DEFINED ALGEBRAIC OPERATIONS

by

Bram Geron

A thesis submitted to the University of Birmingham for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

University of Birmingham

Submitted April 2019

Corrected version December 2019

Abstract

Defined algebraic operations ("DAO") is a novel model of programming, which sits broadly between imperative and purely functional programming. DAO expresses many control-flow idioms in a fashion similar to algebraic effect handling. But operation definition is lexical, and commutes with sequencing (when that type-checks).

DAO has three particular strengths. Firstly, DAO automatically avoids name clashes when writing higher-order programs with nonlocal control. This is demonstrated with a simple example. Secondly, certain buggy programs do not type check due to the lexical nature of DAO.

Thirdly, it validates a strong "theory-dependent" logic, which uses properties of operation definitions to add equivalences inside their scope. For instance, under an operation definition for state, most state equations hold, even under lambdas — the analogous statement for handling is false. This lends extra credibility to the claim that DAO is a form of user-defined effects.

To substantiate these claims, we give a concrete DAO language and logic based on the fine-grain call-by-value lambda calculus, and a number of examples. As an additional contribution, we give a simpler presentation of a method for proving coherence of denotational semantics, first presented in a more sophisticated setting by Biernacki and Polesiuk.

Dedication

To my parents and my brother Koen. To my stepmother and stepsiblings, and their children.

To the beautiful memory of Paul Koekelkoren.

Acknowledgements

I am deeply indebted to my supervisor, Paul Blain Levy, who taught me so many things that ended up enriching my life. I admire the depth to which he understands things, and his efforts to tease out the essential mathematical beauty hidden in things that others leave ad hoc or imprecise. His focus of study is much more fundamental than mine, but when I showed up wanting to make programming more elegant, he gave me the space and he has been tremendously helpful. The end result is a thesis that is both holistic and rigorous, for which I cannot be more grateful. Plus, Paul has a wonderful sense of humour.

Two people in particular inspired my interest for programming languages. The first person is Paul Graham, who wrote bold essays claiming that some languages were objectively better than others, in particular Lisp was better than Java, in part because it is so functional (albeit not purely functional). He dared people to think outside the box and make the world a better place through better programming languages. The second person is James Hague with his essays on *programming in the twenty-first century*, which take a pragmatic view on languages and programming techniques. He convinced me that local side-effects are much more benign than global side-effects, which this PhD thesis attempts to lay out in more detail.

I would like to thank all the people in the Computer Science Theory group at Birmingham. They are wonderful people, friendly, who do research into a wide range of topics between mathematics and computer science, yet they take the time to teach and understand each other, and all the fascinating ways in which their research fields overlap. It has been a privilege to be part of this group during my PhD.

Last but not least, I would like to thank all the friends I made during my stay in Birmingham, both within the School and beyond. Without you, I would not have made it through — you made my stay in Birmingham into a joy.

SHORT CONTENTS

CONTENTS

Chapter 1

INTRODUCTION

1.1 Chapter overview

Defined algebraic operations or **DAO** is a new model of programming. It falls broadly between imperative and purely functional programming, combining lexical binding with effect handling to hopefully make it easier to write correct programs and harder to write bugs. In this chapter, we set the scene:

- In §1.2, we look at both **imperative programming and purely functional programming** in the context of a concrete program, and we show how both **are unsatisfactory** for this program in a different way. We give a broad sketch how the concrete program could be written using *defined algebraic operations*, and have better characteristics than can be achieved using either imperative or purely functional programming.
- In §1.3, we look into **lexical binding**: what it is and why it can be useful. We start the section with an example of why programs with exception handling can be awkward. We sketch a lexical counterpart to exceptions and exception handling, which is better for the example. The example illustrates **the difference between DAO and algebraic effect handling**: lexical binding.

  In contrast to handling, defined operations commute with sequencing.

- In §1.4, we **generalize raising exceptions to raising operations**. The difference is that raising an operation may return in a number of ways, according to the arity of the operation. Exceptions are merely operations that may return in 0 ways when raised.

  This part is completely analogous to algebraic effect handling, with the caveat that in this thesis we only consider n-ary operations, not operations with data.

- In §1.5, we give an example of defined algebraic operations in the actual formal lambda-like calculus that we will study in Chapters 6 and 7. The results here use the theory-dependent logic that will be developed in Chapter 7, and the proofs are in §8.4. *For readers familiar with algebraic effect handlers, this section will be familiar-looking, perhaps misleadingly so.*

- In §1.6, we present the syntax of this language informally. For the formal definition, we defer to Chapter 6.

- In §1.7, we show how with *defined algebraic operations*, you can reason more easily and powerfully than has been the case with *algebraic effect handling*. Our main example is that the get-get equation holds for the obvious "state-like" defined algebraic operations. We show what this means.

- In §1.8, we summarize how we further develop our arguments in the rest of the thesis.

1.2 How should information flow between components?

1.2.1 Imperative programming can be unsatisfactory

Software engineering is in a bit of a pickle. As computer programs are growing ever larger, it is not clear how to organise a large and growing codebase successfully. A particularly hairy question is how information should flow within a software component, when control flow is driven by another component — in particular when user requirements are not clear upfront. A very common but unsatisfactory approach is to encapsulate state in mutable references. This approach seems particularly common in object-oriented circles, but is not limited to them.

For an example, consider voting machine software for choosing between two candidates, A and B. Planning a division into components, it seems reasonable to consider

- a component for adding votes for a single candidate,
- a component for soliciting votes from the user,
- other components as we see fit.

Let us consider the vote-adding component. It could perhaps be in the form of a `Candidate` class, which contains an integer field `numVotes` that starts out at 0. When a new vote is registered, then `numVotes` is incremented.

The program might start by creating two `Candidate` objects. Then as the votes come in, the votes are counted in the respective `Candidate` objects. Finally, we find the candidate whose `Candidate` object contains the maximum number. In pseudocode:

```
def main() {
    // Initialize candidates.
    let candidateA = new Candidate();
    let candidateB = new Candidate();
    // Apply votes.
    for every incoming vote {
        if vote.isForA() {
            candidateA.countVote()
        } else {
            candidateB.countVote()
        }
    }
    // Gather results; return from main.
    return (candidateA.votes(), candidateB.votes())
}
```

Here, each `Candidate` value is a mutable reference.

The problem with this program

This particular program works, but it breaks easily in subtle ways. Here's one way in which it is fragile. Suppose that we decide that we want to make snapshots of the current status. We might naively decide to create a fresh tuple `(candidateA, candidateB)`. Unintentionally, this is a tuple of the *same objects* — rather than a tuple of the contents of the objects. They are the same instances behind the scenes, and the snapshot is updated in tandem with the originals.

The underlying problem here that we are using *mutable references*. Such "imperative" programs compose easily *syntactically*, but the compositions often do not have the result imagined by an inattentive programmer.

1.2.2 Purely functional programming: abolish mutable state?

Some people see the solution in *purely functional programming*, a style of programming which bans mutable references.[1] Avoiding mutable references is nice from many points of view. If values never change, then they do not need to be copied to protect against unexpected modification/aliasing by a different component. The external behaviour of values and programs is drastically simplified.

An argument against the purely functional style is that it seems to make top-down design rather difficult. Let us consider how information flows between software components. In the purely functional style, all the information flows must follow function call/return edges. (Implicit information transfer can happen with mutable references, but we outlawed those.) Therefore in designing a program's control flow, we must be aware of all information flow that

[1] Programming in purely functional style is often done in *purely functional languages* such as Haskell [51]. But one can essentially avoid mutable references in any language, by only creating and initializing values, and not modifying them after their creation.

Purely functional languages tend to contain features that work particularly well when using a purely functional style, such as *laziness*. But they are no guarantee of the purely functional style: Haskell does also contain functionality to explicitly support programming with mutable references.

```
let main() = (
    (* Initialize candidates. *)
    let candA_begin = newCandidate();
    let candB_begin = newCandidate();
    (* Apply votes. *)
    let applyVote : (Vote, (Candidate, Candidate)) -> (Candidate, Candidate)
                  = function (vote, (candA, candB)) -> (
        if vote.isForA() then
            (candA.plusExtraVote(), candB)
        else
            (candA, candB.plusExtraVote())
    );
    let (candA_end, candB_end) =
        fold(incomingVotes, (candA_begin, candB_begin), applyVote)
    (* Gather results; return from main. *)
    return (candA_end.votes(), candB_end.votes())
);;
```

Figure 1.1: Pseudocode for the voting program in purely functional style (without mutable references). The organisation of the source code has needed to change quite significantly to allow for purely functional style. Syntax is in the style of OCaml. We cannot change the `Candidate` objects in a for-loop, because they are not mutable. Instead, `Candidate` objects have a `.plusExtraVote()` method that creates a version of the candidate with an extra vote counted.

We use a combinator (`fold`) to iterate over a sequence of votes, and find the `Candidate` values that result from using `applyVotes` to applying each vote to the respective candidate with `.plusExtraVote()`.

may happen in a program, and design the control flow to facilitate the information flow. This is infeasible in practice when designing larger programs.

In our view, *purely functional programming* is about imposing restrictions. On the one hand,

these restrictions are good and add a large amount of useful structure to programs. On the other hand, they are so restrictive that it is infeasible to write large programs using them. We feel confirmed in this view by a dearth of industrial use of purely functional programming, even though the ideas have been around for multiple decades.

