbracket \varphi \rrbracket$, and from $c_2 \mathcal{R} c_3$ it follows that $(\eta, \lambda \star \cdot c_3) \in \llbracket \varphi \rrbracket$, therefore $c_1 \mathcal{R} c_3$.
7. If $(c_i)_i$ and $(c'_i)_i$ are chains with $\forall i (c_i) \mathcal{R} c'_i$, then $\forall i. (\eta, c_i) \in \llbracket \varphi \rrbracket$. A premise of INDUCTION is that $f$ is admissible in $\varphi$, so by Lemma 7.3.7 we have $(\eta, \bigvee_i c_i) \in \llbracket \varphi \rrbracket$. We do the same for $c'_i$ and get $(\bigvee_i c_i) \mathcal{R} (\bigvee_i c'_i)$.

If $T_1 \approx T_2$ then $\llbracket T_1 \rrbracket \mathcal{R} \llbracket T_2 \rrbracket$. We now focus on proof of 4. Let $\Gamma; Z \vdash T_1 \approx T_2$ be an equation in $E$, where we use $\Gamma = (x_i : A_i)_i$ and $Z = (z_j : B_j \rightarrow *)_j$. Assume $a_i \sim_{A_i} a'_i$ and that $b \sim_{B_j} b'$ implies $f_j(b) \mathcal{R} f'_j(b')$. We want to prove that

$$(\eta, \lambda \star \cdot \llbracket \Gamma; Z \vdash T_1 : \Sigma \rrbracket^{1+2}(a_i), (f_j)_j) \in \llbracket \varphi \rrbracket,$$

with the proof for $T_2$ proceeding similarly. The proof proceeds by induction on $\Gamma; Z \vdash T_1 : \Sigma$, which is included in the premises for INDUCTION, since types must be well formed. This allows us to use induction hypotheses for values and computations that appear in the template. We mark elements used for contexts as $(a_i)_i = \eta_T$ and $(f_j)_j = \zeta$.

- **WF\text{TAPP},** meaning $T_1 = z_j \mathcal{V} v$. We have the IH for well-typed values that states
  $$\llbracket v \rrbracket_{\eta_T} \sim_{B_j} \llbracket v \rrbracket_{\eta_T}. \text{By assumption for } f_j \text{ it follows that } f_j(\llbracket v \rrbracket_{\eta_T}) \mathcal{R} f'_j(\llbracket v \rrbracket_{\eta_T}),$$
  and this means that $(\eta, \lambda \star \cdot f_j(\llbracket v \rrbracket_{\eta_T})) \in \llbracket \varphi \rrbracket$.

- **WF\text{TDO},** meaning $T_1 = \text{do pure } x \leftarrow c \text{ in } T$. Because $c$ is a pure computation, we only have two options for $\llbracket c \rrbracket_{\eta_T}$.
  - If $\llbracket c \rrbracket_{\eta_T} = \bot$, the denotation of the whole template is $\bot$ and the rest follows from IH1.
  - Assume $\llbracket c \rrbracket_{\eta_T} = \text{inv}_{\text{val}}(a)$. Since $c$ is well typed, we also have the hypothesis $\text{inv}_{\text{val}}(a) \sim \text{inv}_{\text{val}}(a)$; since its type has no equations this can only hold if $a \sim a$. By induction it holds that
    $$(\eta, \lambda \star \cdot \llbracket T \rrbracket^{1+2}(\eta_T, a, \zeta)) \in \llbracket \varphi \rrbracket.$$
– WfTOP: All subtyping is resolved by Lemma 7.2.4, so we omit it. The free interpretation of \( o_{A_{op}} \rightarrow B_{op} (v; y.T) \) is

\[
\text{in}_{op}(\![v]\eta_T, \lambda y. [T]^F_{\mathbb{A}|\mathbb{Z}}(\eta_T, y, \zeta)).
\]

Induction for values gives us \( \![v]\eta_T \sim \![v]\eta_T \). The IH for well-formed templates is \( \lambda b. [T]^F_{\mathbb{A}|\mathbb{Z}}(\eta_T, b, \zeta) \sim \lambda b. [T]^F_{\mathbb{A}|\mathbb{Z}}(\eta_T, b, \zeta) \) and if we assume \( b \sim b \) then by the inner induction on templates we have \( (\eta, \lambda \ast. [T]^F_{\mathbb{A}|\mathbb{Z}}(\eta_T, b, \zeta)) \in [\varphi] \). The rest follows from IH3\(_{op}\).

– Other rules proceed by straightforward structural induction.

We now piece everything together. Assume \( f \sim_{\text{unit} \rightarrow \mathbb{A}_\Sigma/E} f' \). Then \( f = \lambda \ast. c \) for some \( c \), and \( c \sim_{\mathbb{A}_\Sigma/E} c \). Because \( \sim_{\mathbb{A}_\Sigma/E} \) is the smallest relation closed under 1-7, it follows that \( c \mathcal{R} c \). By definition of \( \mathcal{R} \) we have \( (\eta, \lambda \ast. c) \in [\varphi] \), which is \( (\eta, f) \in [\varphi] \).

It follows that \( \eta \in [\forall f. \varphi] \).

\[\square\]

### 7.4 Outline of contextual equivalence

#### Contextual equivalence

A **computation context** \( C \) is a computation that has a number of holes \( \langle \rangle \) (possibly under binders). Inserting the same computation \( c \) into each of the holes of \( C \) results in a computation \( C(c) \). If for all \( \Gamma + c : C \), we have \( \cdot + C(c) : \text{unit}!\{\}/\{\} \), we say that \( C \) is a **ground computation context** for \( \Gamma \) and \( \mathcal{C} \).

We say that computations \( \Gamma + c : C \) and \( \Gamma + c' : C' \) are **contextually equivalent** if for all ground computation contexts \( C \) for \( \Gamma \) and \( \mathcal{C} \), we have that \( C(c) \sim^* \text{ret} \) if and only if \( C(c') \sim^* \text{ret} \). We denote contextually equivalent computations by \( \Gamma + c \equiv_C c' \). The definition of contextually equivalent values \( \Gamma + v \equiv_A v' \) is similar.

Ground contexts suffice for the notion of contextual equivalence. For example, we can differentiate between \text{ret} (\text{Left} (\)) and \text{ret} (\text{Right} (\)) with the ground context

\[
\begin{align*}
\text{do } x \leftarrow \langle \rangle & \text{ in } \\
\text{match } x \text{ with } & \\
\text{| Left } _ \rightarrow \text{ ret } (\) & \\
\text{| Right } _ \rightarrow \text{ loop, }
\end{align*}
\]

where \text{loop} is a nonterminating recursive loop. We can similarly use handlers to differentiate between operation calls.

#### Adequacy

The skeleton denotations of \( \text{EEFF} \) coincide with the denotations of \( \text{Eff} \) [6], for which adequacy is shown. Having adequacy on skeletons directly carries over to adequacy on term semantics.

**Lemma 7.4.1.** If \( \cdot + c : \text{unit}!\{\}/\{\} = \text{in}_{\text{val}}(\ast) \), then \( c \sim^* \text{ret} (\) .

**Proof.** If \( \cdot + c : \text{unit}!\{\}/\{\} = \text{in}_{\text{val}}(\ast) \), then \( \langle \cdot + c \ast : \text{unit} \rangle = \text{in}_{\text{val}}(\ast) \) by Lemma 6.4.4. By adequacy for skeleton semantics [6, Corollary 5.8], we have \( c \sim^* \text{ret} (\) . \( \square \)
Corollary 7.4.2.
- If \([\Gamma \vdash c : C] = [\Gamma \vdash c' : C]\), then \(\Gamma \vdash c \equiv_C c'\).
- If \([\Gamma \vdash v : A] = [\Gamma \vdash v' : A]\), then \(\Gamma \vdash v \equiv_A v'\).

Proof. Assume that \(\mathcal{C}(c) \sim^* \text{ret}()\). By soundness of denotational semantics (Theorem 6.5.3) we have \([\mathcal{C}(c)] = \text{in}_{\text{val}}(\star)\). Denotational semantics is structural, and therefore \([\mathcal{C}(c)] = [\mathcal{C}(c')]\). By Lemma 7.4.1 it follows that \(\mathcal{C}(c') \sim^* \text{ret}()\). The proof for values is similar. □

Logic equality implies contextual equivalence

Lemma 7.4.3 (Adequacy). If \(\cdot \vdash c : \text{unit!}()\) \(\sim\) \(\text{in}_{\text{val}}(\star)\) then \(c \sim^* \text{ret}()\).

Proof. The relation \(\sim_{\text{unit!}()}\) is an identity relation because the type has no operations or equations. It follows that \([c] = \text{in}_{\text{val}}(\star)\), and we use Lemma 7.4.1. □

Theorem 7.4.4. Assume that the type system is coupled with a sound logic.
- If \([\Gamma \vdash v : A] \eta \sim_A [\Gamma \vdash v' : A] \eta'\) holds for any \(\eta \sim \eta'\), then \(\Gamma \vdash v \equiv_A v'\).
- If \([\Gamma \vdash c : C] \eta \sim_C [\Gamma \vdash c' : C] \eta'\) holds for any \(\eta \sim \eta'\), then \(\Gamma \vdash c \equiv_C c'\).

Proof. Assume \(\mathcal{C}(c) \sim^* \text{ret}()\) for a ground context \(\mathcal{C}\). By Theorem 6.5.3 we have \([\mathcal{C}(c)] = \text{in}_{\text{val}}(\star)\). Using a proof similar to the proof for Theorem 7.3.2, we show that \([\mathcal{C}(c)] \sim [\mathcal{C}(c')]\) because \([c] \eta \sim [c'] \eta'\) for any \(\eta \sim \eta'\). Therefore, \([\mathcal{C}(c')] \sim \text{in}_{\text{val}}(\star)\), and we conclude by applying Lemma 7.4.3. The proof for values is identical. □

Corollary 7.4.5. Assume that we use the equational logic of Section 5.4.
- If \(\Gamma \vdash v \equiv_A A\), then \(\Gamma \vdash v \equiv_A v'\).
- If \(\Gamma \vdash c \equiv_C C\), then \(\Gamma \vdash c \equiv_C c'\).

Proof. Follows from Theorem 7.4.4 with the soundness of equational logic, shown in Theorem 7.3.4. □

Corollary 7.4.6. Assume that we use the predicate logic of Section 5.5.
- If \(\Gamma \vdash v \equiv_A v'\), then \(\Gamma \vdash v \equiv_A v'\).
- If \(\Gamma \vdash c \equiv_C c'\), then \(\Gamma \vdash c \equiv_C c'\).

Proof. Follows from Theorem 7.4.4 with the soundness of predicate logic, shown in Theorem 7.3.8. □
Chapter 8

Implementation and formalisation

8.1 Bidirectional type inference

The type system in Chapter 4 serves well as an analytical tool, but type derivations cannot be constructed algorithmically. The construction needs to be done by the user, and handcrafting typing derivations is not a feasible way of programming. It is thus crucial to provide a type-inference algorithm for \( EEFF \), built in a way that simplifies the use of equations.

In Eff, type inference is done via a modified Damas–Hindley–Milner inference algorithm, but we consider the bidirectional type-inference algorithm a better fit for \( EEFF \). Firstly, there is no sensible way of inferring effect theories, so type annotations are a necessity, playing into the strengths of bidirectional inference. Secondly, since the language features equations, local signatures, and subtyping, there is an abundance of opportunities for obscure errors in the unification step of Damas–Hindley–Milner.

8.1.1 Changes in syntax

Bidirectional type systems require a type annotation construct \((v : A)\) as part of the syntax. This allows us to simplify the rest of the terms by removing annotations specific to certain term constructors. Recursive functions keep the old annotations, as the new construct cannot replace them.

```
values v ::=  
  \((v : A) | x () | n | \text{fun x \mapsto c | handler (ret x \mapsto c_r; h)} | (v_1, v_2) | \text{Left v | Right v | [ ] | v_1 :: v_2}  

computations v ::=  
  \((c : C) | \text{ret v | do x \mapsto c_1 in c_2 | v_1 v_2} | \text{let rec f x : A \rightarrow C = c_1 in c_2} | \text{op(v; y.c) | with v handle c | absurd v | match v with (x,y) \mapsto c} | (\text{match v with Left x \mapsto c_1 | Right y \mapsto c_2}) | (\text{match v with [ ] \mapsto c_1 | x :: xs \mapsto c_2)  

operation cases h ::=  
  \{ \} | h \cup \{ \text{op(x; k) \mapsto c_op} \}
```
We similarly remove the type annotations for operation calls in the template language, which is otherwise unchanged.

This introduces a second syntax for EEFF, so we refer to the above syntax as the bidirectional syntax and to the syntax of EEFF from Chapter 3 as the declarative syntax.

### 8.1.2 Algorithm

We base our bidirectional type-inference algorithm on the ideas of Dunfield and Krishnaswami [14, 15]. Handler correctness is undecidable [37], so we work in the full logic of Section 5.3. In this setting all handlers respect all equations, so proofs of correctness are trivial. This comes at a cost of logic soundness, but the user has the option to do proofs by hand in a different logic.

Bidirectional inference is based on two modes of inference, checking whether a term has the given type, and synthesising the type of a term.

- **type checking** \( \Gamma \vdash v \iff A \), where both \( v \) and \( A \) are input.
- **type synthesis** \( \Gamma \vdash v \Rightarrow A \), where \( v \) is provided and \( A \) is the output.

While checking is easier, it requires a type at which to check. Synthesis is therefore preferred, as it requires no input type, but is not always an option. For instance, we can synthesise the type of \( \text{true}, 1 \) to be \( \text{bool} \times \text{int} \) because we can synthesise the types of components. In the case of \( \text{Left false} \) full synthesis is not possible, as all we know is that it has a type of form \( \text{bool}+???. \) We are missing a crucial piece of information to fully assign it a type. Even in the declarative syntax of EEFF, we rely on type annotations for sums, functions, handlers, etc. Type annotations serve as a way to provide the type for type checking in cases where synthesis is not possible.