In Figure 1.1 on the preceding page, we show pseudocode for the voting program in purely functional style. The program had to be restructured to facilitate the information flows more explicitly.

1.2.3 Information flows that can skip over stack frames

This thesis investigates *defined algebraic operations*, a variation of a programming paradigm called "algebraic effects and handlers", which is a way of admitting a form of well-behaved user-definable statements. A chief use case of this paradigm is to *insert custom stack frames* which can maintain state and change state, and which can interact with deeper stack frames. However, the state in these stack frames does not possess identity in the sense of conventional heap-based state. As such, it can sidestep problems with aliasing, while still allowing relatively natural control flow. We elaborate on this below.

As an example, we present in Figure 1.2 on the next page a comparable program for counting votes using an envisioned language based on *defined algebraic operations*. It is built around a user-defined statement `withCandidateStartingAt`, which adds a stack frame storing integer state. We can interact with the state in the stack frame using operations `.countVote` (which increments the state) and `.votes` (which obtains the state). The resulting program feels rather similar to the imperative program in §1.2.1, while mitigating some of its problems.

Remark. We do not give a definition of `withCandidateStartingAt` here, but a variation of the syntactic algebra for binary state in §8.4 would be suitable.

An important limitation from the type system and grammar

The grammar and type system play an important role in using this language effectively. For this high-level language we do not detail a type system, but we can write an analogous program

```
def main() {
    // Initialize candidates.
    withCandidateStartingAt(0) {candidateA.
        withCandidateStartingAt(0) {candidateB.
            // Apply votes.
            for every incoming vote {
                if vote.isForA() {
                    candidateA.countVote()
                } else {
                    candidateB.countVote()
                }
            }
            // Gather results; return from main.
            return (candidateA.votes(), candidateB.votes())
        }
    }
    // Here, candidateA and candidateB have become unavailable. Indeed, the
    // corresponding stack frames are gone. All values whose static type
    // mentions candidateA or candidateB are also gone -- indeed, the type
    // system enforces that any value visible from this point is typeable
    // without mentioning candidateA or candidateB.
}
```

Figure 1.2: Pseudocode for the voting program in an envisioned language, using a user-defined `withCandidateStartingAt` statement based on defined algebraic operations. The organisation of the source code resembles for a large part that of the imperative program in §1.2.1, yet like pure FP, this style avoids mutable references and the aliasing problems that come with them. Instead, the `.countVote()` and `.votes()` calls interact with the stack frames created by the corresponding `withCandidateStartingAt`, which stores temporary mutable state that goes away after the end of that `withCandidateStartingAt`.
It is a syntactic error to refer to the stack frames from outside of those stack frames, or to persist values that indirectly refer to these stack frames — indeed, the stack frames no longer exist. This limitation prevents the class of errors caused by aliasing.

in the elaborated lambda-like calculus in Chapter 6, where there would be functions for each `.countVote()` and functions for each `.votes()`. Writing the function names in italic font, their types are:[2]

$$
\begin{aligned}
\mathit{candidateA.countVote} &: 1 \to 1\,!\,\{\underline{\text{candidateA.countVote}}\} \\
\mathit{candidateB.countVote} &: 1 \to 1\,!\,\{\underline{\text{candidateB.countVote}}\} \\
\mathit{candidateA.getVotes} &: 1 \to \text{nat}\,!\,\{\underline{\text{candidateA.getVotes}}\} \\
\mathit{candidateB.getVotes} &: 1 \to \text{nat}\,!\,\{\underline{\text{candidateB.getVotes}}\}
\end{aligned}
$$

That is, they are functions that accept a unit argument and return unit or nat, while potentially accessing the indicated stack frame. Here $\underline{\text{candidateA.countVote}}$ and $\underline{\text{candidateA.getVotes}}$ are operation names that are bound by the outer `withCandidateStartingAt` stack frame, and $\underline{\text{candidateB.countVote}}$ and $\underline{\text{candidateB.getVotes}}$ are operation names that are bound by the inner `withCandidateStartingAt` stack frame.

We feel it is crucial to this thesis that the four (underlined) operation names are bindings created by the `withCandidateStartingAt` stack frames — or syntactically, by the `withCandidateStartingAt` binder. The stack frames no longer exist below the outer `withCandidateStartingAt` — the bound names are not valid outside their binders. This means that we can no longer refer to the four operation names bound by these two stack frames / binders. The type system only lets us call these functions while the respective stack frames are active. Indeed, we cannot express these functions' types outside of the stack frames, so we can certainly not type-check applications of these functions there.

Why this limitation mitigates the problem with aliasing

More than merely a restriction, this prevents us from doing the wrong things. Recall the problem with mutable references we described on page 4, where we wanted to take a snapshot of the election state, but we only copied "pointers" to the mutable references and not the value inside.

[2] These are the *generic effects* that correspond to the algebraic operations that we use in this thesis, although we only consider finitary algebraic operations in this thesis. Algebraic operations and generic effects are equivalent; see Plotkin and Power [60].

In this situation, we can still take an "aliased snapshot" tuple `(candidateA, candidateB)` with references to the two stack frames; we can even pass this tuple to functions. *But we are prevented from returning this aliased snapshot tuple out of* `main`, or passing it to a method on a stack frame outside of `main`, because references to the stack frame are untypeable outside of it.

We are likely to run into these limitations for doing anything real with aliased references, and this will probably remind us to instead take snapshots of the *values inside* the stack frames.

Why non-lexical approaches do not avoid the problems with aliasing

In enforcing this restriction, it seems relevant that this "availability of operations" is expressed with lexical binding. There are two related schemes that do not use lexical binding, with significant limitations for our use case:

- *Global operation names*, like in Plotkin and Pretnar [62]. In this case, two nested calls to `withCandidateStartingAt` will handle the same operations; the inner call to `withCandidateStartingAt` will shadow the outer one.

- *Dynamic generation of operation names/instances*, as suggested by Bauer and Pretnar in their work on in Eff [8]. In this scheme, a new operation/name would be generated for both invocations of `withCandidateStartingAt` at runtime. The operational semantics works for our purpose, but we are unaware of suitable type systems that may be used to check that generated operations are only used within a certain scope.

There is a "halfway option" as well: *global operation names parametrised by global instances*, as suggested in Core Eff of Bauer and Pretnar [7]. In this scheme, there would be a static set of 2 "instances" for Candidate: `candidateA` and `candidateB`. If we implement this vote-counting program in this scheme, we will avoid shadowing, and the type system will be able to check that "aliased snapshots" are not used outside of the `withCandidateStartingAt` stack frames. But shadowing/aliasing would still occur in more complicated situations with

recursion, it would not resolve the semantics problem with nonlexical binding that we will describe in the next section.

Remark. In this section, we use syntax inspired by method calls in object-oriented programming. The actual language described in this thesis defines lexically-bound *operations* (not operations on a lexically-bound name), but we feel that for the purpose of our analysis, the difference is insubstantial.

Remark. The type system in the formal language of this thesis (§6) does not yet support polymorphism over operation names. So we can write a *typeable term* for `withCandidateStartingAt` that can be inserted textually twice and type-checked at slightly different types, to make the example type-check, but we cannot see `withCandidateStartingAt` as a term with a *fixed type* that we can insert twice. Zhang and Myers [71] have created a type system with polymorphism for a similar language; we believe their results carry over to a type system and operational semantics for our language straightforwardly. We investigate the relation with Zhang and Myers's work more on page 46.

1.3 Exceptions and lexical binding

DAO bears resemblance to exception handling — also a mechanism for nonlocal control flow — but exception handling is dynamically scoped instead. Let us look at an example of exception handling in OCaml to understand the difference. Assume a previously defined function `p` from integers to booleans. The following function `f` takes an argument `list`, a list of integers, and finds whether predicate `p` applied to some element of the list is true.

```
open List;;
exception TrueOnSomeInput;;
(*  Function f is of type:  int list -> bool  *)
let f(list) = (
    try (
        list |> map (function x ->
            if p(x) then raise TrueOnSomeInput
        );
        false
    ) with
        TrueOnSomeInput -> true
);;
```

It can be disputed whether this code is written in optimal style, but a lot of code in industry uses patterns like this, and this function is correct.

But the function does not generalize very well. Consider the following function g, a variant of `f` which abstracts from `p`:

```
open List;;
exception TrueOnSomeInput;;
(*  Function g is of type: (int -> bool) * int list -> bool  *)
let g(q, list) = (
    try (
        list |> map (function x ->
            if q(x) then raise TrueOnSomeInput
        );
        false
    ) with
        TrueOnSomeInput -> true
);;
```

Function g has some surprising behaviour. It returns `true` when `q` returns true on some element of `list`, but it also returns `true` when `q` raises `TrueOnSomeInput` on some element of `list`. This was not intended. It makes the external behaviour of `g` very complex to describe: every precise explanation will have to mention the `TrueOnSomeInput` exception, which was intended to be an implementation detail.

Early (lexical) and late (dynamic) binding

The reason behind this mess is the binding mechanism in exceptions. Exceptions use a resolution mechanism which is sometimes called **dynamic** or **late binding**, which binds identifiers across functions. There is a single global name, `TrueOnSomeInput`, which in this case is relevant to its external behaviour. Within the temporal scope of `try ( .. ) with`, this name has an extra binding given by the program fragment `TrueOnSomeInput -> true`. This binding is "visible" from function `q`, whatever it may be. Conversely, if a function would be created within the `try ( .. ) with` but returned out of it, then this function *would not* see the binding for `TrueOnSomeInput`.