For the sake of simplicity we provide one rule per term constructor, with additional rules for switching between modes. Synthesis judgements are the preferred option, but we try to use checking in the premises of rules whenever possible.

\[
\begin{align*}
(x : A) \in \Gamma & \quad \text{S synthVar} \\
\Gamma \vdash x \Rightarrow A & \\
\Gamma \vdash () \Rightarrow \text{unit} & \\
\Gamma \vdash n \Rightarrow \text{int} & \\
\Gamma \vdash v_1 \Rightarrow A & \quad \Gamma \vdash v_2 \Rightarrow B \\
\Gamma \vdash (v_1, v_2) \Rightarrow A \times B & \\
\Gamma \vdash v \Rightarrow A & \quad \Gamma \vdash vs \Rightarrow A \text{ list} \\
\Gamma \vdash v :: vs \Rightarrow A \text{ list} & \\
\Gamma \vdash v \iff A & \\
\Gamma \vdash \text{Left } v \iff A + B & \quad \text{checkLeft} \\
\Gamma \vdash \text{Right } v \iff A + B & \quad \text{checkRight} \\
\Gamma \vdash [] \iff A \text{ list} & \quad \text{checkNil} \\
\Gamma \vdash \text{fun } x \mapsto c \iff A \rightarrow C & \quad \text{checkFun} \\
\Gamma, x : A \vdash c_r \iff D & \\
\Gamma \vdash \text{handler } (\text{ret } x \mapsto c_r; h) \iff A!\Sigma/\mathcal{E} \Rightarrow D & \quad \text{checkHandler}
\end{align*}
\]

With branching computations, we stick to the “no guessing” policy and use the check mode. In practice, this seems to be a minor nuisance, as most computations tend to be in
positions that require checking.

\[
\begin{align*}
\Gamma \vdash v &\iff A & \text{SynthRet} \\
\Gamma \vdash \text{ret} v &\iff A!\{\} / \{\} & \text{SynthApp} \\
\Gamma \vdash v_1 &\iff A \rightarrow C & \Gamma \vdash v_2 &\iff A & \text{SynthLetRec} \\
\Gamma \vdash f x : A &\rightarrow C = c_1 \iff C \\
\Gamma \vdash f : A &\rightarrow C \iff D \\
\Gamma \vdash \text{let rec } f x : A &\rightarrow C = c_1 \in c_2 \iff D \\
\Gamma \vdash v &\iff C \iff D & \Gamma \vdash c \iff C & \text{SynthHandle} \\
\Gamma \vdash v &\iff A \times B & \Gamma \vdash x : A, y : B + c &\iff C & \text{SynthProdMatch} \\
\Gamma \vdash \text{match } v \text{ with } (x, y) &\mapsto c \iff C \\
\Gamma \vdash v &\iff \text{empty} & \Gamma \vdash \text{absurd } v &\iff C & \text{CheckAbsurd} \\
\Gamma \vdash v &\iff A + B & \Gamma \vdash x : A \iff c_1 \iff C & \Gamma \vdash y : B + c_2 \iff C & \text{CheckSumMatch} \\
\Gamma \vdash \text{match } v \text{ with Left } x &\mapsto c_1 \mid \text{Right } y &\mapsto c_2 \iff C \\
\Gamma \vdash v &\iff A \text{ list} & \Gamma \vdash c_1 \iff C & \Gamma \vdash x : A, xs : A \text{ list} + c_2 \iff C & \text{CheckListMatch} \\
\Gamma \vdash \text{match } v \text{ with } [ ] &\mapsto c_1 \mid x :: xs &\mapsto c_2 \iff C \\
\Gamma \vdash v &\iff op : A_{op} \rightarrow B_{op} \in \Sigma & \Gamma \vdash v \iff A_{op} & \Gamma \vdash y : B_{op} + c \iff A!\Sigma/E & \text{CheckOp} \\
\Gamma \vdash op(v; y).c \iff A!\Sigma/E \\
\Gamma \vdash c_1 \iff A!\Sigma/E & \Gamma \vdash x : A + c_2 \iff B!\Sigma/E & \text{CheckDo} \\
\Gamma \vdash \text{do } x &\leftarrow c_1 \text{ in } c_2 \iff B!\Sigma/E \\
\Gamma \vdash [ ] &\iff \{\} \Rightarrow D & \text{TypeCases[]} \\
\Gamma \vdash h \iff \Sigma \Rightarrow D & \Gamma \vdash x : A_{op}, k : B_{op} &\rightarrow D \vdash c_{op} \iff D & \text{TypeCasesU} \\
\Gamma \vdash h \cup \{op(x; k) \mapsto c_{op}\} \iff (\Sigma \cup \{op : A_{op} \rightarrow B_{op}\}) \Rightarrow D \\
\end{align*}
\]

The operation cases are always in a checking position.

\[
\begin{align*}
\Gamma \vdash [ ] &\iff \{\} \Rightarrow D \\
\Gamma \vdash h \iff \Sigma \Rightarrow D & \Gamma \vdash x : A_{op}, k : B_{op} &\rightarrow D \vdash c_{op} \iff D & \text{TypeCasesU} \\
\end{align*}
\]

Having a rule for every construct is not sufficient. Whenever we require synthesis for a term that only has a check rule, we use type annotations. For the case, where we need to check terms with a synthesis rule, we introduce a second mode-switch rule, which also introduces subtyping into the system.

\[
\begin{align*}
\Gamma \vdash v &\iff A & \text{SynthAnnV} & \Gamma \vdash v \iff A \quad A \leq A' & \text{CheckVBysynth} \\
\Gamma \vdash (v : A) &\Rightarrow A \\
\Gamma \vdash c &\iff C & \text{SynthAnnC} & \Gamma \vdash c \iff C \quad C \leq C' & \text{CheckCBysynth} \\
\Gamma \vdash (c : C) &\Rightarrow C \\
\end{align*}
\]

The bidirectional type system also features judgements for well-formedness. There are no changes for types and contexts, while templates are treated as check-only. Templates are
always part of a type that also contains the signature at which to check templates, so we see no reason to provide synthesis.

\[
\Gamma \vdash v \iff A \quad (z : A \rightarrow *) \in Z
\]

always part of a type that also contains the signature at which to check templates, so we see no reason to provide synthesis.

\[
\Gamma \vdash c \Rightarrow A!(\{\})/\{}
\]

\[
\Gamma; Z \vdash \text{pure } x \leftarrow c \text{ in } T \iff \Sigma
\]

\[
\Gamma; Z \vdash \text{match } v \text{ with } (x, y) \mapsto T \iff \Sigma
\]

\[
\Gamma; Z \vdash \text{match } v \text{ with } [[ \] ] \mapsto T \mid x :: xs \mapsto T_2 \iff \Sigma
\]

\[
\Gamma \vdash \text{vtype } \text{in the bidirectional type system, then } \vdash \text{vtype} \text{ in the declarational type system. A similar property holds for computation types, signatures, equations, contexts, and template contexts.}
\]

\[
\text{Proposition 8.1.1. Assume that we are using the annotation-free version of the declarative type system from Chapter 4.}
\]

\[
\text{⋄ If } \Gamma \vdash v \Rightarrow A, \text{ then } \Gamma^e \vdash v^e : A^e. \text{ A similar property holds for synthesizing types of computations.}
\]

\[
\text{⋄ If } \Gamma \vdash v \iff A, \text{ then } \Gamma^e \vdash v^e : A^e. \text{ A similar property holds for checking types of computations and operation cases.}
\]

\[
\text{⋄ If } \vdash A : \text{vtype in the bidirectional type system, then } \vdash A^e : \text{vtype} \text{ in the declarational type system. A similar property holds for computation types, signatures, equations, contexts, and template contexts.}
\]

\[
\text{⋄ If } \Gamma; Z \vdash T \iff \Sigma, \text{ then } \Gamma^e ; Z^e \vdash T^e : \Sigma^e.
\]

\[
\text{Proof (formalised). The proposition requires a simultaneous proof for 12 different statements. Otherwise, it proceeds by induction on the mutually recursive definition of type synthesis, type checking, type well-formedness, and template well-formedness.}
\]
We consider this sufficient proof that the bidirectional type inference works as intended. We conjecture there are possible annotation procedures that can be used for relating the bidirectional variant of \textit{EEFF} to the annotated version from Chapter 4. Such conjectures are left for future work, as constructing terms from typing-derivation proofs is difficult in the current formalisation.

\section{Implementation}

The implementation\footnote{https://github.com/zigaLuksic/eff/tree/EEFF} of \textit{EEFF} is built upon a stripped-down framework of Eff, which is implemented in OCaml (version 4.06). The changes to syntax are kept minimal, and the modules for evaluation are nearly untouched. The main change is the bidirectional type inference, replacing the previously used Damas–Hindley–Milner type inference.

By changing the type system, certain syntactic sugar no longer interacts well with the type system. We simplified handlers by removing the \texttt{finally} clause and reduced the amount of pattern matching. The other significant change is the move from \texttt{perform Op} to \texttt{!Op} for effect invocations. We also switch back to the \texttt{val} notation for value cases of a handler.

\subsection{Improvements to the language}

To achieve a more intriguing prototype language for local theories, the implementation features some extensions that are not formalised.

The \texttt{match} statements and functions are improved through the use of pattern matching. Since we saw no difficulties in the treatment of lists in \textit{EEFF}, we feel confident that recursive data types do not break the approach and allow user-defined variant types in the implementation. We also include additional base values, such as strings or floats, and a number of predefined functions, such as integer multiplication or string concatenation.

\textbf{Operations, equations, and theories}

Local signatures are useful from a theoretical standpoint, but having to write types of operations in every annotation is too cumbersome. The implementation allows defining global annotations, which are taken as the default if no custom annotation was provided. This allows us to shorten \texttt{int!\{Print : string \rightarrow unit\}/\{} to just \texttt{int!\{Print\}/\{} if we fix the type with

\begin{verbatim}
  effect Print : string \rightarrow unit
\end{verbatim}

Throughout the thesis we treated signatures and equations as two different entities, which they, in some sense, are. But when programming, we never need equations without a signature. We therefore switch to defining theories that combine signatures and effects into a single construct.

\begin{verbatim}
  theory eqn_comm for \{Choice : int \times int \rightarrow int\} is
    \{ x:int, y:int ; z:int \rightarrow \_ \_ \_ Choice((x,y); w.z w) \_ Choice((y,x); w.z w) \_ \}
\end{verbatim}

This also affects how the types are displayed.

\begin{itemize}
  \item Effects with no theory are written as \texttt{int!\{(Choice : int \times int \rightarrow int)\}}.
  \item No specified theory and a global type assignment results in \texttt{int!\{Choice\}}.
  \item When using a theory, the type is displayed as \texttt{int!eqn_comm}.
\end{itemize}
Theory definitions are also useful for large sets of effects, since we can define a theory with no equations and use it as a shorter notation for the signature. Theories can also be built by combining multiple theories or extending a previous theory with new equations.

```ocaml
theory getget for {Get} is
  { . ; z : state * state -> * |- 
    Get(() ; y. Get(() ;w. z (y, w))) ~ Get(() ; y. z (y, y) ) }

theory setget for {Get , Set } is
  { x : state ; z : state -> * |- 
    Set(x ; y.Get(() ;w. z w)) ~ Set(x ; y. z x) }

theory state_theory for {Get , Set } is getget and setget
```

To reduce overloading of notation, we decided that each equation is written as a singleton. This may seem odd, but it results in much more readable code. A theory then consists of a collection of equations or previously defined theories, separated by and. Inheriting equations from theories uses subtyping to ensure safety. In the above example, the first theory is constructed for Get only, but by Lemma 4.3.4 it follows that it can be safely included in a theory for a larger signature.

**Primitive effects**

Primitive effects allow actual effectful behaviour, such as printing or random number generation. This requires us to break the abstraction barrier, but we try to do so in a manner that suits the general narrative. Primitive effects are treated like any other operation call; they can even be intercepted by user-defined handlers and then handled away. The difference is that if evaluation results in a regular operation call, we consider it an uncaught exception and warn the user. But if the program results in a primitive operation call, we use built-in OCaml effects to achieve the desired behaviour. This can be thought of as an outer handler that is wrapped around the entire program at all times. The correctness of the outer handler cannot be established within the same logic framework, and it currently allows any theory. This might be improved upon in future versions, after we establish suitable theories for the implemented primitive effects.

### 8.2.2 Adjusting type inference

The bidirectional type inference from Section 8.1 is, in general, sufficient, but we tailor it further to meet our needs. The goal is to have a type system that requires little user help and provides the user with precise information in the case of a type error.

One of the major concerns of using a bidirectional type system is the need for annotations, but in practice, annotations are required mostly when defining functions and handlers. This is showcased by the following examples, which showcase extensive use of functions, handlers, and operation calls. Providing types at the time of definition seems to be only a minor burden and, at the same time, a way towards a safer programming practice.
let state_handler
  : int!state_theory => state -> (int * state)
  = handler
  | effect Get () k -> (fun s -> k s s)
  | effect Set s k -> (fun _ -> k (s))
  | val x -> (fun s -> (x, s))

let test =
  with state_handler handle
  !Set(10 + !Get()); !Get () + 2

type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
effect Yield : int -> unit

let rec (tree_yield : int tree -> unit !{Yield}) = function
  | Empty -> ()
  | Node (lt , x, rt) -> tree_yield lt; ! Yield x; tree_yield rt

let yield_sum : unit !{Yield} => int = handler
  | effect Yield x k -> x + k ()
  | val () -> 0

let leaf : int -> int tree = fun x -> Node (Empty , x, Empty)

let test =
  with yield_sum handle
  tree_yield (Node (leaf 1, 5, Node (leaf 4, 2, leaf 0)))

Increasing the amount of rules

Using the minimal amount of rules for a bidirectional system is great from a theoretical standpoint, but additional rules improve the user experience. The mode switch is meant to correct the issue of certain terms not having a checking rule, but it is sometimes insufficient.

( (0 , fun x -> x) : int * (int -> int) )

Pairs only have a synthesis rule, so the above example forces a mode switch from checking to synthesis. Because function types cannot be synthesized, the typing fails, in spite of the outer annotation. Synthesis is useful because it requires no annotations, but once annotations have been provided, we should not be wasteful by prematurely switching back to synthesis. For that reason, the implementation has checking rules for some terms that already have a synthesis rule. We also provide synthesis rules for operation calls with global annotations.

Additional rules also improve the error reporting of the type system. Bidirectional type systems already feature more precise errors compared to Damas–Hindley–Milner type inference, and by tailoring the rules further, we have more information available at the point of failure, resulting in clearer error messages.
Adapting to patterns

Pattern matching is a luxury that people are reluctant to abandon, and adapting the inference algorithm is not too difficult. Patterns should always be checked, and checking patterns needs to return the bindings of variables to types. The example below should return \( x: \text{int}, y: \text{bool} \), needed to check the function body.

```plaintext
let (f : int * bool -> int) = fun (x, y) -> x
```

Whenever a variable is encountered, we return a binding, while other patterns check sub-patterns and combine their bindings. Duplication of variables is forbidden in a pattern, so there is no issue with shadowing.

Issues with desugaring

Perhaps the most curious problem of implementing the bidirectional inference for \textit{EEFF} comes from the desugaring of call-by-value into fine-grained call-by-value (briefly described in Section 3.4). Recall the rule for \texttt{do} sequencing (represented by \texttt{let} in the implementation).

\[
\begin{align*}
\Gamma \vdash c_1 \Rightarrow A!\Sigma/E & \quad \Gamma, x : A \vdash c_2 \Leftarrow B!\Sigma/E \\
\Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 \Leftarrow B!\Sigma/E
\end{align*}
\]

If there are problems with the synthesis for \( c_1 \), the user can simply add annotations. But during the desugaring step, we automatically insert \texttt{let} (sugar for \texttt{do}) sequencing whenever needed, so the user cannot annotate them properly. It is far too cumbersome to expect a fine-grained CBV style of code, so we need an adjustment of the type system.

The first step is to allow synthesis of \texttt{let} statements. This is done so that simple computations don’t require type annotations.

```plaintext
let three = 1 + (1 + 1)
(* desugars into *)
let three = (let x = 1 + 1 in 1 + x)
```

Synthesis of theories is not desirable, so we simply restrict synthesis of \texttt{let} statements to pure computations, which covers the majority of cases. If effects occur, we want the user to be explicit about them, and therefore annotate accordingly. However, the rule for checking a \texttt{let} statement includes synthesis of \( c_1 \).

```plaintext
let test : int!state_theory = (1 + !Get ()) + 2
(* desugars into *)
let test : int!state_theory =
  let x =
    let y = !Get () in
    1 + y
  in
  x + 2
```

In the above example we have to synthesize the type of a computation that is not pure. We consider the problem specific to effect systems because the effects are clearly set through annotations, but the synthesis is unable to utilize such information. One solution would be to require users to adopt a more explicit coding style for using effects, but that goes against the goal of delivering a natural coding style. We therefore utilize a third mode of inference, which infers the value type but checks the signature. The effect information allows us to synthesise the types of operation calls without globally assigned types. The
mode only propagates to nested `let` constructs and switches to regular synthesis as soon as it encounters a construct that is not sequencing or an operation call. We consider this a temporary solution and are searching for a more principled approach, though it is unclear whether the problem lies in type inference or desugaring.

### 8.3 Formalisation

With the exception of Chapters 6 and 7, all the proofs of this thesis have been fully formalised\(^2\), alongside the majority of the examples encountered throughout the thesis. We consider this a crucial step in ensuring strong foundations for local effect theories.

The formalisation is written in the Coq proof assistant (version 8.6) and split into three branches:

- `EEFF` using the equational logic of Section 5.4.
- `EEFF` using the predicate logic of Section 5.5. This branch also includes the proofs for skeletons used in Chapter 6.
- The bidirectional type system of Section 8.1.

The logic is too heavily coupled with the type system to allow for an easy abstraction—hence the separation into three distinct branches.

The biggest difference between the formalisation and the material presented in this thesis is the representation of variables. In the thesis we use named variables as we consider them easier to read. Since named variables are not well suited for formalisations, we instead use de Bruijn indices [32]. For readers interested in the formalisation code, we strongly suggest they firstly familiarise themselves with de Bruijn indices.

When defining signatures, equations, and operation cases, we use the mathematical notation for sets. Sets are rather difficult to work with, and we instead use list structures. The rules of the type system are written in a way that favours lists, so we require no adjustments. The only benefit of sets over lists is uniqueness of elements (which we ensure through well-formed signatures) and ignoring the order of elements. The order of elements is not an issue thanks to subtyping, which is done in a way that allows reordering of signatures and equations. This is also another reason for the addition of explicit subtyping in the \(\equiv_{D}^{\Sigma} \equiv \Sigma \) relation.

In larger proofs where mutually recursive induction on several relations is required, we have observed possible performance issues of Coq’s termination checker because of many mutually defined relations. Proofs are constructed through `Fixpoint`, so induction is obtained by recursive applications of lemmas. The termination checker needs to ensure that recursive calls are used on suitable sub-derivations, and to avoid enormous compilation times, we adapt a style that reduces the workload of the termination checker. After starting the case analysis, we immediately construct induction hypotheses and remove the option of recursive lemma applications. This increases the amount of code and sometimes requires some amount of a reverse-engineering effort, but ultimately brings compilation time from hours down to seconds.

#### 8.3.1 Formalising substitution

Substitution is often taken lightly when analysing a programming language. The formalisation of `EEFF` with the predicate logic features over 11000 lines of code (not including examples), of which 7000 are dedicated to substitution. While formalising substitution

\(^2\)https://github.com/zigaLuksic/eeff-formalization
represents a considerable amount of work, it is also the part in which we detected most issues with the language.

In the formalisation we use two notions of substitution: the substitution of a single variable and the parallel substitution (called instantiation in the code). We use two substitutions because there are clear cases for the use of both, and emulating one with the other quickly leads to problems.

When proving Lemma 5.4.9 about the type-safety of substitution in the logic, the difficult cases are rules that contain substitution—for instance

\[
\Gamma_1, p : A \times B, \Gamma_2 \vdash c : C \\
\Gamma_1, \Gamma_2 \vdash c[p \mapsto v] \equiv_C \text{match } v \text{ with } (x, y) \mapsto c[p \mapsto (x, y)] \quad \eta_{\text{PAIR}}
\]

In the proof we are faced with the term \(c[p \mapsto v][x \mapsto v']\) but the induction hypotheses are formed for \(c[x \mapsto v'][p \mapsto v]\). Switching the order of substitutions is not simple and becomes even harder in the case where a parallel substitution meets single variable substitution. We therefore strongly advise against extending EEFF without a thorough proof of the substitution lemma, despite the required effort.

### 8.3.2 Formalisation as a reasoning tool

The formalisation includes all the judgements of the logic, which allows one to use it as a tool for constructing proofs of handler correctness (or other properties). Currently a lot of time is spent on dispatching proofs for side conditions, such as well-formedness of types or typing different terms, but it is nonetheless a useful tool. Manually calculating \(T^h\) for more advanced equations is easy to get wrong, so we used the formalisation to double-check examples. While the proofs are currently cumbersome, they are not difficult, so the reasoning logic of EEFF could become useful with improved tooling. Even using simple custom tactics results in a large reduction of proof-code lines needed for examples.
Chapter 9

Conclusion

Local algebraic effect theories offer the improved reasoning capabilities of equational theories without imposing global restrictions on handlers. The type system is naturally upgraded to track the use of different algebraic theories pertaining to the locally occurring effects. This enables the use of different theories in separate parts of the program, and it even allows for different nested theories through the use of handlers as theory transformers.

The language $EEFF$ features plenty of common extensions, such as pairs, sums, lists, recursion, and subtyping. The terms of the language closely mirror those of Eff [6]; in fact all, Eff programs (sans polymorphism) can be run in $EEFF$ by assuming trivial effect theories.

Theories are specified through templates, a rich subset of the core language, which are expressive enough to state common effect theories, such as mutable state or nondeterminism. Templates are included in types, but can contain terms, so a careful treatment is required to avoid possible circularities. To ensure that handlers respect effect theories, the type system is coupled with a logic in which proofs of correctness are constructed. We presented multiple suitable logics in Chapter 5.

The choice of logic impacts on the strength of the theory system, with possible logics ranging from the empty logic up to a predicate logic with induction. By using denotational semantics, we were able to pinpoint requirements for the soundness of a logic, where sound logics result in expected behaviour of denotations. The type system relies on the logic only for handler correctness, so it is fairly straightforward to integrate an arbitrary logic (though proving soundness might prove difficult).

The denotational semantics is given in two stages. In the first stage, we devise denotations of types and terms, where equations are left out. This is improved upon in the second stage, where equations are used to construct partial equivalence relations. Unsatisfactory elements are left out of the relation, and we show that by using a sound logic, all well-typed programs are correct with regard to local theories. We further show that the suggested logics are sound, establishing that logic proofs of handler correctness are sufficient.

The language $EEFF$ has been fully formalised in the Coq proof assistant. The formalisation serves as proof that definitions and theorems contain no hidden circularities. It includes two effect-theory systems that use different logics, doubling as a reasoning toolkit for $EEFF$. Although the formalisation of each logic only contains the raw essentials, proofs of examples are merely cumbersome—not difficult. Through better automation of typing derivations, most proofs would require only slightly more work than by pen and paper.

We also provide an implementation of $EEFF$, complete with bidirectional type inference. The focus of the implementation is user-friendly effect-theory tracking. To reduce the user burden, we use an unsound logic that does not check handler correctness. This simpli-
fication can easily be mitigated by delegating proofs of correctness to the user, who can then use the formalisation to provide said proofs. The implementation checks correctness of theory propagation, theory subtyping, and safety of handler application.

9.1 Comparison to related work

Original approach

The original approach to effect handlers [38, 37] heralded many of the ideas that are used and improved upon in EEFF. Effects are equipped with equations, but theories are global, locking each effect to a single theory. The correctness of handlers is verified by a logic system featuring computational induction and logic inequalities. The approach features a template language, which suffices for a wide variety of equations but only allows the use of a select few primitive functions; this was improved in EEFF with the do pure template. The denotational semantics accounts for theories, but the treatment of handlers that are not correct is rather unsatisfactory. In our approach, the issue is avoided by splitting the semantics into two steps, where all handlers receive a denotation as terms, but only correct handlers are part of the PER treatment.

Effects without equations

The work of Bauer and Pretnar [6] proceeds in a setting with no effect theories. It provides powerful reasoning tools based around the denotational semantics of the language. Equalities include rules, such as β-reductions, η-expansions, and an induction principle. This suffices for multiple significant examples—for instance, validating equations of state under the state handler, albeit in a slightly altered form. The approach lacks a way to abstract away from handler implementations and only has descriptive properties.

This line of approach has been further extended by Biernacki et al. [8] in a language that includes row polymorphism and lift expressions. At the core of reasoning is a logical relation for the denotational semantics. Biernacki et al. construct specialised compatibility lemmas for reasoning in presence of row polymorphism and provide several examples of program equivalences. Similar to the work of Bauer and Pretnar [6], reasoning remains specific to implementations of handlers.

Defined algebraic operations

The use of defined algebraic operations (DAO) [18] instead of conventional handlers solves certain issues with operation names. Aside from mechanisms for binding operation names, the differences between DAO and handlers are minor, making the ideas all the more relevant. Unlike approaches based on denotational semantics, the focus of reasoning with DAO lies in a theory-dependent logic. The system is built from logic rules and allows the use of incrementally stronger logic systems within proofs. Using theories in the logic also allows for the separation of reasoning about code from verifying that implementations validate the theory. The denotational semantics uses PERs to confirm the soundness of the approach, similar to our work in Chapter 7. While theories are used in the logic system, they are not part of the type system, so the approach remains descriptive as opposed to the prescriptive nature of EEFF.
Dependent types

Effect theories may also be studied in a dependently typed setting [2, 3] via a generalisation of algebraic effects and handlers to dependent types. To avoid unsound program equivalences, it employs user-defined algebra types, which carry proofs that handlers satisfy equations of the effect theory, similar to the respects relation of EEFF. The approach provides a wide variety of reasoning techniques. Through the use of equations proofs can be abstracted away from a concrete handler implementation. While it alleviates certain issues of the original approach, this approach remains in the realm of global effect theories.

Reasoning about effect trees

By forgoing handlers and focusing solely on algebraic effects, one can construct specialised tools in the form of modalities and behaviour equivalences [43, 47]. Instead of fixing the theory through equations, the behaviour of effects is fixed by a choice of modalities, which can be more tailored to specific effects. Modalities work on effect trees and can express properties such as “every possible execution terminates”, which is beyond the current reasoning capabilities of EEFF. A considerable benefit of the approach is the additional capability of showing non-equality of programs by providing a logic property that differentiates them. The authors consider the logic difficult to apply in some cases, but suitable as a low-level language to translate into.

Similar approaches with monads

Algebraic effect theories can also be used in conjunction with monads, as showcased by the work of Gibbons and Hinze [20, 19], which has had a large influence on the style of reasoning we want to achieve in EEFF. Imposing requirements on monad implementations results in additional reasoning techniques that apply on a syntactic level, as opposed to the semantic reasoning techniques [6, 8]. Monad requirements are the equivalent of handler correctness, which can be clearly seen by direct comparison of the state monad [19, Example 2.4] and the state handler (Example 5.4.13).

\[
\text{set } s \gg \text{get} \gg \lambda s' \rightarrow k s' = \text{set } s \gg k s \quad \text{requirement for MonadState}
\]

\[
\text{Set}(s; \_ \text{.Get}((); s'.z s')) \sim \text{Set}(s; \_z s) \quad \text{equation of the state handler}
\]

Effect behaviour is abstracted away from concrete monad implementations, providing better reusability of proofs. The authors provide a large number of examples, ranging from the mutable state theory to solving the Monty Hall problem. While the approach is prescriptive in nature, it receives no aid from the type system.

Automated approaches

Project F* [4, 29] is based around verifying effectful code. The mechanism driving the verification are Dijkstra monads, and although early work [4] uses a fixed set of effects, it has since been extended to encompass user-defined algebraic effects [29]. It includes dependent types, effects, refinement types, and an SMT solver to assist in writing proofs. The inclusion of effect handlers is slightly unsatisfactory in their general setting, relying on external proof of handler correctness.
9.2 Future directions

Exploring equations

While there are many possibilities for improving EEFF, it would perhaps be more fruitful to first explore possible use cases. The presented system features a lightweight logic that is versatile and easy-to-use, which would be a wasted opportunity if there are no suitable problems to solve. Example 5.5.12 raises hope that there are opportunities outside of neat, artificially built examples. By exploring use cases, one obtains a better understanding of the shortcomings, which ideally guides future development.

The first step is to thoroughly examine the expressivity of equations. In Example 5.4.13 we have shown how to work with the theory of state, which features simple equations. The equation of Example 5.5.12 is a conditional equation, which describes behaviour under certain conditions.

\[
\text{match } \text{Next}() \text{ with}
\begin{cases}
\text{Some } x \mapsto z(\text{Next}()) \\
\text{None } \mapsto z(\text{None})
\end{cases}
\sim \text{Next}(); z(\text{Next}())
\]

Can we take equations even further? The do pure construct allows for a variety of new possibilities; for instance, equations that specify properties about operation output. Suppose that we use an operation 

\[
\text{OrderedRandList}((); \text{lst}. \text{do pure } b \leftarrow \text{is}_\text{ordered} \text{lst in}) \sim \text{OrderedRandList}((); \text{lst}. z \text{ true})
\]

There are bound to be other innovative uses for equations, and we consider it worthwhile to explore them.

On the other hand, we wish to use the system for larger and more meaningful examples. There has been work on effect-dependent optimisations [24], which could benefit from local theories. Optimisations can be performed locally with the type system guaranteeing their safety. Since theories are user-defined, there is also a possibility of user-defined optimisations, though it seems far off. Another application might be fields using algebraic effects that are heavily reliant on effect guarantees. Effect theories for probabilistic choice could help with probabilistic programming [42, 46, 9].

Polymorphism

Effect systems report which effects may occur, and without a way of weakening effect information, the resulting language is greatly hampered. The extensions most commonly used to alleviate the issues are subtyping and polymorphism. We first explored the option of subtyping as we already had an idea for the notion of subtyping on theories, but we feel that both subtyping and polymorphism need to be studied in the setting of local theories. One of the most important future directions is certainly the adaptation of EEFF to polymorphism, with recent work also showing that bidirectional type inference interacts well with higher-rank polymorphism [15]. The first step is certainly devising a correct notion of polymorphism for theories. The natural approach is to consider polymorphism in all components of the computation type; for instance,

\[
\text{map} : \forall \alpha, \beta, \sigma, \varepsilon. (\alpha \rightarrow \beta!\sigma/\varepsilon) \rightarrow (\alpha \text{ list } \rightarrow \beta \text{ list}!\sigma/\varepsilon)!{}\{}/{}.\]
Signatures and equations are inevitably linked, so perhaps it would be better to combine a signature and equations into a single theory component. Another challenge is designing a logic for proofs of handler correctness that works well with polymorphism.

Perhaps an easier first step is to try and add open handlers, where operation calls with no suitable operation case are propagated outwards. This approach could perhaps already be done in EEFF, where one would extend handlers with identity cases for previously unknown operations. The main problem of open handlers is that proofs of correctness rely on the precise typing information. Constructing an additional proof each time a handler needs to be opened might not be too cumbersome if handlers are used sparingly. But a more interesting question is whether we can show correctness of open handlers without switching to a fully polymorphic system.

**Stronger logics and tools**

The only requirement of the coupled logic system is to provide a respects relation (hopefully in a sound way). This opens up not only options to improve the current logics (perhaps adding list induction), but also for coupling with more exotic systems. As an example, the logic we use in the implementation is not even sound. We are hopeful that there might be an option for a tool like QuickCheck [12], which could do automated testing of handler correctness. This does not remove the burden of proof from the user, but aids in discovery of mistakes before a proper proof is attempted. Another option is to dispatch proofs to an SMT solver, similarly to the F\* project [29].