Operationally, "dynamic binding" can be implemented by maintaining a stack of current bindings for each dynamic variable.

The opposite of late binding is **early** or **lexical binding**. This is what normal variables use: they are available within the *textual* scope of that binding. In our example, `q` and `list` are (lexical) variables. Variable `q` is available inside the function that we create, and would be even if that function would hypothetically be returned from `g`.[3] Conversely, variables `q`, `list`, and `x` *are not* available to the function *denoted by* `q` — although of course it takes the *value denoted by* `x` as an argument.

Operationally, lexical binding can be implemented by β-expansion (capture-avoiding substitution into the program text), or by pairing the program text for functions with a list of variable bindings, thus creating a *function closure*.

[3] One could say that the binding is "done" before the program even starts to execute: "early" indeed.

Remark. Early versions of LISP used dynamic binding for program variables [67], and this is still the default in Emacs Lisp [27]. Later Lisps switched to lexical binding by default, such as Scheme [69, 66], where lexical binding is considered a main distinguishing feature of the language.[4]

Nullary DAO: "exceptions" with lexical binding

This thesis is about *defined algebraic operations* or *DAO*, which is a generalisation of a lexical form of exception handling. Here is the analogous function g using DAO. Syntactically it looks completely the same as g, except that `try ( .. ) with` has been replaced with `try ( .. ) wherealg`. In the rest of this thesis, the notation used is $-$ wherealg $-$.

```
open List;;
(* no declaration here for TrueOnSomeInput *)
let g_dao(q, list) = (
    try (
        list |> map (function x ->
            if q(x) then raise TrueOnSomeInput
        );
        false
    ) wherealg
        TrueOnSomeInput -> true
);;
```

This code looks and behaves very similarly in practice, but instead of an exception it raises a *defined operation*, `TrueOnSomeInput`, which is defined in the penultimate line for the extent of the `try ( .. )` block. Below in Section 1.4, we will look more at other types of algebraic operations; here we are using a *nullary* defined algebraic operation, which is the same as an ordinary exception without data, and with a different binding mechanism.

[4] Steele Jr. and Sussman [69] write *lexical scoping* for *lexical binding*. Sperber et al. [66] write *static scoping*.

The meaning of `g_dao` is a lot simpler than that of `g`. Given a function `q`, the meaning of `g_dao(q, [1; 2; 3])`[5] is precisely the same as:

```
(* Meaning of g_dao(q, [1; 2; 3]): *)
if q(1) then true
  else if q(2) then true
    else if q(3) then true
      else false
```

That is: evaluate `q(1)`, `q(2)`, and `q(3)`, prematurely returning `true` as soon as any of the three calls to `q` return `true`, and returning `false` otherwise. The expression above is very much equivalent to `g_dao(q, [1; 2; 3])`, including in the cases where `q(1)`, `q(2)`, `q(3)` could themselves raise defined operations.

Alain Frisch previously proposed a similar construct, a lexical form of exceptions for OCaml under the name *static exceptions* [28].

Equivalences, like in lambda calculus

We do not give a detailed proof here that `g_dao(q, [1; 2; 3])` indeed simplifies to the nice nested `if` expression. But we will point out how the relevant proof steps follow from the base logic in Figure 6.4 on page 133, when applied for the OCaml-like language that we employ for this introduction. We expand the definition of `g_dao(q, [1; 2; 3])`,

```
(* g_dao(q, [1; 2; 3]) is equivalent to: *)
try (
    [1; 2; 3] |> map (function x ->
        if q(x) then raise TrueOnSomeInput
    );
    false
```

(continues on next page)

[5] OCaml writes `[1; 2; 3]` for the three-element list containing 1, 2, 3.

(continued from previous page)

```
) wherealg
    TrueOnSomeInput -> true
```

and assume that we can expand as follows:

```
(* g_dao(q, [1; 2; 3]) is equivalent to: *)
try (
    let x1 = q(1) in
    if x1 then raise TrueOnSomeInput else
        let x2 = q(2) in
        if x2 then raise TrueOnSomeInput else
            let x3 = q(3) in
            if x3 then raise TrueOnSomeInput else false
) wherealg
    TrueOnSomeInput -> true
```

The logic says that **operation definition commutes with sequencing on the right** when that typechecks. Translated to this OCaml-like syntax: when `M` does not mention the operations defined in `[..definitions..]`, then:

```
    try (   let x = M in N   ) wherealg [..definitions..]
=   let x = M in    try ( N )  wherealg [..definitions..]
```

In our example, `q` was an argument to `g_dao`, so by construction it cannot mention those operations and the rule applies.

To complete the proof, we use this rule and additionally the following rules:

- `if x then..else..` commutes with operation definition,
- evaluation of `try (raise TrueOnSomeInput) wherealg TrueOnSomeInput -> true` to `true`,
- evaluation of `try (false) wherealg TrueOnSomeInput -> true` to `false`.

Consequence: DAO has a simpler syntax than handling

In the formal syntax, this rule looks as follows:

$$(M \text{ to } x.\, N) \text{ wherealg } \mathcal{A} \quad \equiv \quad M \text{ to } x.\, (N \text{ wherealg } \mathcal{A}) \tag{1.1}$$

As a consequence of the soundness of the logic (Theorem 85), and a clever choice of syntax, in practice **we may omit the parentheses** and simply write:

$$\boxed{M \quad \text{to } x. \quad N \quad \text{wherealg } \mathcal{A}}$$

Compare furthermore Benton and Kennedy's ternary construct [9] for exceptions handling, which is sometimes spelled:

$$M \left\{ \begin{array}{ll} \text{to } x. & N \\ \text{catch } E_1. & P_1 \\ \quad \vdots & \\ \text{catch } E_n. & P_n \end{array} \right.$$

(Its meaning is as follows. If M returns a value V, then $N[V/x]$ should be evaluated without exceptions $E_1, \ldots, E_n$ being caught by the listed handlers.)

As we use operation definition instead of handling, if we want to refer to operations defined elsewhere, we can simply rename the newly defined operations to avoid conflicts. We can thus employ a binary operation definition construct (without "to-clause") without a loss in expressivity.

Remark. Equation (1.1) has a curious-looking consequence: $M \equiv (M \text{ wherealg } \mathcal{A})$, when it type-checks, because when it type-checks, then M does not refer to any operations defined in $\mathcal{A}$. This is Lemma 82, and the proof is rather trivial.

1.4 Non-nullary algebraic operations

Exceptions and errors (in the sense of aborting a program) are only one case of algebraic operations. In this thesis, we consider ***n*-ary algebraic operations**. Parameter n of the operation, which is a natural number, corresponds to **the number of ways in which execution may resume after the operation is called**. For instance, we might have a binary algebraic operation `askCoffeeOrTea` which asks the user if they prefer coffee or tea. Keeping in line with ML syntax, we might write the following function to get the user's preference as a boolean:

```
let prefersTea() = (
    raise askCoffeeOrTea
      | Coffee -> false
      | Tea    -> true
);;
```

The syntactic similarity with ML pattern matching in lines 3 and 4 is on purpose: `raise askCoffeeOrTea` can continue in two ways, namely *Coffee* or *Tea*.

For another example, we might have a binary operations `flipCoin`, which flips a coin and which can continue with either `Heads` or `Tails`:

```
let numberOfHeads() = (
    raise flipCoin
      | Heads -> 1
      | Tails -> 0
);;
```

Indeed, we can bind the result to a variable, and do it again:

```
let numberOfHeads2() = (
    let heads1 = raise flipCoin
      | Heads -> 1
      | Tails -> 0
    let heads2 = raise flipCoin
      | Heads -> 1
      | Tails -> 0
    heads1 + heads2
);;
```

We can now fold the last line into the second `raise`:

```
let numberOfHeads2() = (
    let heads1 = raise flipCoin
      | Heads -> 1
      | Tails -> 0
    raise flipCoin
      | Heads -> heads1 + 1
      | Tails -> heads1 + 0
);;
```

It follows from a law of our language that the last two programs are equivalent, because we only consider operations that are "algebraic", which means the following for a binary operation `Op`. Let `t`, `u`, `v` stand for three terms. The following program raises `Op` and binds the result of either `t` or `u` to `x`, after which `x` may be used in `v`:

```
let x = raise Op
    | Left  -> t
    | Right -> u
v
```

`Op` being **algebraic** means that the following term is equal:

```
raise Op
  | Left  -> (let x = t; v)
  | Right -> (let x = t; v)
```

Intuitively, an operation is algebraic if it happens but there is no observable end: there is no code scope associated with the result of a dice roll or coffee/tea preference. In the last example, nothing special happens after `t` or `u`, the program just continues executing, and this is expressed by the last two programs being equivalent.

The common formal definition says that algebraic operations commute with sequencing computations on the right. In a monadic semantics, an n-ary algebraic operations is a family of morphisms $\alpha_x : (T\,x)^n \to (T\,x)$ that is a natural transformation in the Kleisli category; see Plotkin and Power [58, 59].[6]

Because algebraic operations "do not have an end", some people use an equivalent presentation called *generic effects* [60]. Here, we model the same concept with merely a computation returning which branch was chosen — for instance, `flipCoin` would be a function returning a boolean. Let us translate the `numberOfHeads` example from page 17:

```
let numberOfHeadsGenEff() = (
    match flipCoin() with
      | false -> 0
      | true  -> 1
);;
```

In this thesis we choose to use the algebraic operations presentation, because it facilitates our formal development. Practical implementations such as Koka [47] and Eff [8] tend to use the generic effects presentations, but it is trivial to implement the generic effect using an algebraic operation, and conversely a program using algebraic operations is trivially translated to use generic effects.