Perhaps, the first step should merely be making proofs easier to write. The goal of the formalisation was to avoid mistakes when designing EEFF, with the role of reasoning being secondary. It should be possible to ease the use of the formalisation as a reasoning tool by providing better tactics. Another angle is to improve the inference algorithm of EEFF to additionally output a Coq file, which contains all of the lemmas that the user must prove in order for the program to be fully type-checked.
Bibliography


Appendix A

Collected judgements

A.1 Operational semantics

\[
\begin{align*}
    & c_1 \leadsto c'_1 \quad \text{DoStep} \quad \text{DoRet} \\
    & \text{do } x \leftarrow c_1 \text{ in } c_2 \leadsto \text{do } x \leftarrow c'_1 \text{ in } c_2 \\
    & \text{do } x \leftarrow op_{A \rightarrow B}(v; y, c_1) \text{ in } c_2 \leadsto \text{do } x \leftarrow c'_1 \text{ in } c_2 \\
    & (\text{fun } (x : A) \mapsto c) \mapsto c[x \mapsto v] \quad \text{AppFun} \\
    & \text{let rec } f : A \rightarrow C = c_1 \text{ in } c_2 \leadsto \text{let rec } f : A \rightarrow C = c'_1 \text{ in } c_1[x \mapsto y] \\
    & c \leadsto c' \quad \text{HandleStep} \\
    & \text{with } v \text{ handle } c \leadsto \text{with } v \text{ handle } c' \\
    & \text{with } (\text{handler } (\text{ret } (x : A) \mapsto c_r; h)) \text{ handle } (\text{ret } v) \leadsto c_r[x \mapsto v] \\
    & H = \text{handler } (\text{ret } (x : A) \mapsto c_r; h) \quad \text{HANDLEOp} \\
    & \text{with } H \text{ handle } (op_{A \rightarrow B}(v; y, c) \mapsto c_{op}[x \mapsto v, k \mapsto (\text{fun } (y : B_{op}) \mapsto \text{with } H \text{ handle } c)] \\
    & \text{match } (v_1, v_2) \text{ with } (x, y) \mapsto c \leadsto c[x \mapsto v_1, y \mapsto v_2] \\
    & \text{match } (\text{Left}_{A+B} v) \text{ with } \text{Left } x \mapsto c_1 \mid \text{Right } y \mapsto c_2 \leadsto c_1[x \mapsto v] \\
    & \text{match } (\text{Right}_{A+B} v) \text{ with } \text{Left } x \mapsto c_1 \mid \text{Right } y \mapsto c_2 \leadsto c_2[y \mapsto v] \\
    & \text{match } [ ] \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 \leadsto c_1 \\
    & \text{match } (\text{v :: vs}) \text{ with } [ ] \mapsto c_1 \mid x :: xs \mapsto c_2 \leadsto c_2[x \mapsto v, xs \mapsto vs] \\
\end{align*}
\]
A.2 Typing judgements

Subtyping for types, signatures, and equations

\[
\begin{array}{l}
\text{unit} \leq \text{unit} & \text{STyUnit} \\
\text{int} \leq \text{int} & \text{STyInt} \\
\text{empty} \leq \text{empty} & \text{STyEmpty} \\
A \leq A' & \text{STyList} \\
B \leq B' & \text{STySum} \\
A \times B \leq A' \times B' & \text{STyProd} \\
A \rightarrow C \leq A' \rightarrow C' & \text{STyFun} \\
\Sigma \leq \Sigma' & \text{STySig} \\
E \leq E' & \text{STyEqs} \\
\end{array}
\]

Well-formed types, contexts, and equations

\[
\begin{array}{l}
\vdash \text{unit} : \text{vtype} & \text{WfTyUnit} \\
\vdash \text{int} : \text{vtype} & \text{WfTyInt} \\
\vdash \text{empty} : \text{vtype} & \text{WfTyEmpty} \\
\vdash A : \text{vtype} & \text{WfTyFun} \\
\vdash C : \text{ctype} & \text{WfTyHandler} \\
\vdash A \rightarrow C : \text{vtype} & \text{WfTyProd} \\
\vdash A \times B : \text{vtype} & \text{WfTySum} \\
\vdash A : \text{vtype} & \text{WfTyList} \\
\vdash A \text{ list} : \text{vtype} & \text{WfTySig} \\
\vdash \Sigma : \text{sig} & \text{WfTyCtx} \\
\vdash E : \text{Eqs} & \text{WfEqs} \\
\vdash \Sigma : \text{sig} & \text{WfTyctx} \\
\vdash Z : \text{ctx} & \text{WfTyctx} \\
\vdash A : \text{vtype} & \text{WfTyctx} \\
\vdash C : \text{ctx} & \text{WfTyctx} \\
\vdash \text{empty} : \text{ctx} & \text{WfTyctx} \\
\end{array}
\]
Well-formed templates

\[
\begin{array}{l}
\Gamma \vdash v : A \quad (z : A \to *) \in Z \\
\hline
\Gamma ; Z \vdash z \, v : \Sigma \\
\end{array}
\]
\[\text{WfTApp}\]

\[
\begin{array}{l}
\Gamma \vdash c : A! / / \}
\hline
\Gamma , x : A ; Z \vdash T : \Sigma \\
\Gamma ; Z \vdash \text{do pure } x \leftarrow c \text{ in } T : \Sigma \\
\end{array}
\]
\[\text{WfTD}\]

\[
\begin{array}{l}
\Gamma \vdash v : A \times B \\
\hline
\Gamma , x : A , y : B ; Z \vdash T : \Sigma \\
\end{array}
\]
\[\text{WfTPodMatch}\]

\[
\begin{array}{l}
\Gamma ; Z \vdash \text{match } v \text{ with } (x,y) \mapsto T : \Sigma \\
\end{array}
\]
\[\text{WfTSumMatch}\]

\[
\begin{array}{l}
\Gamma ; Z \vdash \text{match } v \text{ with } [ ] \mapsto T_1 \mid x :: xs \mapsto T_2 : \Sigma \\
\end{array}
\]
\[\text{WfTListMatch}\]

\[
\begin{array}{l}
\Gamma \vdash v : A + B \\
\hline
\Gamma , x : A ; Z \vdash T_1 : \Sigma \\
\Gamma , y : B ; Z \vdash T_2 : \Sigma \\
\end{array}
\]
\[\text{WfTSumMatch}\]

Well-typed values \(\Gamma \vdash v : A\) (where \(\vdash \Gamma : \text{ctx}\) and \(\vdash A : \text{vtype}\))

\[
\begin{array}{l}
(x : A) \in \Gamma \\
\hline
\Gamma \vdash x : A \\
\end{array}
\]
\[\text{TypeVar}\]

\[
\begin{array}{l}
\end{array}\]
\[\text{TypeUnit}\]

\[
\begin{array}{l}
\end{array}\]
\[\text{TypeInt}\]

\[
\begin{array}{l}
\end{array}\]
\[\text{TypeFun}\]

\[
\begin{array}{l}
\end{array}\]
\[\text{TypeHandler}\]

\[
\begin{array}{l}
\end{array}\]
\[\text{TypeVSubsume}\]
Well-typed computations $\Gamma \vdash c : C$ (where $\vdash \Gamma : \text{ctx}$ and $\vdash C : \text{ctype}$)

$$
\begin{align*}
\Gamma \vdash v : A & \quad \text{TYPERET} \quad \Gamma \vdash v_1 : A \to C & \quad \Gamma \vdash v_2 : A & \quad \text{TYPEAPP} \\
\Gamma \vdash \text{ret} v : A!\{\} & \quad \text{TYPELetRec} & \quad \Gamma \vdash f : A \to C & \quad \Gamma \vdash c_1 : C & \quad \Gamma \vdash f : A \to C & \quad \Gamma \vdash c_2 : D \\
\Gamma \vdash \text{let rec } f \ x : A \to C = c_1 \ \text{in } c_2 : D & & \\
\frac{\text{let } a : A \rightarrow \Sigma \Rightarrow \Gamma \vdash v : A \rightarrow C \leadsto \frac{\text{absurd } \Sigma \Rightarrow \Gamma \vdash 1 \equiv 1}{\Gamma \vdash \text{absurd}_C v : C}}{\Gamma \vdash \text{absurd}_C v : C} & & \\
\frac{\text{do } x \leftarrow c_1 \ \text{in } c_2 : B!\Sigma/\text{E} \leadsto \frac{\text{match } \text{with } (x, y \mapsto c_2 : C)}{\Gamma \vdash \text{do } x \leftarrow c_1 \ \text{in } c_2 : B!\Sigma/\text{E}}}{\Gamma \vdash \text{empty}} & & \\
\frac{\text{match } v \text{ with } \{ \text{Left } x \mapsto c_1 | \text{Right } y \mapsto c_2 : C \}}{\Gamma \vdash v : A \rightarrow B} & & \\
\frac{\text{match } v \text{ with } [ ] \mapsto \text{match } v \text{ with } [ x : \text{xs} \mapsto c_2 : C]}{\Gamma \vdash v : A \text{ list}} & & \\
\frac{\Gamma \vdash c : C \quad C \leq C'}{\Gamma \vdash c : C'} & & \\
\end{align*}
$$

Well-typed operation cases $\Gamma \vdash h : \Sigma \Rightarrow D$ (where $\vdash \Gamma : \text{ctx}$, $\vdash \Sigma : \text{sig}$, and $\vdash D : \text{ctype}$)

$$
\begin{align*}
\Gamma \vdash \{ \} : D & \quad \text{TYPECases} \{ \} \\
\Gamma \vdash h : \Sigma \Rightarrow D & \quad \text{TYPECases} \{ \text{op}_A \to B(x; k) \mapsto c_{op} \} \quad \{ \Sigma \cup \{ \text{op} : A \to B \} \} \Rightarrow D \\
\end{align*}
$$

A.3 Equational logic

Equations on values $\Gamma \vdash v \equiv_A v'$ (where $\Gamma \vdash v : A$ and $\Gamma \vdash v' : A$)

$$
\begin{align*}
\Gamma \vdash v_1 \equiv_A v_2 & \quad \text{VeqSym} \\
\Gamma \vdash v_2 \equiv_A v_1 & \quad \text{VeqTrans} \\
\frac{x : A' \in \Gamma}{\Gamma \vdash x \equiv_A x} & \quad \text{VeqVar} \\
\frac{\Gamma \vdash v_1 \equiv_A v_2 \quad \Gamma \vdash v_2 \equiv_A v_3}{\Gamma \vdash v_1 \equiv_A v_3} & \quad \text{VeqTrans} \\
\frac{\Gamma \vdash () \equiv \text{unit} ()}{\Gamma \vdash n \equiv \text{int} n} & \quad \text{VeqInt} \\
\frac{\Gamma \vdash (v_1, v_2) \equiv_A (v_1', v_2')}{\Gamma \vdash (v_1, v_2) \equiv_A (v_1', v_2')} & \quad \text{VeqPair} \\
\frac{\Gamma \vdash \text{Left}_{A_1 + B_1} v \equiv_{A_1 + B_1} \text{Left}_{A_2 + B_2} v'}{\Gamma \vdash \text{Left}_{A_1 + B_1} v \equiv_{A_1 + B_1} \text{Left}_{A_2 + B_2} v'} & \quad \text{VeqLeft} \\
\end{align*}
$$

152
Equations on computations $\Gamma \vdash c \equiv_C c'$ (where $\Gamma \vdash c : C$ and $\Gamma \vdash c' : C$)

\[
\begin{align*}
\Gamma \vdash c_1 \equiv_C c_2 & \quad \text{CRQSYM} \\
\Gamma \vdash c_2 \equiv_C c_1 & \quad \text{CRQTRANS} \\
\Gamma \vdash v \equiv_A v' & \quad \text{CRQRET} \\
\Gamma \vdash \text{ret } v \equiv_{A \Sigma/E} \text{ret } v' & \quad \text{CRQABSORB} \\
\Gamma \vdash x : A \vdash c_1 \equiv_{B \Sigma/E} c_1' & \quad \text{CEQDO} \\
\Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 \equiv_{B \Sigma/E} \text{do } x \leftarrow c_1' \text{ in } c_2' & \quad \text{CEQAPP} \\
\Gamma \vdash v \equiv_{C \equiv_D} v' & \quad \text{CEQHANDLE} \\
\Gamma \vdash v \equiv_{D \equiv_D} v' & \quad \text{CEQLETREC} \\
\Gamma \vdash \text{if } \text{then } x \rightarrow c_1 \text{ else } c_2 \equiv_{A \Sigma/E} \text{if } \text{then } x \rightarrow c_1' \text{ else } c_2' & \quad \text{CEQOP} \\
\Gamma \vdash v \equiv_{A \times B} v' & \quad \text{CEQPRODMATCH} \\
\Gamma \vdash v \equiv_{A + B} v' & \quad \text{CEQSUMMATCH} \\
\Gamma \vdash \text{do } x \leftarrow \text{ret } v \blacktriangleright_{C} x \rightarrow v & \quad \beta\text{DoRet}
\end{align*}
\]
\[\Gamma \vdash \text{do } x \leftarrow op_{A \rightarrow B}(v \mid y).c \in c_2 \equiv_{C} \text{do } x \leftarrow c_1 \text{ in } c_2 \]

\[\Gamma \vdash (\text{fun } (x : A) \mapsto c) \in C \equiv_{C} c[x \mapsto v] \]

\[\Gamma \vdash \text{let rec } f \mapsto x : A \mapsto c = c_1 \text{ in } c_2 \equiv_{D} c_2[f \mapsto \text{let rec } f \mapsto x : A \mapsto c = c_1[x \mapsto y]]\]

\[\Gamma \vdash \text{with handler } (\text{ret } (x : A) \mapsto c_r \mid h) \text{ handle } (\text{ret } v) \equiv_{C} c_r[x \mapsto v] \]

\[H = \text{handler } (\text{ret } (x : A) \mapsto c_r \mid h) \equiv_{C} \text{op}_{A \rightarrow B}(v \mid k) \mapsto c_{op} \in h \]

\[\Gamma \vdash \text{with } H \text{ handle } \text{op}_{A \rightarrow B}(v \mid y) \equiv_{C} \text{op}[x \mapsto v, k \mapsto (\text{fun } y : B_{op} \mapsto \text{with } H \text{ handle } c)] \]

\[\Gamma \vdash \text{(match } (v_1, v_2) \text{ with } (x, y) \mapsto c) \equiv_{C} c[x \mapsto v_1, y \mapsto v_2] \]

\[\Gamma \vdash \text{(match } (\text{Left}_{A + B} v) \text{ with } \text{Left } x \mapsto c_1 \mid y \mapsto c_2) \equiv_{C} c_1[x \mapsto v] \]

\[\Gamma \vdash \text{(match } (\text{Right}_{A + B} v) \text{ with } y \mapsto c_2 \mid \text{Right } x \mapsto c_1 \equiv_{C} c_2[y \mapsto v] \]

\[\Gamma \vdash \text{(match } [] \text{ with } [] \mapsto c_1 \mid x \mapsto xs \mapsto c_2) \equiv_{C} c_1[x \mapsto v, xs \mapsto vs] \]

\[\Gamma, e : \text{empty}, \Gamma_2 \vdash c \mapsto C \]

\[\eta_{\text{EMPTY}} \]

\[\Gamma, p : A \times B, \Gamma_2 \vdash c \mapsto C \]

\[\Gamma, \text{match } v \text{ with } (x, y) \mapsto c[p \mapsto (x, y)] \equiv_{C} \text{match } v \text{ with } Left \mapsto \{c[s \mapsto \text{Left}_{A + B} x]\} \mid \text{Right } \mapsto \{c[s \mapsto \text{Right}_{A + B} x]\} \]

\[\Gamma, l : A \text{ list }, \Gamma_2 \vdash c \mapsto C \]

\[\eta_{\text{LIST}} \]

\[\Gamma, \text{do } c \equiv_{C} \text{do } x \leftarrow c \text{ in } \text{ret } x \]

\[\eta_{\text{Do}} \]

\[((x_i : A_i), (z_j : B_j) \mapsto *)_j \vdash T_1 \sim T_2 \in \mathcal{E} \quad A\Sigma / \mathcal{E} \leq C \quad (\Gamma \vdash v_i : A_i) \quad (\Gamma \vdash u_j : B_j \rightarrow A\Sigma / \mathcal{E})_j \]

\[\Gamma \vdash T_1[A\Sigma / \mathcal{E}][v_i \mapsto v_i], (z_j \mapsto u_j)_j \equiv_{C} T_2[A\Sigma / \mathcal{E}][v_i \mapsto v_i], (z_j \mapsto u_j)_j \]

\text{OOTB}
Equations on operation cases $\Gamma \vdash h \equiv_{\Sigma = D} h'$.

There must exist $\Sigma_1, \Sigma_2$ such that $\Sigma \leq \Sigma_1$ and $\Sigma \leq \Sigma_2$ with $\Gamma \vdash h : \Sigma_1 \Rightarrow D$ and $\Gamma \vdash h' : \Sigma_2 \Rightarrow D$.

$$\Gamma \vdash h \equiv() =_{D} h' [\text{HSeqSig}()]$$

$$\frac{(op_{A_1 \rightarrow B_1}(x; k) \mapsto c_{op}) \in h}{\Gamma, x : A, k : B \Rightarrow D \vdash c_{op} \equiv_{D} c'_{op} \quad \Gamma \vdash h \equiv_{\Sigma = D} h'}$$

$$\Gamma \vdash h \equiv() =_{D} h' [\text{HSeqSigU}()]$$

Correctness of handler cases $\Gamma \vdash h : E \Rightarrow \Sigma$ respects $D$

It must hold that $\vdash E : \Sigma$, and $\Gamma \vdash h : \Sigma \Rightarrow D$.

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } \{\}}{\text{RespectEqs}()}$$

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } E}{\Gamma, (x_i : A_i), (z_j : B_j \rightarrow D) \vdash T_1^h \equiv_D T_2^h}$$

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } (E \cup \{ (x_i : A_i) ; (z_j : B_j \rightarrow *) \})}{\text{RespectEqsU}()}$$

Instantiating templates $T^h_D$ (where $\Gamma ; Z \vdash T : \Sigma$ and $\Gamma \vdash h : \Sigma \Rightarrow D$)

The computation $c_{op}$ is the operation case for $op$ in $h$. If $I$ is used in place of $h$ this represents identity cases, with $op(x; k) \mapsto op(x; y.k y)$.

$$\frac{(z_i v)^D = z_i v}{(op_{A \rightarrow B}(v; y.T))^D = c_{op}[x \mapsto v, k \mapsto (\text{fun } y : B \mapsto T^h_D)]}$$

$$\frac{\text{do pure } x \leftarrow c \text{ in } T^h_D = \text{do } x \leftarrow c \text{ in } T^h_D}{\text{absurd } v^D = \text{absurd}_D v}$$

$$\frac{\text{match } v \text{ with } (x, y) \mapsto T^h_D = \text{match } v \text{ with } (x, y) \mapsto (T^h_D)}{\text{match } v \text{ with } [ ] \mapsto T_1 | x :: xs \mapsto T_2^h_D = \text{match } v \text{ with } [ ] \mapsto (T_1^h_D | x :: xs \mapsto (T_2^h_D)}$$

$$\frac{\text{match } v \text{ with Left } x \mapsto T_1 | \text{Right } y \mapsto T_2^h_D}{\text{match } v \text{ with Left } x \mapsto (T_1^h_D) | \text{Right } y \mapsto (T_2^h_D)}$$
A.4 Predicate logic

Well-formedness of logic formulae and hypotheses

\[
\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1 \equiv_A v_2 : \text{form}} \quad \text{WfEq}
\]

\[
\frac{\Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C}{\Gamma \vdash c_1 \equiv_C c_2 : \text{form}} \quad \text{WfCeq}
\]

\[
\frac{\vdash \Sigma : \text{sig} \quad \Sigma \subseteq \Sigma_1 \quad \Sigma \subseteq \Sigma_2}{\Gamma \vdash h_1 : \Sigma_1 \quad \Gamma \vdash h_2 : \Sigma_2 \Rightarrow D \quad \Gamma \vdash h_2 : \Sigma_2 \Rightarrow D}{\Gamma \vdash h_1 \equiv_{\Sigma=D} h_2 : \text{form}} \quad \text{WfEq}
\]

\[
\frac{\Gamma \vdash \top}{\Gamma \vdash \top : \text{form}} \quad \text{WfT}
\]

\[
\frac{\Gamma \vdash \bot}{\Gamma \vdash \bot : \text{form}} \quad \text{Wf⊥}
\]

\[
\frac{\Gamma \vdash \varphi_1 : \text{form}}{\Gamma \vdash \varphi_1 \land \varphi_2 : \text{form}} \quad \text{WfA}
\]

\[
\frac{\Gamma \vdash \varphi_1 : \text{form} \quad \Gamma \vdash \varphi_2 : \text{form}}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2 : \text{form}} \quad \text{Wf⇒}
\]

\[
\frac{\Gamma, x : A \vdash \varphi : \text{form}}{\Gamma \vdash \forall x : A. \varphi : \text{form}} \quad \text{Wf∀}
\]

\[
\frac{\Gamma, x : A \vdash \varphi : \text{form}}{\Gamma \vdash \exists x : A. \varphi : \text{form}} \quad \text{Wf∃}
\]

\[
\frac{\Gamma \vdash \psi : \text{hyp} \quad \Gamma \vdash \varphi : \text{hyp}}{\Gamma \vdash \psi, \varphi : \text{hyp}} \quad \text{WfHypU}
\]

Changes and additions to equation judgements

We inherit all equation judgements from equational logic by switching \(\Gamma\) with \(\Gamma | \Psi\), since they do not use hypotheses. All rules require all parts to be well-formed. The rule \(\eta\) Do can be dropped.

\[
\frac{\Gamma | \Psi \vdash h_2 \equiv_{\Sigma=D} h_1}{\Gamma | \Psi \vdash h_1 \equiv_{\Sigma=D} h_2 \quad \Gamma | \Psi \vdash h_2 \equiv_{\Sigma=D} h_3}{\Gamma | \Psi \vdash h_1 \equiv_{\Sigma=D} h_3} \quad \text{HeqSvm}
\]

\[
\frac{\Gamma | \Psi \vdash h \equiv_{\Sigma=D} h'}{\Gamma | \Psi \vdash h \cup \{op_{A \rightarrow B}(x; k) \mapsto c\} \equiv_{\Sigma \cup \{op_{A \rightarrow B}(x; k) \mapsto c\}} h' \cup \{op_{A \rightarrow B}(x; k) \mapsto c\}} \quad \text{HeqExtend}
\]

\[
\frac{\Gamma | \Psi \vdash v_1 \equiv_{A'} v_2 \quad A' \subseteq A}{\Gamma | \Psi \vdash v_1 \equiv_{A} v_2} \quad \text{VeqSubsume}
\]

\[
\frac{\Gamma | \Psi \vdash c_1 \equiv_{C} c_2 \quad C' \subseteq C}{\Gamma | \Psi \vdash c_1 \equiv_{C} c_2} \quad \text{VeqSubsume}
\]

\[
\frac{x : A \in \Gamma}{\Gamma | \Psi \vdash x \equiv_{A} x} \quad \text{VeqVar}
\]

\[
\frac{\Gamma | \Psi \vdash \text{handler}(x \mapsto c; h) \equiv_{A\Sigma/E=D} \text{handler}(x \mapsto c'; h')}{\Gamma | \Psi \vdash \text{handler}(x \mapsto c; h) \equiv_{A\Sigma/E=D} \text{handler}(x \mapsto c'; h')} \quad \text{VeqHandler}
\]

\[
\frac{((x_i : A_i); (z_j : B_j \mapsto *)_j \vdash T_1 \sim T_2) \in E \quad (\Gamma \vdash v_i : A_i)_i \quad (\Gamma \vdash u_j : B_j \rightarrow A!\Sigma/E)_j \quad \vdash T_1^I_{A\Sigma/E}[(x_i \mapsto v_i)_i, (z_j \mapsto u_j)_j] \equiv_{A\Sigma/E} T_2^I_{A\Sigma/E}[(x_i \mapsto v_i)_i, (z_j \mapsto u_j)_j]}{\Gamma | \Psi \vdash \text{OOTB}}
\]

\[
\frac{\Gamma | \Psi \vdash \Sigma \Rightarrow D \text{ respects } D \quad \Gamma | \Psi \vdash \Sigma \Rightarrow D \text{ respects } D}{\Gamma | \Psi \vdash \text{RespectEq} \Sigma U} \quad \text{RespectEqSU}
\]

156
Judgements for formulae $\Gamma \vdash \varphi$ (where $\vdash \Gamma : \text{ctx}$, $\Gamma \vdash \Psi : \text{hyp}$, and $\Gamma \vdash \Gamma : \text{form}\varphi$)

\[\begin{array}{c}
\frac{\varphi \in \Psi}{\Gamma \vdash \varphi} \quad \text{IsHYP} \\
\frac{\Gamma \vdash \varphi}{\Gamma \vdash \bot} \quad \text{TIN} \\
\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \quad \text{aEL} \\
\frac{\Gamma \vdash \varphi_1 \land \varphi_2}{\Gamma \vdash \varphi_1} \quad \text{aELleft} \\
\frac{\Gamma \vdash \varphi_1 \land \varphi_2}{\Gamma \vdash \varphi_2} \quad \text{aELright} \\
\frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \lor \varphi_2} \quad \text{vINleft} \\
\frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \lor \varphi_2} \quad \text{vINright} \\
\frac{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2}{\Gamma \vdash \varphi_1} \quad \Rightarrow \text{IN} \\
\frac{\Gamma \vdash \varphi_2 \Rightarrow \varphi_1}{\Gamma \vdash \varphi_2} \quad \Rightarrow \text{EL} \\
\frac{\Gamma \vdash \varphi[\varphi[x \mapsto v]]}{\Gamma \vdash \forall x : A. \varphi} \quad \text{vEL} \\
\frac{\Gamma \vdash \varphi \Rightarrow \varphi}{\Gamma \vdash \varphi} \quad \Rightarrow \text{EL} \\
\frac{\Gamma \vdash \forall x : A. \varphi}{\Gamma \vdash \exists x : A. \varphi} \quad \text{vIN} \\
\frac{\Gamma \vdash \exists x : A. \varphi}{\Gamma \vdash \exists x : A. \varphi} \quad \exists \text{EL}
\end{array}\]

DoLoop
\[\begin{array}{c}
\frac{\Gamma \vdash \text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } c}{\text{let } \Gamma \vdash \varphi}
\end{array}\]

HandleLoop
\[\begin{array}{c}
\frac{\Gamma \vdash \text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } c}{\text{let } \Gamma \vdash \varphi}
\end{array}\]

\[\begin{array}{c}
\text{Induction} \\
\frac{\varphi \text{ is admissible in } \psi}{\frac{\Gamma \vdash \varphi \Rightarrow \varphi}{\Gamma \vdash \varphi}} \\
\frac{\Gamma \vdash \varphi \Rightarrow \varphi}{\Gamma \vdash \varphi}
\end{array}\]

\[\begin{array}{c}
\frac{\Gamma : \text{ctx}, \Gamma \vdash \Psi : \text{hyp}, \Gamma \vdash \Gamma : \text{form} \varphi}{\Gamma \vdash (\text{let rec } g \_ : \text{unit} \rightarrow C = g () \text{ in } c)}
\end{array}\]
Appendix B

Collected denotational semantics

B.1 Type and term semantics

Semantics of well-formed types

- \([\text{unit}] = \{\star\}\)
- \([\text{int}] = \mathbb{N}\)
- \([\text{empty}] = \emptyset\)
- \([A \times B] = [A] \times [B]\)
- \([A + B] = [A] + [B]\)
- \([A \text{ list}] = [A]^\ast\)
- \([A \rightarrow C] = [A] \rightarrow [C]\)
- \([C \Rightarrow D] = [C] \rightarrow [D]\)

\([\Sigma]X = T_I(X,\{[A_{op}]\}_{op \in \Sigma},\{[B_{op}]\}_{op \in \Sigma})\)

\([A!\Sigma/E] = [\Sigma][A]\)

Here \(T_I(A, (A_i)_i, (B_i)_i)\) is the solution of the domain equation

\[
F(D) = \left( A + \prod_{i \in I} A_i \times (B_i \rightarrow D) \right)_\perp.
\]

Semantics of contexts

- \([\cdot] = \{\star\}\)
- \([\Gamma, x : A] = [\Gamma] \times [A]\)

Semantics of subtyping

- \([\text{unit} \leq \text{unit}] = id_{\text{unit}}\)
- \([\text{int} \leq \text{int}] = id_{\text{int}}\)
- \([\text{empty} \leq \text{empty}] = id_{\text{empty}}\)

- \([A + B \leq A' + B'] = \lambda x. \begin{cases} t_1([A \leq A']a) &; x = t_1(a) \\ t_2([B \leq B']b) &; x = t_2(b) \end{cases}\)

- \([A \times B \leq A' \times B'] = \lambda (a,b). ([A \leq A']a)([B \leq B']b)\)

- \([A \text{ list} \leq A' \text{ list}] = A(a_i)_{i=0}^n \cdot ([A \leq A']a_i)_{i=0}^n\)

- \([A \rightarrow C \leq A' \rightarrow C'] = \lambda f. ([C \leq C'] \circ f \circ [A' \leq A])\)

- \([C \Rightarrow D \leq C' \Rightarrow D'] = \lambda g. ([D \leq D'] \circ g \circ [C' \leq C])\)

- \([A!\Sigma/E \leq A'!\Sigma'/E'] = \text{lift}_{[\Sigma \leq \Sigma']A'}([A \text{ list}] \circ \text{id}_{\text{val}}([A \leq A']a))\)

Semantics of values and operation cases

- \([\Gamma \vdash \text{unit}]\eta = \star\)
- \([\Gamma \vdash n : \text{int}]\eta = n\)
- \([\Gamma \vdash x_k : A_k \vdash x_i : A_i]\eta = \eta_i\)
- \([\Gamma \vdash (v_1, v_2) : A \times B]\eta = ([v_1]\eta, [v_2]\eta)\)
- \([\Gamma \vdash \text{Left}_{A+B} v : A + B]\eta = \text{id}(v)\eta\)
- \([\Gamma \vdash \text{Right}_{A+B} v : A + B]\eta = \text{id}(v)\eta\)
- \([\Gamma \vdash \text{let} \rightarrow A+B v : A + B]\eta = v\eta\)
- \([\Gamma \vdash v_1 : v_2 : A \text{ list}]\eta = [v_1]\eta :: [v_2]\eta\)

- \([\Gamma \vdash v_1 \Rightarrow v_2 : A \text{ list}]\eta = [v_1]\eta :: [v_2]\eta\)

159
\[ \Gamma \vdash (\text{fun } x : A \mapsto c) : A \rightarrow \mathbb{C}(\eta = \lambda a \in \mathbb{A}). \mathbb{C}(\Gamma, x : A + c : \mathbb{C}(\eta, a)) \]

\[ \Gamma \vdash \text{handler } (\text{ret } x : A \mapsto c_r ; h) : A \Sigma/E \Rightarrow \mathbb{D}(\eta = \text{lift}_{\mathbb{B}(\eta, \eta)}(\lambda a \in \mathbb{A}). \mathbb{C}(\eta, a)) \]

\[ \Gamma \vdash \{ \text{op}_{A \rightarrow B}(x ; k) \mapsto c_{op} \}_{op} : \Sigma \Rightarrow \mathbb{D}(\eta = \{ \lambda a . \lambda k . \langle \Gamma, x : A_{op}, k : B_{op} \rightarrow \mathbb{D} + c_{op} : \mathbb{D}(\eta, a, k) \}_{op} : A_{op} \rightarrow B_{op} \}_{\Sigma} \]