[6] In some of the literature — including [58] — algebraic operations are called *algebraic effects*.

1.5 Examples of defined algebraic operations

We will proceed to give some examples of defined algebraic operations, and how they can be used. DAO consists of two parts: operations and operation definitions. *Operations* are basically a form of custom syntax using uninterpreted symbols. *Operation definitions* give meaning to the operations within the scope of that definition. It is possible to define

1. operations that break out of a block of code;
2. operations that multiply a new value into a monoid;
3. operations that read from or write to state;
4. operations with multiple branches — for instance to model nondeterminism — which evaluate none, some, or all of their branches all the way to the end of the scope, and combine the end results in some way;
5. operations that save a checkpoint, resume from a checkpoint, or throw away unneeded checkpoints.

Remark. These examples are also possible with *algebraic effect handling*; see Bauer and Pretnar [8] for worked-out examples.

Backtracking example

The formal language we will use in this thesis is based on fine-grain call-by-value or *FGCBV* [48]. We go through the language in detail below in §1.6. But let us first give a concrete example of defined algebraic operations in this language, for use case #4 above. For the example, we furthermore assume a natural number type and function terms $minus$ and max.

We will use a binary $\underline{\text{either}}$ operation to naturally express a nondeterministic computation, and use an operation definition to evaluate all the branches. The program will nondeterministically pick x to be either 4 or 8, and then pick y to be either 2 or 3, then compute $x - y$. The code

for the inner program is as follows — we write $M \text{ to } x.\, N$ for sequencing in FGCBV.

$$\underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x.$$
$$\underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.$$
$$\mathit{minus}\ \langle x, y\rangle$$

We use the new $-\ \text{wherealg}\ -$ language construct to give an operation definition for $\underline{\text{either}}$ which finds the maximum possible end result. It works as follows. When $\underline{\text{either}}(M_1, M_2)$ is invoked:

1. We first evaluate the continuation of the left branch, as if $\underline{\text{either}}(M_1, M_2)$ was M_1 instead. We evaluate all the way to completion, until we have the maximum of the left branch: the maximum of $4-2$ and $4-3$ is 2.

2. Then, we evaluate the continuation of the right branch to completion, as if there was M_2 instead — this gives $\max(8-2, 8-3) = 6$.

3. We return the maximum of the results of the two branches: $\max(2, 6) = 6$.

If the inner program returns a value without invoking $\underline{\text{either}}$, then that value is simply returned and wherealg has no effect.

Here is the complete program. The definition of $\underline{\text{either}}$ is the part after $=$ in the bottom line.

$$\{$$
$$\quad \underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x.$$
$$\quad \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.$$
$$\quad \mathit{minus}\ \langle x, y\rangle$$
$$\} \quad \text{wherealg } \Big(\underline{\text{either}}\ k_1\ k_2\ =\ k_1\ \langle\rangle \text{ to } v_1.\ k_2\ \langle\rangle \text{ to } v_2.\ \mathit{max}\ \langle v_1, v_2\rangle\Big)$$

Note that even though $\underline{\text{either}}$ looks deceptively much like a function, it is really not: if we called a function twice to find the maximum value for x and y, then the result would have been $\max(4, 8) - \max(2, 3) = 5$ not 6.

Of course, we could do a nondeterministic computation like this easily in any programming language, by generating all possible values for $x - y$ using nested loops or function calls, then picking the maximum of all generated values. But the nice thing about the *operations* approach is that it reads much more naturally. This approach is also very flexible: we could decide at runtime whether or not to make a nondeterministic choice at all, just by putting $\underline{\text{either}}$ in an if-statement.

Remark. Even though the program fragment uses effects (namely the $\underline{\text{either}}$ operation), the program as a whole is a pure computation, and it will be typed as such. Indeed, its typing judgement and semantics do not mention $\underline{\text{either}}$, which was defined inside the program and has no bearing at all on the interaction between the program and the environment.

Other operation definitions for $\underline{\text{either}}$

We give two more examples of operation definitions related to nondeterminism. First, a degenerate operation definition that always picks the left branch. We repeat the whole program even though only the last line changes:

$$
\begin{array}{l}
\{ \\
\quad \underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x. \\
\quad \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y. \\
\quad \mathit{minus}\ \langle x, y \rangle \\
\} \quad \text{wherealg } \Big(\underline{\text{either}}\ k_1\ k_2\ =\ k_1\ \langle\rangle \Big)
\end{array}
$$

And the following program finds the average of all generated values. It does so by computing the number of leaf results and their sum, then dividing the sum by the number. We assume

suitable functions add and $divide$. The latter might round towards zero, for instance.

$$
\begin{array}{l}
\{ \\
\quad \underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x. \\
\quad \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y. \\
\quad minus\ \langle x, y\rangle \text{ to } leaf. \\
\quad \text{return } \langle 1, leaf\rangle. \qquad // \ \textsf{number} = 1, \textsf{sum} = leaf \\
\} \quad \text{wherealg } \Big(\underline{\text{either}}\ k_1\ k_2 \ = k_1\ \langle\rangle \text{ to } pair_1.\ \text{case } pair_1 \text{ of } \langle number_1, sum_1\rangle. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad k_2\ \langle\rangle \text{ to } pair_2.\ \text{case } pair_2 \text{ of } \langle number_2, sum_2\rangle. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad add\langle number_1, number_2\rangle \text{ to } number. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad add\langle sum_1, sum_2\rangle \text{ to } sum. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{return } \langle number, sum\rangle\Big) \\
\text{to } pair.\ \text{case } pair \text{ of } \langle number, sum\rangle. \\
\text{return } divide\langle sum, number\rangle.
\end{array}
$$

We give further examples in Chapter 8.

1.6 Formal language: FGCBV with defined algebraic operations

1.6.1 Fine-grain call-by-value plus defined algebraic operations

We present this work based around a programming language for DAO, in the style of *fine-grain call-by-value* lambda calculus or *FGCBV* [48]. We give the complete grammar in Figure 1.3 on the next page. In this section, we go slowly through all the language constructs.

We separate (passive) *values* and (active) *computations*. **Values** are the things represented by the variables in our context, and are manipulated without causing side-effects. We typically

$$
\begin{array}{lrcl}
\text{values} & V, W & ::= & x \mid \lambda x.\, M \mid \langle\rangle \mid \text{inl } V \mid \text{inr } V \mid \langle V, W\rangle \\
\text{computations} & M, N, P & ::= & \text{return } V \mid \alpha\, \vec{M} \mid M \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_\alpha \\
& & & \mid \text{let } V \text{ be } x.\, M \mid M \text{ to } x.\, N \mid V\, W \\
& & & \mid \text{case } V \text{ of } \{\} \mid \text{case } V \text{ of } \{\langle\rangle.\, M\} \\
& & & \mid \text{case } V \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} \mid \text{case } V \text{ of } \{\langle x, y\rangle.\, M\}
\end{array}
$$

Figure 1.3: The grammar of our defined algebraic operation language. There is a new kind of identifier, called an *operation*. We write abstract operations as $\alpha, \beta, \cdots$. We underline specific ones, e.g: $\underline{\text{either}}$. Construct let is syntactic sugar for other combinations, for instance $\text{return } V \text{ to } x.\, M$.

use V, W to range over values. Values may be the unit value $\langle\rangle$, a pair of values $\langle V, W\rangle$, or a left or right injected value of a sum type. A value can also be a variable from the context.

Computations are the kinds of things that are executed. We typically use M, N, P to range over computations. The simplest kind of computation $\text{return } V$ just **returns** a value V, which we call its *result*.

Another kind of computation performs **input/output** with the environment; this environment might for instance be the user that operates the program. Our language uses unified input/output: there is a set of possible output types ("operations") L that the program must choose from when it chooses to interact. We use α, β, γ for abstract operation names, and write concrete operation names in underlined $\underline{\text{roman}}$ font. Each operation is assigned an *arity* by the *arity assignment* $\Delta : L \to \mathbb{N}$, which is the number of possible responses that the environment might give back. The program can continue in different ways depending on what response was given by the environment, so if we have an operation β with binary arity $\Delta(\beta) = 2$, then in our language $\beta(\text{return } 3, \text{return } 4)$ represents the program that outputs "β", and then returns 3 or 4 depending on whether the environment picked the left or the right response. There may be also be nullary operations — invoked as $\gamma()$ when $\Delta(\gamma) = 0$ — which terminate the program without yielding a return value.

Thus, input and output are unified in our programming language: input is only ever given in response to output; a unary operation can be used to model output without input, but asking for input is always a proper action.