**Semantics of computations**

\[ \Gamma \vdash \text{ret } v : \mathbb{C}(\eta = \text{in}_{\mathbb{C}}(\mathbb{v}(\eta)) \]
\[ \Gamma \vdash \text{op}_{A \rightarrow B}(v; y) : \mathbb{C}(\eta = \text{in}_{\mathbb{C}}(\eta \leq A' \Rightarrow \mathbb{D}(\eta; \lambda b \cdot \langle \Gamma, y : B + c : \mathbb{C}(\eta, \mathbb{B}(\eta) \leq B(b) \rangle) \]

\[ \Gamma \vdash v_1 v_2 : \mathbb{C}(\eta = (\mathbb{v}_1(\eta) \mathbb{v}_2(\eta)) \]
\[ \Gamma \vdash \text{with } v \text{ handle } c : \mathbb{D}(\eta = (\mathbb{v}(\eta) \mathbb{c}(\eta)) \]

\[ \Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 : B \Sigma/E \Rightarrow \mathbb{D}(\eta = \text{lift}_{\mathbb{B}(\eta, \eta)}(\lambda a \in \mathbb{A}). \langle \Gamma, x : A + c_2 : B \Sigma/E(\eta, a) \rangle)(\mathbb{c}_1(\eta)) \]

\[ \Gamma \vdash \text{let rec } f x : A \mapsto c = c_1 \text{ in } c_2 : \mathbb{D}(\eta = \langle \Gamma, f : A \mapsto c + c_2 : \mathbb{D}(\eta, \mu f . \lambda a \in \mathbb{A}. \langle \Gamma, x : A + f : A \mapsto c + c_1 : \mathbb{C}(\eta, a, f) \rangle) \]

\[ \Gamma \vdash \text{absurd}_{\mathbb{C}} v : \mathbb{C}(\eta = \text{empty}_\mathbb{C}(\eta)) \]

\[ \Gamma \vdash v \text{ with } (x, y) \mapsto c : \mathbb{C}(\eta = \langle \Gamma, x : A, y : B + c : \mathbb{C}(\eta, a, b) \rangle) \quad \text{for } \langle \mathbb{v}(\eta) = (a, b) \rangle \]

\[ \Gamma \vdash \text{match } v \text{ with } \text{Left } x \mapsto c_1 | \text{Right } y \mapsto c_2 : \mathbb{C}(\eta = \langle \Gamma, x : A + c_1 : \mathbb{C}(\eta, a), \text{if } \mathbb{v}(\eta) = c_1(a) \rangle, \Gamma, y : B + c_2 : \mathbb{C}(\eta, b), \text{if } \mathbb{v}(\eta) = c_2(b) \rangle \]

\[ \Gamma \vdash \text{match } v \text{ with } [ ] \mapsto c_1 | x :: xs \mapsto c_2 : \mathbb{C}(\eta = \langle \Gamma + c_1 : \mathbb{C}(\eta), \text{if } \mathbb{v}(\eta) = x \rangle, \Gamma, x : A, xs : A \text{ list } + c_2 : \mathbb{C}(\eta, a_0, (a_i)_{i=1}^n), \text{if } \mathbb{v}(\eta) = a_0, \ldots, a_n \rangle \]

**B.2 Theory semantics**

**Semantics of Templates**

\[ \llbracket \cdot \rrbracket \mathbb{Y} = \{\ast\} \]
\[ \llbracket Z, z : A \mapsto * \rrbracket \mathbb{Y} = \llbracket \mathbb{Z} \rrbracket \mathbb{Y} \times \mathbb{Y}(\mathbb{A}) \]

160
The relation \(c_1 \sim c_2\) is the empty relation.
- \(\sim_{\text{empty}}\) is the empty relation.
- \(\sim_{\text{unit}}\) and \(\sim_{\text{int}}\) are identity relations.
- \((a, b) \sim_{\text{AXB}} (a', b') \iff a \sim_{A} a' \land b \sim_{B} b'\)
- \(\tau_1(a) \sim_{A+B} \tau_1(a') \iff a \sim_{A} a'\) and \(\tau_2(b) \sim_{A+B} \tau_2(b') \iff b \sim_{B} b'\)
- \((a_i)_{i=0}^n \sim_{\text{Alist}} (a'_i)_{i=0}^n \iff \forall i = 0, \ldots, n. a_i \sim_{A} a'_i\)
- \(f \sim_{\text{op}} f' \iff (\forall a, a' \in [A]. a \sim_{A} a' \implies f(a) \sim_{C} f'(a'))\)
- \(h \sim_{\text{D}} h' \iff (\forall c, c' \in [C]. c \sim_{C} c' \implies h(c) \sim_{D} h'(c'))\)
- \(H \sim_{\text{D}} H' \iff \left(\forall op : A \to B \in \Sigma. \forall a, a' \in [A]. \forall \kappa, \kappa' \in [B \to D]. a \sim_{A} a' \land \kappa \sim_{B \to D} \kappa' \implies H_{\text{op}}(a, \kappa) \sim_{D} H'_{\text{op}}(a', \kappa')\right)\)

Relations for computations

The relation \(\sim_{\text{C}}\) for \(C = A\Sigma/E\) is the smallest relation closed under the following rules:

1. \(\bot \sim_{\text{C}} \bot\).
2. If \(a \sim_{A} a'\) then \(\text{inva}(a) \sim_{A\Sigma/E} \text{inva}(a')\).
3. For \(op : A_{\text{op}} \to B_{\text{op}} \in \Sigma\), if \(a \sim_{A_{\text{op}}} a'\) and if \(b \sim_{B_{\text{op}}} b'\) implies \(\kappa(b) \sim_{C} \kappa'(b')\) then \(\text{in}_{op}(a, \kappa) \sim_{C} \text{in}_{op}(a', \kappa')\).
4. Let \(\Gamma; Z \vdash T_1 - T_2\) be an equation in \(E\), where \(\Gamma = (x_i : A_i)_i\) and \(Z = (z_j : B_j \to *)_j\). If we have \(a_i \sim_{A_i} a'_i\) for all \(i\) and if \(b \sim_{B_j} b'\) implies \(f_j(b) \sim_{C} f'_j(b')\) for all \(j\), then
   \[\Gamma; Z \vdash T_1 : \Sigma^{\Gamma \downarrow \Sigma}((a_i)_i, (f_j)_j) \sim_{C} \Gamma; Z \vdash T_2 : \Sigma^{\Gamma \downarrow \Sigma}((a'_i)_i, (f'_j)_j)\]
5. \(c_1 \sim_{C} c_2\) implies \(c_2 \sim_{C} c_1\).
6. \(c_1 \sim_{C} c_2\) and \(c_2 \sim_{C} c_3\) imply \(c_1 \sim_{C} c_3\).
7. For chains \((c_i)_i\) and \((c'_i)_i\), if \(\forall i. c_i \sim_{C} c'_i\) then \(\lor_i c_i \sim_{C} \lor_i c'_i\).
Appendix C

Povzetek v slovenščini

C.1 Uvod


Ob definiciji operacij podamo signaturo, ki vsebuje tipe operacij, in učinkovno teorijo, ki opisuje delovanje učinkov. Implementacijo učinkov določa prestreznik, ki naj bi spošтовal enačbe učinkovne teorije. Prestreznike, ki spoštujejo teorijo, imenujemo pravilni. Takšni prestrezniki ne razlikujejo med izračuni, ki jih učinkovna teorija enači, temveč jih preslikajo v ekvivalentne rezultate. Določanje pravilnosti prestreznikov je v splošnem neodločljivo [36] in zahteva logiko za delo z algebrajskimi učinki [38]. Dokazi pravilnosti so dodatno breme za uporabnike, vendar lahko logiko uporabimo tudi za sklepanje o programih z učinki.

Če zahtevamo, da so vsi prestrezniki pravilni glede na globalno učinkovno teorijo, bodisi izgubimo koristne prestreznike bodisi pa moramo uporabljati šibkeje teorije. Na primer, pri nedeterministični izbiri ponavadi privzamemo komutativnost argumentov, torej da je izbira med x in y enaka kot izbira med y in x. Posledično prestreznik, ki sestavi seznam rezultatov za vse možne izbire, ni pravilen glede na teorijo, saj vrstni red argumentov izbire vpliva na vrstni red elementov v seznamu. Če komutativnost odstranimo iz teorije, te lastnosti ni več možno uporabljati pri dokazovanju v drugih delih programa. Zaradi nekompatibilnosti teorij, ki jih različni prestrezniki spoštujejo, pogosto privzamemo teorijo brez enačb. Večina novejših pristopov [23, 7, 26, 8] učinkovne teorije v celoti zamenari. S tem poenostavimo uporabo prestreznikov, saj ni več potrebno dokazovati pravilnosti, vendar izgubimo enačbe kot orodje v logiki.

Druga veja pristopov [2, 3, 18, 43] pa se osredotoča na prednosti uporabe učinkovnih
teorij, s katerimi lahko opišemo vedenje učinkov in se oddaljimo od konkretnih implementacij. Podobne tehnike so možne pri modeliranju računskih učinkov z monadami [20, 19, 1]. Takšna orodja so ključna za področja, kjer so učinki poglavitnega pomena [42, 46, 9], in za optimizacije ob prisotnosti učinkov [24].

**Cilj**

Namen doktorske disertacije je poiskati način, ki združuje prednosti učinkovnih teorij s fleksibilnostjo prestreznikov v jezikih, ki odmislijo enačbe. Osnovna ideja je generalizacija globalnih učinkovnih teorij na lokalne učinkovne teorije, podoba preskoku iz globalnih signatur učinkov na lokalne [23]. Namesto enotne globalne teorije sedaj učinkovno teorijo podamo v tipu izračuna. Lokalnost nam omogoča uporabo močnejše logike v delih programa, kjer učinkovno teorijo uporabljamo, na preostale dele programa pa ne postavljamo dodatnih omejitev. Prav tako lahko gnezdimo uporabo različnih teorij za iste učinke, kjer uporabljamo prestreznike kot transformatorje teorij.

Pri določanju tipov so večje spremembe le pri prestreznikih, za katere moramo pokazati, da spoštujejo učinkovno teorijo. Če želimo delati v določeni teoriji, jo označimo v tipu programa, in sistem tipov pa poskrbi, da lahko uporabljamo zgolj tisti prestrezniki, s čimer se oddaljimo od konkretne implementacije prestreznika, ko sklepamo o lastnostih programa.

Uporabnost pristopa želimo pokazati z implementacijo jezika, ki uporablja lokalne učinkovne teorije. Za izpeljavo tipov se kot naravna izbira ponuja dvosmerna izpeljava tipov. Uporabo učinkovnih teorij mora označiti uporabnik, kar je skladno z označevanjem tipov, ki ga zahteva pristop dvosmerne izpeljave tipov. Pomembne lastnosti jezika želimo formalizirati v dokazovalniku Coq, s čimer zagotovimo, da definicije in dokazi niso ciklični. Formalizacijo lahko prav tako uporabljamo kot orodje za sklepanje.

### C.2 Prestrezniki algebrajskih učinkov


```
let rec print_list l =
  match l with
  | [] -> ()
  | x :: xs -> !Print x; print_list xs
```

Funkcije ob klicu zajamejo tudi nadaljevanje oz. kontinuacijo programa. Če izvedemo program `!Print "a"; !Print "b"`, se sproži klic operacije `Print` s argumentom "a" in kontinuacijo, ki čaka na rezultat klica operacije in nato nadaljuje z `!Print "b"`. Operacije so zgolj konstrukt jezika, ki skrbi za zajem kontinuiacij in za pravilno širjenje do prestreznikov. Implementacija učinka je podana s prestrezniki. Eden od možnih prestreznikov za `Print` vrne vrednost izračuna skupaj z nizom vseh izpisov. Če izračun vrne rezultat `x`, prestreznik vrne par `(x, "")`, saj ni bilo zahtevajočo izpisu. V primeru, da prestrežemo klic `Print`, najprej nadaljujemo z računanjem kontinuacije `k ()`. Zajeta kontinuacija `k` že implicitno uporablja prestreznik `collect_prints`, torej vrne par, ki vsebuje rezultat `x` in izpis `out`. Sedaj upoštevamo, da smo prestregli klic za izpis niza `s`,
ki ga dodamo na začetek končnega izpisa, torej \((x, s \rightarrow \text{out})\) (saj se je klic zgodil pred kontinuacijo).

```haskell
let collect_prints = handler 
  | effect Print s k ->
    let (x, out) = k () in
    (x, s \rightarrow \text{out})
  | val x -> (x, "")
```

Ker obnašanje učinkov določajo prestrezniki, v telesu funkcije `print_list` nimamo zagotovil o tem, kako bodo elementi seznamu izpisani. Lahko uporabimo prestreznik, ki podvoji vsak izpis ali pa zanemari določene izpise. Lahko predpostavimo, da bomo vedno uporabljali `collect_prints`, vendar s tem izgubimo fleksibilnost, ki nam jo prestrezniki omogočajo. V mnogih primerih so lastnosti, na katere se zanašamo, bolj splošne od konkretnih implementacij. Na primer, želimo, da ni pomembno, ali izpišemo \(x\) in nato \(y\), ali pa takoj izpišemo \(x \wedge y\). Takšne lastnosti lahko pogosto izrazimo v obliki enačb:

\[
E_{\text{print}} := (\text{!Print } x; \text{!Print } y \sim \text{!Print } (x \wedge y))
\]

Sistem tipov najprej nadgradimo, da sledi uporabi različnih učinkov v programih, kjer v tipu izračuna dodatno označimo, katere operacije se lahko kličejo med izvajanjem.

```
print_list : string list → unit\{Print\}
```

Ideja lokalnih učinkovnih teorij je dodatna nadgradnja, kjer v tipih dodatno sledimo želeni učinkovni teoriji.

```
print_list : string list → unit\{Print\}/E_{\text{print}}
```

Pri tem seveda za vse prestreznike preverimo, da tem teorijam res ustrezajo. Tako lahko `collect_prints` dodelimo spodnji tip zgolj, če ne loči med izračunoma \(\text{!Print } x; \text{!Print } y\) in \(\text{!Print } (x \wedge y)\).

```
collect_prints : A\{Print\}/E_{\text{print}} ⇒ (A × string)\{\}/\{}
```

### C.3 Jezik EEFF


- vrne vrednosti tipa \(A\);
- med izvajanjem lahko pokličejo operacije iz signature \(\Sigma\);
- njihovo enakost razumemo glede na teorijo \(E\).

Za zapis enačb uporabljamo *predloge*. Gradniki predlog predstavljajo podmnožico jezikov skupaj s posebnimi sprememljivkami \(z\), s katerimi označimo splošne programe, ki čakajo na vrednost. Omejitev enačb na predloge nam olajša preverjanje pravilnosti prestreznikov in omogoča uporabo v različnih tipih.

V enačbah uporabljamo izraze jezika, ki lahko vsebujejo označbe s tipi (npr. funkcije). Posledično potrebujemo hkratno definicijo za:

- vrednosti \(v\),
- izračune \(c\),
C.4 Sistem tipov

V poglavju 4 opišemo relacije, ki jih uporabljamo sistemu tipov. Zaradi prepletenosti potrebujemo hkratno definicijo sledečih relacij:

- \( \Gamma \vdash v : A \), v kontekstu \( \Gamma \) ima \( v \) tip \( A \);
- \( \Gamma \vdash c : C \), v kontekstu \( \Gamma \) ima \( c \) tip \( C \);
- \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( E \), v kontekstu \( \Gamma \) operacijske veje \( h \) pokrije operacije v \( \Sigma \) z uporabo izračuna tipa \( D \);
- \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( E \), v kontekstu \( \Gamma \) operacijske veje \( h \) pokrije operacije v \( \Sigma \) z uporabo izračuna tipa \( D \);
- \( \Gamma ; Z \vdash T : \Sigma \), v kontekstih \( \Gamma \) in \( Z \) je predlog \( T \) dobro definiran glede na \( \Sigma \);
- \( \vdash E : \Sigma \), enačbe \( E \) so dobro definirane glede na signaturo \( \Sigma \).

Ker lahko pravilnost operacijskih vej dokazamo zgolj v logiki, ki pogosto uporablja informacije o tipih, je zgornji hkratni definiciji potrebno dodati tudi vse relacije logike, o katerih govorimo kasneje. Pravila za konstrukcijo so zbrane v dodatku A.2.