Another kind of computation is a **sequencing** $M \text{ to } x.\, N$, which first runs computation M, and then runs N with x bound to the result of M, if and when M has a result. (The "$x.$" notation was chosen to suggest that this construct binds x in what follows.)

$$M, N, P \ ::= \ \cdots \mid M \text{ to } x.\, N \mid \cdots$$

Indeed, if the left part immediately returns a value then there is no proper sequencing:

$$\text{return } V \text{ to } x.\, M \ \equiv \ M[V/x]$$

And if the left part first executes I/O, then that is what the sequenced computation does, too:

$$\alpha(\vec{M}) \text{ to } x.\, N \ \equiv \ \alpha(\overrightarrow{M \text{ to } x.\, N}) \tag{1.2}$$

This notation is shorthand for the following: suppose that we have an $\Delta(\alpha)$-ary collection of computations $(M_i)_{i \in \{1,..,\Delta(\alpha)\}}$ and a computation N, then

$$\Big(\alpha\ (M_i)_{i \in \{1,..,\Delta(\alpha)\}}\Big) \text{ to } x.\, N \ \equiv \ \alpha\ \Big(M_i \text{ to } x.\, N\Big)_{i \in \{1,..,\Delta(\alpha)\}} \quad . \tag{1.2}$$

Essentially, this means that operations happen "at an instant", and they do not have any relevant scoping. On the left side, $\alpha(\vec{M})$ is of the same type as each M_i. On the right side, $\alpha(\overrightarrow{M \text{ to } x.\, N})$ is of the same type as N. But we can apply α at any computation type (as long as the computation type allows α at all). The equation expresses that it does not matter at all what type we invoke α at; the only thing that matters is the branch number that it essentially returns.

Some people say that operation α is *algebraic* to mean (1.2). In our system of defined algebraic operations, all operations are algebraic. Recall our explanation of algebraicity on page 18.

Unsurprisingly, the sequencing of computations is associative.

$$(M \text{ to } x.\, N) \text{ to } y.\, P \ \equiv \ M \text{ to } x.\, (N \text{ to } y.\, P)$$

We define two **convenience constructs**. One sequences two computations, when the result of the first computation is irrelevant in the second:

$$M; N \quad \stackrel{\text{def}}{=} \quad M \text{ to } x.\, N$$

The semicolon is not part of the programming language proper, but simply an abbreviation. The second construct binds a value to a variable:

$$\text{let } V \text{ be } x.\, M \quad \stackrel{\text{def}}{=} \quad \text{return } V \text{ to } x.\, M$$

In other languages this might be written $\text{let } x = V \text{ in } M$, but we prefer to keep the syntax more uniform. We choose to make let part of the language proper, even though it is redundant, because it is easier to give semantics to than to sequencing, and thus it assists in our presentation.

Remark. Many programming languages write let for sequencing: instead of $M \text{ to } x.\, N$ they might write $\text{let } M \text{ be } x.\, N$. Some other languages, including Haskell's `do` notation, denote sequencing using something like $x \leftarrow M;\ N$ or $x = M;\ N$. In this thesis, we exclusively use the word let for binding a *value* to a variable, rather than doing something potentially effectful before continuing in N. We remind the reader that this thesis uses a fine-grain *call-by-value*, and so variables never refer to computations, and mentioning a variable will never cause an effect (in contrast to *call-by-name* languages).

We have presented a fair amount of our language now. The syntax that remains is

- lambda abstractions, presented immediately below,
- pattern matching, presented in §1.6.2,
- defined operations, presented in §1.6.4 and §1.6.5.

and we give an operational semantics throughout this chapter, starting in §1.6.3.

Lambda abstraction

Computations are not necessarily active already. They may be delayed in the form of a **lambda abstraction**, which is a value:

$$V, W \ ::= \ \cdots \mid \lambda x.\, M \mid \cdots$$

Here M is a computation that may use variable x in addition to the existing context. Example:

$$\lambda n.\, \beta(\text{return } n, \text{return } 4)$$

Lambda expressions create *functions*, which are values, and as such they may also be present in the environment and we can refer to them using variables. The delayed computation may be activated by *applying* the function to a value, which provides a value for the new variable. **Function application** is written using juxtaposition: to apply a function variable f to the value 3, we write:

$$f\, 3$$

If we see both the function body and its argument, then we can evaluate by substituting:

$$(\lambda x.\, M)\, V \ \equiv \ M[V/x]$$

So $(\lambda n.\, \beta(\text{return } n, \text{return } 4))\ 3$ evaluates to $\beta(\text{return } 3, \text{return } 4)$. We call computations of the form $\text{return } V$ or $\alpha(\vec{M})$ terminals: they can no longer be simplified on the outside.

1.6.2 Pattern matching

An important remaining construct is pattern matching, to deconstruct values that we have. We have four pattern match constructs (for nullary and binary sums and products); all pattern match constructs look alike. A pair value can be deconstructed by writing

$$\text{case } V \text{ of } \{\langle x, y\rangle.\, M\}$$

where M additionally has variables x and y in context. When we see the two parts of V, we can evaluate using the "beta law for products":

$$\text{case } \langle W, W' \rangle \text{ of } \{\langle x, y \rangle .\, M\} \;\equiv\; M[W/x, W'/y]$$

For sum types we write $\text{case } V \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\}$, which we evaluate similarly:

$$\text{case inl } W \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} \;\equiv\; M[W/x]$$
$$\text{case inr } W' \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} \;\equiv\; N[W'/y]$$

The unit type 1 is similar: $\text{case } V \text{ of } \{\langle\rangle .\, M\}$ and

$$\text{case } \langle\rangle \text{ of } \{\langle\rangle .\, M\} \;\equiv\; M$$

That leaves the zero type 0, which does not have any values. But it may be presumed in a context. In such impossible contexts there are value expressions of the uninhabited type, and so there are zero possible cases to consider:

$$\text{case } V \text{ of } \{\}$$

There is no beta law for this pattern match construct.

1.6.3 Operational semantics of FGCBV with algebraic operations

The operational semantics of our language applies the beta rules on the outside, until we reach a terminal,

$$\text{terminals} \qquad T \;::=\; \text{return } V \mid \alpha(\vec{M})$$

That is, we do not evaluate under operations, but we do evaluate on the left side of a sequencing. Recall that we have these beta rules:

$$\begin{aligned}
\text{return } V \text{ to } x.\, M &\equiv M[V/x] \\
\alpha(\vec{M}) \text{ to } x.\, N &\equiv \alpha(\overrightarrow{M \text{ to } x.\, N}) \\
(\lambda x.\, M)\, V &\equiv M[V/x] \\
\text{case } \langle\rangle \text{ of } \{\langle\rangle.\, M\} &\equiv M \\
\text{case inl } V \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} &\equiv M[V/x] \\
\text{case inr } W \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} &\equiv N[W/y] \\
\text{case } \langle V, W\rangle \text{ of } \{\langle V, W\rangle.\, M\} &\equiv M[V/x, W/y]
\end{aligned}$$

To make precise what we mean by applying the beta rules "on the outside", we introduce evaluation contexts, which are terms with a single hole. For our current purposes, the hole in an evaluation context is found by navigating into the left side of sequencings. (Later we will introduce the $M \text{ wherealg } \mathcal{A}$ construct, which evaluates inside the left side as well.)

$$\text{evaluation contexts} \quad \mathcal{E}, \mathcal{F} ::= [\,] \mid \mathcal{E}[\,] \text{ to } x.\, N$$

Then our operational semantics in §6.7 rewrites along those beta equalities within evaluation contexts:

$$\begin{aligned}
\mathcal{E}\,[\text{return } V \text{ to } x.\, M] &\to \mathcal{E}\,[M[V/x]] \\
\mathcal{E}\left[\alpha(\vec{M}) \text{ to } x.\, N\right] &\to \mathcal{E}\left[\alpha(\overrightarrow{M \text{ to } x.\, N})\right] \\
\mathcal{E}\,[(\lambda x.\, M)\, V] &\to \mathcal{E}\,[M[V/x]] \\
\mathcal{E}\,[\text{case } \langle\rangle \text{ of } \{\langle\rangle.\, M\}] &\to \mathcal{E}\,[M] \\
\mathcal{E}\,[\text{case inl } V \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\}] &\to \mathcal{E}\,[M[V/x]] \\
\mathcal{E}\,[\text{case inr } W \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\}] &\to \mathcal{E}\,[N[W/y]] \\
\mathcal{E}\,[\text{case } \langle V, W\rangle \text{ of } \{\langle V, W\rangle.\, M\}] &\to \mathcal{E}\,[M[V/x, W/y]]
\end{aligned}$$

In Section 6.7 we present the operational semantics in a more systematic way, namely in medium-step style, but we feel that an explanation using contexts is clearer at this point.

1.6.4 Defined algebraic operations (DAO) with one unary operation

In a system with input/output, the algebraic operations might show a message on the user's screen, in response to which the user would choose one of the available inputs. With defined algebraic operations (DAO), there is an alternative: the defined behaviour of an algebraic operation has access to part of the computation, reified as a function. On an actual computer, this part of the computation might correspond to an upper segment of the stack, which we take off and reify as a function. In its place, we put a stack segment corresponding to the operation definition — bound to the new function.

To give an example, let us consider an abstract unary operation $\underline{\text{squeak}}$ without any inherent meaning. Suppose that we have a term $P_{\underline{\text{squeak}}}[k]$ that has access to the top portion of the stack in the form of a delayed computation k that takes $\langle\rangle$ as its argument: to evaluate the top of the stack, we write $k\,\langle\rangle$. We give the operation definition as $\mathcal{A} = (\underline{\text{squeak}}\ k = P_{\underline{\text{squeak}}})$. We can start writing terms with the $\underline{\text{squeak}}$ operation; to indicate where the stack should be split, we use wherealg. For example:

$$\Big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\, \underline{\text{squeak}}(\text{return } x)\Big) \text{ wherealg } \mathcal{A} \text{ to } y.\, N \tag{1.3}$$

This is the small-step evaluation rule for our operation definition $\mathcal{A}$: let both $\mathcal{E}$ and $\mathcal{F}$ stand for evaluation contexts, then

$$\mathcal{E}\Big[\mathcal{F}\big[\underline{\text{squeak}}(M)\big] \text{ wherealg } \mathcal{A}\Big] \;\to\; \mathcal{E}\Big[P_{\underline{\text{squeak}}}\Big[\big(\lambda\langle\rangle.\, \mathcal{F}[M] \text{ wherealg } \mathcal{A}\big) \,/\, k\Big]\Big] \quad . \tag{1.4}$$

In (1.3), $\mathcal{F}$ would match the stack segment $(-\ \text{to}\ x.\, \underline{\text{squeak}}(\text{return}\ x))$. Finally, we are allowed to evaluate under $-\ \text{wherealg}\ \mathcal{A}$:

$$\text{evaluation contexts} \quad \mathcal{E}, \mathcal{F} \ ::= \quad \cdots \ \mid\ \mathcal{E}[\,] \text{ wherealg } \mathcal{A}$$

The evaluation context *binds* the operations defined in $\mathcal{A}$; this is the big difference with *handlers* of algebraic operations.

Example: evaluate the continuation once

For an example, suppose that $P_{\underline{\text{squeak}}}$ is simply $k\,\langle\rangle$ — it just resumes the continuation — so that the rule transforms

$$\begin{aligned}\mathcal{E}\Big[\mathcal{F}\big[\underline{\text{squeak}}(M)\big]\ \text{wherealg}\ \mathcal{A}\Big] &\to \mathcal{E}\Big[\big(\lambda\langle\rangle.\,\mathcal{F}[M]\ \text{wherealg}\ \mathcal{A}\big)\ \langle\rangle\Big] \\ &\to \mathcal{E}\Big[\mathcal{F}[M]\ \text{wherealg}\ \mathcal{A}\Big] \quad .\end{aligned}$$

(Exercise for the reader: verify this!)

Then we evaluate in the example term (1.3) as follows. When applying rules, we indicate with a long underline the part that matches the inside of $\mathcal{E}$ in the matched rule:

$$\begin{array}{l}
\underline{\big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)\big)\ \text{wherealg}\ \mathcal{A}\ \text{to}\ y.\,N} \qquad \text{(defined operation)} \\
\underline{\big(\lambda\langle\rangle.\,\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)\ \text{wherealg}\ \mathcal{A}\big)\ \langle\rangle\ \text{to}\ y.\,N} \qquad \text{(function application)} \\
\big(\underline{\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)}\big)\ \text{wherealg}\ \mathcal{A}\ \text{to}\ y.\,N \qquad \text{(return/sequencing)} \\
\underline{\underline{\text{squeak}}(\text{return } 3)\ \text{wherealg}\ \mathcal{A}\ \text{to}\ y.\,N} \qquad \text{(defined operation)} \\
\underline{\big(\lambda\langle\rangle.\,\text{return } 3\ \text{wherealg}\ \mathcal{A}\big)\ \langle\rangle\ \text{to}\ y.\,N} \qquad \text{(function application)} \\
\text{return } 3\ \text{wherealg}\ \mathcal{A}\ \text{to}\ y.\,N
\end{array}$$

To evaluate further, we need the rule that says that if an operation is defined but not used, then it may as well not be defined:

$$\mathcal{E}\,[\text{return } V\ \text{wherealg}\ \mathcal{A}] \to \mathcal{E}\,[\text{return } V] \tag{1.5}$$

We evaluate further:

$$\begin{array}{l}
\underline{\text{return } 3\ \text{wherealg}\ \mathcal{A}\ \text{to}\ y.\,N} \qquad \text{(return/wherealg)} \\
\underline{\text{return } 3 \text{ to } y.\,N} \qquad \text{(return/sequencing)} \\
N[3/y]
\end{array}$$

Example: aborting the computation

If it is not yet obvious: the particular $P_{\underline{\text{squeak}}}$ (the definition of the operation) matters very much. If we had had $P_{\underline{\text{squeak}}} = \text{return } 5$ then the continuation would not have been evaluated

— invoking $\underline{\text{squeak}}(-)$ would effectively abort the current computation:

$$\dfrac{\dfrac{\big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\, \underline{\text{squeak}}(\text{return } x)\big) \text{ wherealg } \mathcal{A} \text{ to } y.\, N}{\underline{\text{return } 5 \text{ to } y.\, N}}\ \text{(return/sequencing)}}{N[5/y]}\ \text{(defined operation)}$$

Observe that the wherealg and the "$\text{to } x.\, \underline{\text{squeak}}(\text{return } x)$" are immediately gone after the first step, because according to Eq. (1.4) the wherealg and the continuation are only recreated in the substitution for k, and k does not appear in $P_{\underline{\text{squeak}}}$.

Example: evaluating the continuation twice

We now give an example with $P_{\underline{\text{squeak}}} = k\,\langle\rangle;\ k\,\langle\rangle$, that is: 'outputting' $\underline{\text{squeak}}$ will evaluate the continuation, discard it, then evaluate it again. But the continuation still has $\underline{\text{squeak}}$ defined using $P_{\underline{\text{squeak}}}$, so with every time that $\underline{\text{squeak}}$ is 'output', evaluation time approximately doubles, and evaluation can take time exponential in the number of $\underline{\text{squeak}}$ output. This specific example is not necessarily useful by itself as $P_{\underline{\text{squeak}}}$ simply throws away the result of $k\,\langle\rangle$, but there might be use cases for evaluating the continuation multiple times.

We show the (lengthy) calculation of the evaluation in Figure 1.4. It shows that

$$\big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\, \underline{\text{squeak}}(\text{return } x)\big) \text{ wherealg } \big(\underline{\text{squeak}}\ k = k\,\langle\rangle;\ k\,\langle\rangle\big) \text{ to } y.\, N$$

evaluates to $N[3/y]$.

Observe that every occurrence of $\underline{\text{squeak}}$ is carefully guarded by a $\text{wherealg } (\underline{\text{squeak}}\ k = \cdots)$. This property is maintained automatically by construction.

1.6.5 Defined algebraic operations in general

Other arities

There are two more aspects that we should look at. The first aspect is that we may have operations of any finite arity, not just unary operations. For instance, recall that in Section 1.5 we

$\underline{\big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\, \underline{\text{squeak}}(\text{return } x)\big) \text{ whererealg } \mathcal{A}} \text{ to } y.\, N$ (defined operation)

$\underline{(\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle}; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (fun. app.)

$\underline{(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x))} \text{ whererealg } \mathcal{A}; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (ret/seq)

$\underline{\underline{\text{squeak}}(\text{return } 3) \text{ whererealg } \mathcal{A}}; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (defined operation)

$\underline{(\lambda\langle\rangle.\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle}; (\lambda\langle\rangle.\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (fun. app.)

$\underline{\text{return } 3 \text{ whererealg } \mathcal{A}}; (\lambda\langle\rangle.\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (ret/whalg)

$\underline{\text{return } 3}; (\lambda\langle\rangle.\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle; (\lambda\langle\rangle.(\text{return } 3 \text{ to } x.\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.N$ (ret/seq)

$\underline{(\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle};\ (\lambda\langle\rangle.\,(\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (fun. app.)

$\underline{\text{return } 3 \text{ whererealg } \mathcal{A}};\ (\lambda\langle\rangle.\,(\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (return/whererealg)

$\underline{\text{return } 3};\ (\lambda\langle\rangle.\,(\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (return/sequencing)

$\underline{(\lambda\langle\rangle.\,(\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)) \text{ whererealg } \mathcal{A})\,\langle\rangle} \text{ to } y.\,N$ (function application)

$\underline{(\text{return } 3 \text{ to } x.\,\underline{\text{squeak}}(\text{return } x))} \text{ whererealg } \mathcal{A} \text{ to } y.\,N$ (return/sequencing)

$\underline{\underline{\text{squeak}}(\text{return } 3) \text{ whererealg } \mathcal{A}} \text{ to } y.\,N$ (defined operation)

$\underline{(\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle};\ (\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (function application)

$\underline{\text{return } 3 \text{ whererealg } \mathcal{A}};\ (\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (return/whererealg)

$\underline{\text{return } 3};\ (\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle \text{ to } y.\,N$ (return/sequencing)

$\underline{(\lambda\langle\rangle.\,\text{return } 3 \text{ whererealg } \mathcal{A})\,\langle\rangle} \text{ to } y.\,N$ (function application)

$\underline{\text{return } 3 \text{ whererealg } \mathcal{A}} \text{ to } y.\,N$ (return/whererealg)

$\underline{\text{return } 3 \text{ to } y.\,N}$ (return/sequencing)

$N[3/y]$

Figure 1.4: Evaluation of $\big(\underline{\text{squeak}}(\text{return } 3) \text{ to } x.\,\underline{\text{squeak}}(\text{return } x)\big) \text{ whererealg } \mathcal{A} \text{ to } y.\,N$ with $\mathcal{A} \stackrel{\text{def}}{=} \big(\underline{\text{squeak}}\ k = k\,\langle\rangle;\ k\,\langle\rangle\big)$.

had a binary operation $\underline{\text{either}}$, defined:

$$\Big(\underline{\text{either}}\ k_1\ k_2\ =\ k_1\,\langle\rangle \text{ to } v_1.\ k_2\,\langle\rangle \text{ to } v_2.\ max\,\langle v_1, v_2\rangle\Big)$$

Let us abbreviate the right hand side of the equals sign as $P_{\underline{\text{either}}}$. Observe that it is parametrised over two continuations k_1, k_2, as opposed to $P_{\underline{\text{squeak}}}$ which was parametrised over just k. This corresponds to the arity of the defined operation.