Jezik EEFF vsebuje tudi podtipe, ki jih opišemo v razdelku 4.1. Razširitev s podtipe je pogosto težavna, vendar nam omogoča enostavnije programiranje, zato je pomembno, da preučimo interakcijo lokalnih teorij in podtipov. Podtipe razširimo tudi na enačbe, saj lahko izračun tipa \( A!\Sigma/E \) vedno obravnavamo kot izračun tipa \( A!\Sigma/E' \), čim \( E' \) vsebuje vse enačbe \( E \). Na enačbe moramo gledati kot na omejitve možnih implementacij, omejitve pa lahko vedno varno zaostrimo.

V razdelku 4.2 definiramo pravila za preverjanje dobre definiranosti tipov in predlog ter pravila za dodeljevanje tipov. Samih sprememb v dodeljevanju tipov je malo, pomembna pa je sprememba v pravilu za tipizacijo prestreznikov, kjer dodamo zahtevo \( \Gamma \vdash h : \Sigma \Rightarrow D \) respects \( E \), s katero poskrbimo za pravilnost prestreznika glede na \( E \).

\[
\begin{align*}
\Gamma, x : A &\vdash c_r : D & \Gamma \vdash h : \Sigma \Rightarrow D &\quad\Gamma \vdash h : \Sigma \Rightarrow D \text{ respects } E \quad\text{ s katero poskrbimo za pravilnost prestreznika glede na } E.
\end{align*}
\]

Razdelek 4.3 je namenjen dokazovanju lastnosti sistema tipov – na primer lema 4.3.5, ki govori o substituciji, in izrek o varnosti, ki poskrbi za ujemanje operacijske semantike in tipov.
Izrek (Varnost).

Ohranitev Če velja $\top \vdash c : C$ in $c \sim c'$, potem velja $\top \vdash c' : C$.

Napredek Če velja $\top \vdash c : A!\Sigma/\mathcal{E}$, potem bodisi

$\top \vdash c'$, za katerega velja $c \sim c'$, bodisi

je $c$ oblike rez $v$ za neko vrednost $v$, bodisi

je $c$ oblike $\text{op}_A \rightarrow_B (v; k)$ za neko operacijo $\text{op} \in \Sigma$.

Poglavje zaključimo z razdelkom 4.4 o uporabi predlog. Kot pove ime, lahko predloge pretvorimo v izračune oblike, ki jo določa predloga. Dobro definirane predloge lahko pretvorimo v izračune, ki jim lahko vedno dodelimo tip (Lema 4.4.1).

C.5 Logika

Za dokazovanje pravilnosti prestreznikov je potrebno sistem tipov povezati z logiko, v kateri lahko takšne dokaze konstruiramo. Izbiro logike ne vpliva na obliko pravil sistema tipov. Edina točka, kjer pri dodeljevanju tipa potrebujemo logiko, je pri konstrukciji relacije respects, kar dopušča precejšno fleksibilnost pri izbiri logike.

Prazna logika ne vsebuje nobenih logičnih pravil. Posledično ni mogoče konstruirati dokaza za $\Gamma \vdash \Sigma: D \Rightarrow h \text{ respects } \mathcal{E}$, torej v jeziku ne moremo tipizirati prestreznikov. Uporaba te logike reducira $\text{EFF}$ na jezik z učinki, ki pa (z izjemo učinkov s primitivnim pomenom) imajo zgolj vlogo signalov za prekinitev izvajanja.

Prosta logika vsebuje le pravilo za izgradnjo dokaza pravilnosti glede na prazne teorije.

$\Gamma \vdash \Sigma: D \Rightarrow h \text{ respects } \{\}$

Pri uporabi te logike jezik efektivno skrčimo na Eff, saj lahko uporabljamo prestreznike le na izračunih, ki ne privzamejo učinkovne teorije. Uporaba lokalnih teorij je torej direktna nadgradnja pristopov brez teorij.

Polna logika vsebuje pravilo, ki dokaže pravilnost poljubnega prestreznika.

$\Gamma \vdash \Sigma: D \Rightarrow h \text{ respects } \mathcal{E}$

To pravilo naredi logiko nezdravo, saj lahko dokažemo pravilnost prestreznikov, ki ne ustrezajo intuitivni ideji pravilnosti. Uporaba takšne logike je smiselna, kadar preverbo pravilnosti prestreznikov zaupamo uporabniku. Strogo gledano takšno logiko uporablja algoritem za izpeljavo tipov v implementaciji, kjer breme dokazov pravilnosti prepustimo programerju.

Logika z enačbami je primer enostavne logike za sklepanje o enakosti programov ter je predstavljena v razdelku 5.4 in dodatku A.3. Poleg respects vsebuje še pravila za tri relacije:

- $\Gamma \vdash v_1 \equiv_A v_2$ pove, da sta v kontekstu $\Gamma$ vrednosti $v_1$ in $v_2$ enaki pri tipu $A$.
- $\Gamma \vdash c_1 \equiv_C c_2$ pove, da sta v kontekstu $\Gamma$ izračuna $c_1$ in $c_2$ enaka pri tipu $C$.
- $\Gamma \vdash h_1 \equiv_{\Sigma ; \Rightarrow D} h_2$ pove, da v kontekstu $\Gamma$ operacijske veje $h_1$ in $h_2$ vse operacije signature $\Sigma$ pretvorijo v enake izračune tipa $D$.  

167
Logika vsebuje strukturna pravila za izraze, $\beta$-redukcije in $\eta$-razširitve. Za izračuna $c_1$ in $c_2$ lahko pri tipu $A\Sigma/E$ trdimo, da sta enaka, če za enačbo $T_1 \sim T_2 \in E$ izračun $c_1$ ustreza obliki predloge $T_1$ in izračun $c_2$ obliki $T_2$. Da pokažemo, da prestreznik spoštuje enačbo $T_1 \sim T_2$, moramo dokazati, da ne razlikuje med izračuni oblike $T_1$ in $T_2$.

**Predikatna logika** je predstavljena v razdelku 5.5 in dodatku A.4. Logiko z enačbami nadgradimo z logičnimi vezniki in vsebuje relacije: $\Box \Gamma \vdash \varphi$ pove, da formula $\varphi$ drži v kontekstu $\Gamma$, če privzamemo hipoteze $\Psi$. $\Box \Gamma \vdash \varphi$ zagotavlja, da je formula $\varphi$ dobro definirana v kontekstu $\Gamma$. $\Box \Gamma \vdash \Psi$ zagotavlja, da so vse hipoteze $\Psi$ dobro definirane v $\Gamma$.

Enakost izrazov je sedaj le ena izmed možnih formul. Glavna prednost predikatne logike je princip indukcije na izračunih [37, 6]. Pravilo `Induction` se opira na dejstvo, da izračuni ali vrnejo vrednost ali pokličejo operacijo ali pa se nikoli ne izvedejo do konca. Ker logika ne vsebuje univerzalnega kvantifikatorja za izračune tipa $C$, se obrnemo na ekvivalentno formulacijo indukcije za funkcije tipa $\text{unit} \rightarrow C$.

**C.6 Denotacijska semantika jezika**

Za razliko od pristopa z globalnimi teorijami [37], ki nepravilne prestreznike zanemari, pristopimo k problemu z dvostopenjsko denotacijsko semantiko. V poglavju 6 interpretiramo programe kot matematične objekte, kjer učinkovne teorije še ne igrajo vloge, denotacijsko semantiko teorij pa opišemo v poglavju 7.

Denotacije tipov in izrazov predstavimo v razdelku 6.2 in dodatku B.1. Ker jezik uporablja rekurzijo, za denotacije tipov uporabimo preddomene in domene (za razliko od predhodnjega dela [28], kjer uporabljamo množice).

\[
\begin{align*}
\text{⟦unit⟧} &= \{\star\} \\
\text{⟦int⟧} &= \mathbb{N} \\
\text{⟦empty⟧} &= \emptyset \\
\text{⟦A × B⟧} &= \text{⟦A⟧} × \text{⟦B⟧} \\
\text{⟦A + B⟧} &= \text{⟦A⟧} + \text{⟦B⟧} \\
\text{⟦A list⟧} &= \text{⟦A⟧}^* \\
\text{⟦A → C⟧} &= \text{⟦A⟧} → \text{⟦C⟧} \\
\text{⟦C ⇒ D⟧} &= \text{⟦C⟧} ⊸ \text{⟦D⟧}
\end{align*}
\]

Za konstrukcijo domene, ki jo uporabimo za interpretacijo tipov izračunov, uporabimo

\[
\Sigma \Gamma \Gamma \rightarrow \Psi \Rightarrow \varphi = T_{\Sigma}(A, (\text{⟦A op⟧})_{op \in \Sigma}, (\text{⟦B op⟧})_{op \in \Sigma}),
\]

kjer je $T_I(A, (A_i), (B_i))$ rešitev enačbe

\[
F(D) = \left( A + \prod_{i \in I} A_i \times (B_i \rightarrow D) \right)_\bot.
\]

Operacijske veje tipa $\Sigma \Rightarrow D$ predstavimo z družino funkcij, ki za vsako $op : A \rightarrow B \in \Sigma$ vsebuje funkcijo $H_{op} : \text{⟦A⟧} × (\text{⟦B⟧} \rightarrow \text{⟦D⟧}) \rightarrow \text{⟦D⟧}$. Konstrukcijo lahko posplošimo na

\[
\text{interp}_\Sigma(Y) = \prod_{op : A \rightarrow B \in \Sigma} \text{⟦A⟧} × (\text{⟦B⟧} \rightarrow Y) \rightarrow Y.
\]

Kontekste predstavimo kot produkt elementov, ki bi jih spremenljivke lahko zasedle. Podobno storimo za kontekste predlog, le da je potrebno dodatno podati domeno $Y$, v katero elementi slikajo.

\[
\begin{align*}
\text{⟦·⟧} &= \{\star\} \\
\text{⟦Γ, x : A⟧} &= \text{⟦Γ⟧} \times \text{⟦A⟧} \\
\text{⟦Z, z : A → *⟧} &= \text{⟦Z⟧} \times Y^{[A]}
\end{align*}
\]
Zanimajo nas le izrazi, ki jim lahko priredimo tip, zato je najlažje, da denotacije izrazov zgradimo prek izpeljave tipa.

\[
\begin{align*}
\Gamma \vdash v : A & : \Gamma \rightarrow [A] \\
\Gamma \vdash c : C & : \Gamma \rightarrow [C] \\
\Gamma \vdash h : \Sigma \Rightarrow D & : \Gamma \rightarrow \text{interp}_\Sigma([D])
\end{align*}
\]

Zaradi podtipov imamo več možnosti za konstrukcijo \(\Gamma \vdash v : A\), vendar se izkaže, da za denotacijo \(\Gamma \vdash v : A\) ni pomembno, katero izberemo. To pokažemo v razdelku 6.4, kjer preidemo na jezik skeletov, v katerem je izpeljava tipa enolična. Z uporabo logične relacije, ki povezuje denotacije izrazov z denotacijami skeletov, v trditvi 6.4.9 pokažemo koherentnost denotacijske semantike.

V razdelku 6.5 se posvetimo še drugim lastnostim denotacijske semantike, kot na primer lema 6.5.2 o interakciji s substitucijo. Pokažemo tudi izrek 6.5.3 o zdravosti, ki zagotavlja pravilnost prestreznika v denotacijski semantiki. Kadar sistem tipov uporablja kar obravnavamo v razdelku 7.3. Pravimo, da je logika \(~\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)~ vrsta logike, če \(\Sigma \Rightarrow D\) respekteva \(\Sigma\) in formalizacije denotacijske logike.

Relacija \(\sim_{\Sigma, E}\) poleg strukture elementov upošteva tudi enačbe \(E\). Če \(E\) vsebuje enačbo \(T_1 \sim T_2\), in sta \(c_1\) in \(c_2\) elementa \([C]\), ju povežemo z \(\sim_C\), če lahko \(c_1\) in \(c_2\) dobimo z uporabo \([T_1]\) in \([T_2]\).

Za denotacije prestreznikov velja \(h \sim_C D h'\) natanko tedaj, ko iz \(c \sim_C c'\) sledi \(h(c) \sim_D h'(c')\), torej \(h\) in \(h'\) slikata ekvivalentne elemente v ekvivalentne rezultate. Intuitivno gledano \(h \sim_C D h\) zagotavlja, da je \(h\) pravilen glede na enačbe tipa \(C\). S tem namenom uporabljamo delne ekvivalentne relacije, saj \([C] \Rightarrow D\) vsebuje tudi prestreznike, ki niso pravilni.

Interakcija med semantiko izrazov in semantiko učinkov teorij je odvisna od logike, kar obravnavamo v razdelku 7.3. Pravimo, da je logika \(zd\) ljudska, če \(h : \Sigma \Rightarrow D\) respects \(E\) in formalizira pravilnost prestreznika v denotacijski semantiki. Kadar sistem tipov uporablja zdravo logiko, iz \(\Gamma \vdash v : A\) sledi \([\Gamma \vdash v : A]\eta \sim_A [\Gamma \vdash v : A]\eta'\), čim velja \(\eta \sim \eta'\) (podobna lastnost velja za izračune). Pokažemo tudi, da sta logika z enačbami in predikatna logika iz poglavja 5 zdravi, ter s protiprimerom pokažemo, da polna logika ni zdrava.

V razdelku 7.4 orišemo izpeljava dokaza zadostnosti denotacijske semantike. Pokažemo tudi, da sta vrednosti \(v_1\) in \(v_2\) kontekstno ekvivalentni, če velja \(v_1 \equiv v_2\).

**Trditev (Zadostnost).** Če velja \([\cdot \vdash c : \text{unit!}/\{\}])\sim_{\text{in}_{\text{val}}(\bullet)}\), velja tudi \(c \sim^{*} \text{ret}()\).

### C.7 Semantika teorij učinkov

Drugi del semantike jezika \(EEFF\) je semantika teorij učinkov, ki jo predstavimo v poglavju 7. V razdelku 7.1 definiramo semantiko predlog \(T\), nato pa v razdelku 7.2 za vsak tip \(A\) konstruiramo delno ekvivalentno relacijo \((\sim_A) \subset [A]\times [A]\). Konstrukcija relacije je odvisna od tipa. Na primer, za \(\text{int}\) vzamemo identično relacijo, relacija \(\sim_{\text{int}}\) pa je definirana po komponentah.

Relacija \((\sim_{\Sigma, E})\) poleg strukture elementov upošteva tudi enačbe \(E\). Če \(E\) vsebuje enačbo \(T_1 \sim T_2\), in sta \(c_1\) in \(c_2\) elementa \([C]\), ju povežemo z \(\sim_C\), če lahko \(c_1\) in \(c_2\) dobimo z uporabo \([T_1]\) in \([T_2]\).

Za denotacije prestreznikov velja \(h \sim_C D h'\) natanko tedaj, ko iz \(c \sim_C c'\) sledi \(h(c) \sim_D h'(c')\), torej \(h\) in \(h'\) slikata ekvivalentne elemente v ekvivalentne rezultate. Intuitivno gledano \(h \sim_C D h\) zagotavlja, da je \(h\) pravilen glede na enačbe tipa \(C\). S tem namenom uporabljamo delne ekvivalentne relacije, saj \([C] \Rightarrow D\) vsebuje tudi prestreznike, ki niso pravilni.

Interakcija med semantiko izrazov in semantiko učinkov teorij je odvisna od logike, kar obravnavamo v razdelku 7.3. Pravimo, da je logika \(zd\), če \(h : \Sigma \Rightarrow D\) respects \(E\) in formalizira pravilnost prestreznika v denotacijski semantiki. Kadar sistem tipov uporablja zdravo logiko, iz \(\Gamma \vdash v : A\) sledi \([\Gamma \vdash v : A]\eta \sim_A [\Gamma \vdash v : A]\eta'\), čim velja \(\eta \sim \eta'\) (podobna lastnost velja za izračune). Pokažemo tudi, da sta logika z enačbami in predikatna logika iz poglavja 5 zdravi, ter s protiprimerom pokažemo, da polna logika ni zdrava.

V razdelku 7.4 orišemo izpeljava dokaza zadostnosti denotacijske semantike. Pokažemo tudi, da sta vrednosti \(v_1\) in \(v_2\) kontekstno ekvivalentni, če velja \(v_1 \equiv v_2\).

**Trditev (Zadostnost).** Če velja \([\cdot \vdash c : \text{unit!}/\{\}])\sim_{\text{in}_{\text{val}}(\bullet)}\), velja tudi \(c \sim^{*} \text{ret}()\).

### C.8 Implementacija in formalizacija

Poglave 8 vsebuje kratak opis implementacije\(^1\) in formalizacije\(^2\) jezika \(EEFF\).

\(^{1}\)https://github.com/zigaLuksic/eff/tree/EEFF
\(^{2}\)https://github.com/zigaLuksic/eeff-formalization
Sistem tipov, ki ga predstavimo v poglavju 4, je priročen za analizo, vendar neprimerno za implementacijo. Zato v razdelku 8.1 predstavimo dvosmerni sistem tipov, ki ga lahko uporabimo kot algoritrem za izpeljavo tipov. Dvosmerni sistem uporablja dve vrsti tipizacije:

- **preverjanje** $\Gamma \vdash v \iff A$, kjer sta $v$ in $A$ podana.
- **sinteza** $\Gamma \vdash v \Rightarrow A$, kjer je $v$ podan in je $A$ rezultat sinteze.

Uporaba dvosmernega sistema tipov zahteva označevanje tipov na določenih mestih programa, saj za nekatere izraze sinteza ni možna. Učinkovne teorije v vsakem primeru poda uporabnik, zato je označevanje v vsakem primeru nujno.

Implementacija jezika EEFF tako uporablja dva sistema tipov, ki pa sta si na srečo zelo podobna. V grobem velja, da če vrednosti $v$ v kontekstu $\Gamma$ algoritem izpelje tip $A$ v dvosmernem sistemu tipov, potem lahko v sistemu tipov iz poglavja 4 pokažemo $\Gamma \vdash v : A$.

V razdelku 8.2 opišemo še preostanek implementacije. Dodali smo naprednejso uporabo vzorcev in možnost definiranja rekurzivnih tipov. Ker temelji jezika že vsebujejo produkte, vsote in sezname, ne pričakujemo dodatnih zapletov z razširitvama. V jeziku sicer uporabljamo lokalne signature učinkov, kar pa postane okorno pri pisanju signatur z več učinki. Zato smo v implementaciji dodali možnost globalne označbe, ki se uporabi v primeru, da operaciji v signaturi ne določimo tipa.

Učinkovne teorije in signature se v implementaciji obravnavajo kot enotna komponenta. Vsako teorijo definiramo pri točno določeni signaturi, saj v primeru, da bi bilo potrebno signaturo povečati, za to poskrbimo z uporabo podtipov.

```plaintext
theory eqn_comm for {Choice : int * int -> int} is
{ x:int, y: int ; z: int -> * |-
Choice((x,y); w.z w) ~ Choice((y,x); w.z w ) }
```

Da ni potrebno ponovno pisati enakih enačb, lahko pri definiranju nove teorije vključimo vse enačbe že obstoječe teorije. Sistem tipov nato preveri, ali sta teoriji združljivi. Kot že omenjeno, implementacija ne preverja pravilnosti prestreznikov, poskrbi pa za pravilno uporabo teorij v programu.

Za dokazovanje pravilnosti lahko uporabimo formalizacijo, ki jo opišemo v razdelku 8.3. Definicije jezika so zelo prepletene, kar zahteva dodatno previdnost pri dokazovanju trditeljev. Dokazovalniki so izvrstno orodje za takšne probleme, zato smo EEFF formalizirali v dokazovalniku Coq. Z izjemo poglavja 6 in poglavja 7 so vsi dokazi in primeri preverjeni z dokazovalnikom. Formalizacija zajema:

- **EEFF**, sklopljen z logiko z enačbami,
- **EEFF**, sklopljen s predikatno logiko,
- dokaze o pravilnosti dvosmernega sistema tipov.

Prvotna naloga formalizacije je odkrivanje napak jezika, vendar jo lahko uporabljamo tudi kot orodje za sklepanje. Pri pisanju dokazov za pravilnost prestreznikov v formalizaciji je tretnato precej dela posvečenega določanju tipov, kar bi lahko v nadaljevanju poenostavili z uporabo močnejših taktik v dokazovalniku.

V večini poglavij privzamemo različne trditve o substituciji, vendar jih ne dokazemo, saj so dokazi tehnično zahtevni. Formalizacija jezika EEFF s predikatno logiko vsebuje čez 11000 vrstic kode, približno 7000 vrstic pa je namenjenih zgolj substituciji. Dokazovanje trditeljev o substituciji zahteva veliko dela, vendar pri tem pogosto odkrijemo napake v jeziku, ki bi jih sicer spregledali.

170
C.9 Zaključek

Lokalne učinkovne teorije nudijo dodatne možnosti pri dokazovanju lastnosti programov brez globalnega omejevanja prestreznikov. Novost pristopa je nadgradnja tipov, ki sedaj vsebujejo podatke o učinkovni teoriji. Sistem tipov poskrbi za sledenje lokalno privzetim teorijam, kar nam omogoča uporabo različnih teorij za iste učinke, ki jih lahko tudi gnezdimi s primerino uporabo prestreznikov. Lokalne teorije se brez večjih zapletov razširijo s pogostimi nadgradnjami kot so pari, vsole, seznami, rekurzija in podtipi.

Za dokazovanje pravilnosti prestreznikov je sistem tipov povezan z logiko. Na voljo imamo več izbir za logiko, ta izbira pa vpliva na lastnosti jezika. Definirali smo tudi zdravost logike, ki je zadosten pogoj, da pravilni prestrezniki ne razlikujejo med ekvivalentnimi izračuni. Jezik \textit{EEFF} smo opremili z dvostopenjsko denotacijsko semantiko, kjer izraze interpretiramo kot matematične objekte, učinkovne enačbe pa kot delne ekvivalenčne relacije.

Implementacija jezika \textit{EEFF} obsega tudi razširitev, ki niso bile obravnavane v disertaciji, s čimer do sedaj ni bilo zapletov. Za izpeljavo tipov uporabljamo dvosmerni sistem tipov, ki poskrbi za varno uporabo učinkovnih teorij. Na žalost izpeljava tipov ne avtomatizira dokazov pravilnosti, ki so zato prepuščeni uporabniku. S pomočjo formalizacije v dokazovalniku Coq smo dokazali varnost sistema tipov. Uporabili smo jo tudi za dokaz pravilnosti primerov v poglavju 5, s čimer smo pokazali uporabnost formalizacije kot orodja za sklepanje.

Primerjava s sorodnimi deli

Izvorni pristop

Začetna dela na prestreznikih učinkov [38, 37] že predstavijo nekatere ideje, ki jih uporabljamo in nadgradimo v jeziku \textit{EEFF}. Vsakemu učinku pripada učinkovna teorija podana z enačbami, vendar so teorije globalne, kar fiksira teorijo za učinke v vseh delih programa. Pravilnost prestreznikov se dokazuje v logiki, ki dodatno ponuja tudi logične neenakosti za delo z divergenco. Za enačbe se uporabljajo predloge, ki pa ne dopuščajo uporabe funkcij z izjemo nekaterih vnaprej določenih primitivnih funkcij. To smo v jeziku \textit{EEFF} nadgradili s predlogo do pure. Denotacijska semantika izvirnega pristopa upošteva teorije, vendar preprosto zanemari vse prestreznike, ki ne spoštujejo teorije. Ta problem smo zaoblili z dvostopenjsko denotacijsko semantiko, kjer imajo vsi prestrezniki denotacijo, vendar so le pravilni prestrezniki vključeni v delne ekvivalenčne relacije.

Učinki brez enačb


Definirane algebrajske operacije

Definirane algebrajske operacije (DAO) [18] so alternativni pristop k algebrajskim učinkom, ki reši nekatere težave kot imeni operacij. Z izjemo mehanizmov za imenovanje operacij so razlike med DAO in prestrezniki majhne, zaradi česar so tehnike sklepanja za DAO toliko bolj relevantne. Za razliko od denotacijsko usmerjenih pristopov DAO uporablja logiko, ki upošteva učinkovne teorije, in dovoljuje postopno nadgradjevanje logike v dokazih. S pomočjo enačb tudi omogoča delitev sklepanja na sklepanja o kodi in dokazovanje, da implementacija spoštuje teorijo. Podobno kot za jezik EEFF se denotacijska semantika DAO zanaša na delne ekvivalenčne relacije. Teorije so omejene zgolj na logiko, ki je ločena od sistema tipov, zato pristop ne omogoča omejevanja možnih implementacij učinkov na določeno teorijo.

Odvisni tipi

Teorije učinkov lahko uporabljamo tudi v povezavi z odvisnimi tipi [2, 3]. Pri algebrajskih učinkih in prestreznikih, prilagojenih na odvisne tipe, zdraznost zagotovimo s posebnimi tipi za algebre, ki nosijo dokaze o pravilnosti prestreznikov. Pristop uporablja širok nabor tehnik za sklepanje in omogoča oblikovanje, razlikovanje učinkov, s čimer v dokazih ne potrebujemo kode prestreznikov. Uporaba odvisnih tipov izboljša nekatere ideje izvornega pristopa, vendar teorije ostajajo globalne.

Sklepanje z učinkovnimi drevesi

Če odmislimo prestreznike in se v celoti posvetimo algebrajskim učinkom, lahko uporabljamo orodja kot so modalnosti in vedenjske ekvivalence [43, 47]. Namesto da teorijo določimo z enačbami, se vedenje učinkov fiksira z izbiro modalnosti, ki so prilagojene posameznim učinkom. Modalnosti delujejo na učinkovnih drevesih in lahko izražajo lastnosti kot “vsak možen izračun se izvede do konca”, česar trenutne logike za EEFF ne omogočajo. Dodatna prednost je dokazovanje neenakosti programov s tem, da podamo logično lastnost, ki med programoma razlikuje. Pristop ima po mnenju avtorjev članka potencial kot del nizko-nivojskega jezika, v katerega prevajamo.

Pristopi z monadami

Modeliranje računskih učinkov z monadami prav tako omogoča uporabo teorij učinkov. Pristop, ki sta ga opisala Gibbons in Hinze [20, 19], je služil kot navdih za stil sklepanja, ki smo ga želeli doseči z jezikom EEFF. Z dodatnimi zahtevami za implementacije monad pridobimo nove možnosti za sklepanje, ki poteka na nivoju programskega jezika in ne potrebuje denotacijske semantike. Zahteve za monade so enakovredne pravilnosti prestreznikov, kar vidimo iz primerjave zahtev za monado pomnilnika [19, Example 2.4] in prestreznika za pomnilnik (primer 5.4.13).

\[
\begin{align*}
\text{set} \ s \gg \text{get} & \gg \lambda s' \to k \ s' = \text{set} \ s \gg k \ s \quad \text{zahteva za MonadState} \\
\text{Set}(s; \ _\text{Get}(((); \ s'.z \ s'))) \sim \text{Set}(s; \ _\text{z} \ s) \quad \text{enačba prestreznika za pomnilnik}
\end{align*}
\]

Gibbons in Hinze podata mnogo primerov uporabe, vse od teorije pomnilnika do reševanja problema Monty Hall. Na žalost zahteve za implementacijo monad niso direkten del programskega jezika. Tipi programa posledično ne določajo teorije.
Prístopi z avtomatizacijo


Nadaljnje delo

Raziskovanje možnosti

Za nadaljnji razvoj jezika EEFF je ključnega pomena, da se osredotočimo na možne uporabe. Kot kaže primer 5.5.12, je pristop uporaben tudi pri bolj zapletenih problemih in ne zgolj pri umetno zastavljenih primerih. S proučevanjem primerov uporabe lažje odkrijemo pomanjkljivosti pristopa.

Delo na optimizaciji kode z učinki [24] se morda lahko nadgraditi z optimizacijami, ki so varne zgolj v nekaterih učinkovnih teorijah. Sistem tipov nato poskrbi, da se takšne optimizacije uporabljajo zgolj v delih programa, kjer ta učinkovna teorija velja. Dodatna priložnost so tudi področja, ki uporabljajo algebrajske učinke in se zanašajo na lastnosti učinkov. Primer takšnega področja je probabilistično programiranje [42, 46, 9].

Pomembno je tudi raziskati, katere lastnosti programov je možno izraziti z enačbami. Primer 5.5.12 uporablja pogojno enačbo, ki opisuje vedenje učinkov pod določenimi pogoji. Vključitev nove predloge do pure omogoča nove enačbe, ki jih je vredno raziskati.

Polimorfizem

Za splošno uporabo je potrebno EEFF nadgraditi tudi s polimorfizmom. Pri podtipih je interakcija z učinkovnimi teorijami intuitivna, pri polimorfizmu pa je situacija bolj zapletena. Naraven pristop bi bil polimorfizem v vseh komponentah, na primer

\[
\text{map} : \forall \alpha, \beta, \sigma, \varepsilon (\alpha \rightarrow \beta!\sigma/\varepsilon) \rightarrow (\alpha \text{ list} \rightarrow \beta \text{ list}\!\sigma/\varepsilon)!\{\}/\},
\]

vendar komponente tipov med seboj niso popolnoma neodvisne. Morda je bolj smiselno, da se signature in teorije združi v enoto komponento. Hkrati moramo zagotoviti tudi primerno logiko za dokazovanje pravilnosti prestreznikov, ki je prilagojena za polimorfizem. Pri razširitvi implementacije si lahko pomagamo z napredki pri dvosmernih izpeljavah za polimorfne tipe [15].

Pred razširitvijo s polimorfnimi tipi si je morda vredno ogledati odprte prestreznike, kjer se klici operacij, za katere prestreznik nima primerne veje, širijo navzven. To lahko delno doseže v jeziku EEFF tako, da prestreznike vsakič primerno dopolnimo, vendar nastane problem z dokazi, ki so konstruirani ob natančnih tipih. Če mora uporabnik ponovno dokazati pravilnost vsakič, ko je takšna razširitev potrebna, hitro nastane preveč dela. Bolj zanimivo vprašanje je, ali lahko nadgradimo logiko z dokazi za odprte prestreznike z manj truda, kot bi ga zahteval prehod na polimorfne tipe.

Izboljšave logike in orodij

Edina zahteva za logiko je, da vsebuje relacijo respects za dokazovanje pravilnosti prestreznikov. Zdravost logike je zaželena, vendar ne nujna, saj v implementaciji algoritem za izpeljavo tipov formalno gledano uporablja nezdravo logiko. S tako ohlapnimi zahtevami
imamo na voljo širok spekter možnosti, vse od razširjanja logik z novimi pravili (morda indukcija na seznamih) pa do bolj eksotičnih opcij. Zanimiva bi bila uporaba orodja kot je QuickCheck [12], ki omogoča avtomatizirane teste pravilnosti. Takšni sistemi ne zagotovijo pravilnosti, vendar lahko poiščejo napake, še preden uporabnik prične s pisanjem dokaza. Druga možnost je uporaba SMT reševalnika za dokaze, podobno kot pri projektu F* [29].

Druga smer izboljšav pa so naprednejša orodja za pisanje dokazov. Formalizacija jezika EEFF je v prvi meri namenjena odkrivanju napak, vendar se lahko uporablja tudi za pisanje dokazov in sklepanje o programih. Izboljšave taktik in delna avtomatizacija bi vsekakor olajšale delo s formalizacijo kot orodjem za sklepanje. Prilagodi se lahko tudi implementacijo, in sicer tako, da algoritem za izpeljavo tipov proizvede Coq datoteko z vsemi trditvami, ki jih mora uporabnik dokazati.