The small-step evaluation rule from Equation (1.4) adapts accordingly:

$$\mathcal{E}\Big[\mathcal{F}\big[\underline{\text{either}}(M_1, M_2)\big] \text{ whererealg } \mathcal{A}\Big]\ \rightarrow\ \mathcal{E}\Big[P_{\underline{\text{either}}}\big[\big(\lambda\langle\rangle.\,\mathcal{F}[M_i] \text{ whererealg } \mathcal{A}\big)\,/\,k_i\big]_{i\in\{1,2\}}\Big]$$

To give an example, let us execute the program from §1.5 but with a definition for $\underline{\text{either}}$ that just executes the right hand continuation:

$$\begin{array}{l}\{\\ \quad \underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x.\\ \quad \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.\\ \quad \mathit{minus}\,\langle x, y\rangle\\ \}\ \ \text{wherealg}\ \big(\underline{\text{either}}\ k_1\ k_2\ =\ k_2\,\langle\rangle\big)\end{array}$$

And let us abbreviate $\mathcal{B} = \big(\underline{\text{either}}\ k_1\ k_2\ =\ k_2\,\langle\rangle\big)$ and $P_{\underline{\text{either}}} = k_2\,\langle\rangle$. We evaluate:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\big\{\underline{\text{either}}(\text{return } 4, \text{return } 8) \text{ to } x.\ \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.\ \mathit{minus}\,\langle x, y\rangle\big\} \text{ wherealg } \mathcal{B}}{\big(\lambda\langle\rangle.\ \big\{\text{return } 8 \text{ to } x.\ \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.\ \mathit{minus}\,\langle x, y\rangle\big\} \text{ wherealg } \mathcal{B}\big)\ \langle\rangle}\text{(def. op.)}}{\big\{\underline{\text{return } 8 \text{ to } x.\ \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.\ \mathit{minus}\,\langle x, y\rangle}\big\} \text{ wherealg } \mathcal{B}}\text{(fun. app.)}}{\big\{\underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ to } y.\ \mathit{minus}\,\langle 8, y\rangle\big\} \text{ wherealg } \mathcal{B}}\text{(return/sequencing)}}{\big(\lambda\langle\rangle.\ \big\{\text{return } 3 \text{ to } y.\ \mathit{minus}\,\langle 8, y\rangle\big\} \text{ wherealg } \mathcal{B}\big)\ \langle\rangle}\text{(defined operation)}}{\big\{\underline{\text{return } 3 \text{ to } y.\ \mathit{minus}\,\langle 8, y\rangle}\big\} \text{ wherealg } \mathcal{B}}\text{(function application)}}{\underline{\mathit{minus}\,\langle 8, 3\rangle} \text{ wherealg } \mathcal{B}}\text{(return/sequencing)}}{\underline{\text{return } 5 \text{ wherealg } \mathcal{B}}}\text{(presumed function)}}{\text{return } 5}\text{(return/wherealg)}$$

The evaluation rule for other arities is analogous.

Evaluation contexts bind operations

Secondly, we mentioned before that $\mathcal{E}[\ ]\ \text{wherealg}\ \mathcal{A}$ can be an evaluation context, and it binds the operations in $\mathcal{A}$. We saw before that this was needed to evaluate in the left hand side of a wherealg, for instance in this step above:

$$\big\{\text{return } 3 \text{ to } y.\ \mathit{minus}\,\langle 8, y\rangle\big\} \text{ wherealg } \mathcal{B} \quad\longrightarrow\quad \mathit{minus}\,\langle 8, 3\rangle \text{ wherealg } \mathcal{B}$$

It also causes $\mathcal{E}[\ ]\ \text{wherealg}\ \mathcal{A}$ to be an acceptable evaluation context for the $\mathcal{F}$ part of the small-step evaluation rule. Let us again abbreviate

$$\begin{aligned}\mathcal{A} &= \big(\underline{\text{squeak}}\ k\ =\ k\,\langle\rangle\big)\\ \mathcal{B} &= \big(\underline{\text{either}}\ k_1\ k_2\ =\ k_2\,\langle\rangle\big)\quad ,\end{aligned}$$

then

$$\{\underline{\text{squeak}}(\text{return}\ \langle\rangle); \underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)\}\ \text{wherealg}\ \mathcal{B}\ \text{wherealg}\ \mathcal{A}$$

evaluates as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\underline{\text{squeak}}(\text{return}\ \langle\rangle); \underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)\}\ \text{wherealg}\ \mathcal{B}\ \text{wherealg}\ \mathcal{A}}{\big(\lambda\langle\rangle.\ \{\text{return}\ \langle\rangle; \underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)\}\ \text{wherealg}\ \mathcal{B}\ \text{wherealg}\ \mathcal{A}\big)\ \langle\rangle}\ \text{(defined operation)}}{\{\underline{\text{return}\ \langle\rangle; \underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)}\}\ \text{wherealg}\ \mathcal{B}\ \text{wherealg}\ \mathcal{A}}\ \text{(function application)}}{\{\underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)\}\ \text{wherealg}\ \mathcal{B}\ \text{wherealg}\ \mathcal{A}}\ \text{(return/sequencing)}}{(\lambda\langle\rangle.\ \text{return}\ 4\ \text{wherealg}\ \mathcal{B})\,\langle\rangle\ \text{wherealg}\ \mathcal{A}}\ \text{(defined operation)}}{\underline{\text{return}\ 4\ \text{wherealg}\ \mathcal{B}}\ \text{wherealg}\ \mathcal{A}}\ \text{(function application)}}{\underline{\text{return}\ 4\ \text{wherealg}\ \mathcal{A}}}\ \text{(return/wherealg)}}{\text{return}\ 4}\ \text{(return/wherealg)}$$

Here in the first step, we apply the evaluation rule with

$$\mathcal{F}\ =\ \big(\{-; \underline{\text{either}}(\text{return}\ 3, \text{return}\ 4)\}\ \text{wherealg}\ \mathcal{B}\big)\ ,$$

indeed involving an operation definition. We know for the meaning of $\underline{\text{squeak}}$ to look in $\mathcal{A}$, because $\underline{\text{squeak}}$ refers to the place where it was bound, namely in $\mathcal{A}$.

In contrast, such a blanket small-step evaluation rule does not work for handlers. In the analogous handling computation, $\mathcal{A}$ may handle $\underline{\text{squeak}}$ but in addition, $\mathcal{B}$ *may also* handle $\underline{\text{squeak}}$. In operational semantics for handling, $\mathcal{F}$ is typically restricted to be an evaluation context that does not handle operations; see for instance the *hoisting frames* in Kammar et al. [40].

It seems that an analogous evaluation rule would work for *handled operations*, with the side condition that $\mathcal{F}$ only handles *other* operations. This seems to be the essential behaviour encoded using even smaller-step semantics in [40], §3.3 on forwarding open handlers.

No shadowing for operation names

As M $\mathrm{wherealg}$ $\mathcal{A}$ *binds* operation names within M, we must use capture-avoiding substitution when substituting with terms that bind operation names, as is usual with substituting values for variables.

We go even further, and shall avoid any shadowing of operation names — that is, anytime we write a term that binds an operation, we will use an operation name that does not appear in the context. We do this because the type of a term or variable may mention operation names that are in context, and we wish to avoid any confusion about what operation name a type refers to. We believe that this is not a significant restriction, as any term or derivation represented as a binding diagram can be written as a term with concrete operation names and no shadowing.

This concludes our introduction to the formal language of Chapter 6. For reference, we copy again the grammar of our DAO language, which we give together with a type system in Figure 6.2 on page 117.

computations	M, N, P	$::= \mathrm{return}\ V \mid \alpha\,\vec{M} \mid M\ \mathrm{wherealg}\ \mathcal{A}$
		$\mid \mathrm{let}\ V\ \mathrm{be}\ x.\, M \mid M\ \mathrm{to}\ x.\, N \mid V\,W$
		$\mid \mathrm{case}\ V\ \mathrm{of}\ \{\} \mid \mathrm{case}\ V\ \mathrm{of}\ \{\langle\rangle.\, M\}$
		$\mid \mathrm{case}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\} \mid \mathrm{case}\ V\ \mathrm{of}\ \{\langle x, y\rangle.\, M\}$
values	V, W	$::= x \mid \lambda x.\, M \mid \langle\rangle \mid \mathrm{inl}\ V \mid \mathrm{inr}\ V \mid \langle V, W\rangle$
syntactic algebras	$\mathcal{A}, \mathcal{B}$	$::= (\alpha\,\vec{k} = N_\alpha)_\alpha$

We give a more complete medium-step operational semantics in Section 6.7, and further examples in Chapter 8.

1.7 Reasoning with theories on operations

Programs with *defined operations* tend to satisfy lots of equations because of their behaviour. For instance:

- For operations that are defined to evaluate all branches and combine the results in a commutative way — like our example in § 1.5 on page 20 — the branches can be swapped around if there are no other side-effects. We give the statement and proof throughout §7.6.
- Reading the state and immediately writing back what was read is equivalent to only reading the state. We prove this for our example in §8.4.
- Writing the state and immediately reading it again is equivalent to only writing the state. We also prove this for our example in §8.4.
- For operations that multiply a new value into a monoid: multiplying the unit value into a monoid is equivalent to doing nothing, and multiplying two values into a monoid is equivalent to multiplying by their multiplication.

Without trying to give all detail just yet, here is an example of a reasoning statement that we can get from the "theory-dependent" logic of Chapter 7. Suppose that we have a binary operation $\underline{\text{getstate}}$ and two unary operations $\underline{\text{setstate}_0}$ and $\underline{\text{setstate}_1}$ that are supposed to represent binary state. Let us abbreviate computation

$$getstate = \underline{\text{getstate}}(\text{return false}, \text{return true})$$

and let $\mathcal{A}$ be an operation-definition for $\underline{\text{getstate}}, \underline{\text{setstate}_0}, \underline{\text{setstate}_1}$ such as the one in §8.4, that satisfies

$$\underline{\text{getstate}}(\underline{\text{getstate}}(x, y), \underline{\text{getstate}}(z, w)) = \underline{\text{getstate}}(x, w) \tag{1.6}$$

which expresses that $getstate; getstate = getstate$. Let $M[\,]$ be a computation using operations $\underline{\text{getstate}}, \underline{\text{setstate}_0}, \underline{\text{setstate}_1}$ with a computation hole of type bool. Then we always know

that:

$$M[getstate; getstate] \text{ wherealg } \mathcal{A} \quad \equiv_{\varnothing} \quad M[getstate] \text{ wherealg } \mathcal{A}$$

In §8.4, we give an example of an operation-definition $\mathcal{A}$ that satisfies Equation (1.6), and a proof that it does. **Note that the analogous statement for handlers is false.** We explain this in Remark 160 on page 202.

Summarising, the logic allows us to reason in contexts where operations are defined: inside $[\ ]$ wherealg $\mathcal{A}$, we may always replace $getstate; getstate$ by $getstate$ (or reversely) without changing the meaning of the program as a whole.

Relevance

We expect the relevance of such equational reasoning to be threefold:

- they give programmers a strong foundation from which to understand their programs;
- this strong foundation may help them to refactor their programs, which makes the programs more maintainable and aids software longevity;
- equations point out potential optimisation opportunities within the compiler, which may lead to better generated code.

Remark. The initial work on effect handlers [62, 64] did consider a theory on the operations, but interest in this seems to have quickly faded. We consider past work on this in §2.5. On page 207, we hypothesize why reasoning with *defined algebraic operations* might be easier than reasoning with algebraic effect handling.

1.8 Contributions

In this thesis, we propose *defined algebraic operations*, a new model of programming, and argue that it might make it easier for programmers to write correct programs, and harder to write bugs.

The parts of this thesis come together to make this argument as follows.

1. (§1) We motivate defined algebraic operations, and give an informal introduction.

We study a concrete language with defined algebraic operations:

2. (§6.1–§6.4) We give syntax for a concrete (mathematical) programming language with defined algebraic operations.

3. (§6.5, §6.7) We give a set-based denotational semantics and an operational semantics, which we relate with an adequacy result.

4. (§6.6) We present a "base" equational logic $\equiv$ on programs that characterises much of the language, and prove it sound with respect to the denotational semantics.

We show how equational theories on the operations can be used to support more powerful reasoning about programs:

5. (§7.1–§7.2) We introduce a notion of equational theory for our notion of operations.

6. (§7.6) We say what it means for an operation definition to satisfy a theory. We give an extended "theory-dependent" equational logic $\equiv_\Theta$ on programs under an ambient theory Θ. This new logic on programs relates more programs than the base logic from §6.6. In particular, it relates additional programs that are compositions of code-using-operations with operation definitions — even when the ambient theory is empty. We give a step-by-step example over the course of §7.6, and in §8.4 we show that a definition for binary state satisfies most of the state equations.

7. (§7.5) We add structure to the set-based denotational semantics. (§7.6.5) The augmented denotational semantics validates the logic. When the ambient theory is empty, then the logic is sound for the original semantics from §6.

8. (§8) We give further examples of programs with defined algebraic operations, and we demonstrate how our logics facilitate reasoning in a novel way.

In earlier chapters, we do some supporting work:

9. (§2) We compare DAO with related work.

10. (§3) We take a look at a language for *labelled iteration*, modelling the `continue` $label;$ syntax of Java-like languages. This lets us practice a bit with lexical binding structures more generally than just variables. The *labels* introduced in §3 are similar to the operations introduced in §6.

    Labelled iteration per §3 cannot go across function boundaries; in §5 we generalise to allow this.

11. We wish to define the semantics of our language by induction on the typing derivation, but typing derivations are not unique: we need that our semantics is *coherent*. (§4) We distill the general structure of coherence proofs for the type of language that we study, from a more sophisticated situation in Biernacki and Polesiuk [13]. The language examined in §4 contains arbitrary sum and product types, no trees or iteration, but the argument does extend to trees and iteration as we see in Chapters 5 and 6.

The generalisation in §5 was to prepare for defined *parametrised* operations. In the end, parametrised operations have not been treated in this thesis, although in the conclusion in §9 we sketch a prospective to them.

Reading guide

The essence of this thesis is in §6 and §7. Chapter 7 builds on Chapter 6, but the pair itself are broadly self-containing: the reader can start reading from §6, and the essential definitions should be understandable. As we proceed through the thesis and define and prove similar things in multiple chapters, we lose some of our verbosity — assuming that the reader has read the thesis up to that point — so a reader who starts in the middle and has trouble filling in the details of some concept may find it beneficial to refer to our parallel explanation in earlier chapters.

Chapter 2

COMPARISON TO RELATED WORK

2.1 Introduction, limitations, relationships

The DAO language in this thesis is fairly simplistic compared to most handler languages in the literature. We study a terminating language, and our operations do not take or produce any data: we say that they are not *parametrised*. That is, we can express operations $\underline{\mathrm{getstate}}, \underline{\mathrm{setstate}_0}, \underline{\mathrm{setstate}_1}$ for a binary state, but for a general value type A we cannot get and set values of that type, as is commonly done in the handler literature.

There are three reasons for the simplicity of our language. Firstly, it makes the presentation purer: it lets us focus on certain aspects of our language and expose them better. Secondly, the matter of recursion seems relatively orthogonal to the problems studied in this thesis: it seems trivial to extend the semantics in §6 from sets to domains, and it seems plausible that we can augment this model to make the theory-dependent logic sound.

Thirdly: in this thesis, the arity of operations is a natural number, or equivalently, a finite set. There is a fancier notion of signature in the literature, namely one where the coarity of operations is a (sequence of) base types and where the arity of operations is a family of (sequences of) base types, so that operations additionally require parameters of the former types and generate (bind) values of the latter types. This happens for instance in Plotkin and

Pretnar [61], where the coarity of operations (for the values passed in) is a sequence of base types, and the arity of operations (the variables bound in resumptions) is a family of sequences of base types, namely one sequence of types per resumption. A variant of this idea appears in Plotkin and Pretnar [62], where the coarity and arity are single types, which must both be sums of products of base types, and the language used in effect equations may additionally construct injections and tuples, and pattern-match on them.[1]

If base types are not a concern, and if function types are not permissible in operation coarities/arities, then there is no essential gain in generalising from finitary operations to operations that are parametrised by types. As we will remark on page 111, the language in this thesis supports an arbitrary number of operations of arbitrary arity. Any type formed from merely countable sum and product types (no functions or ground types) has a countable number of elements, and we can simply replace an operation of *type* coarity/arity by a *countable family* of operations of *countable* arity. Nevertheless, in Chapter 9 we sketch some initial directions towards defined algebraic operations with type coarity/arity.

We will proceed to further relate the present work to the literature in a number of particular aspects, mainly comparing it to the literature on handlers for algebraic effects. The study of handlers for algebraic effects has intensified recently, totalling many tens of articles at present. A comprehensive review is therefore out of scope. We defer to the "collaborative bibliography of work related to the theory and practice of computational effects" maintained by Jeremy Yallop together with the community at https://github.com/yallop/effects-bibliography .

2.2 Control structures

DAO and handlers inhabit a more general field of control structures. Forster et al. [26] draw a comparison between handlers, delimited continuations, and monadic reflection: they give three concrete (typed) languages, one with each language feature. They show that the untyped

[1] To be more precise, a "signature type" in [62] is inductively either a base type, a nullary or binary product of signature types, or a finite sum of signature types. Such types are essentially the same as a finite sum of a finite product of base types, in a way that we do not make precise here.

parts of the languages are mutually macro-expressible, and one such macro-translation preserves types. They prove that in three other directions, there cannot exist a type-preserving macro-translation, and they conjecture that such type-preserving macro-translations also do not exist in the two remaining directions. However, they remark that these nonexistence results are rather sensitive to the precise collection of features present in each language.

In the author's intuition, DAO and handling can both be seen as a defunctionalised [19] form of delimited continuations with prompts. Namely, when capturing a delimited continuation, one captures a prefix of the stack (up to the prompt) and replaces it with a parametrised computation given at the capturing site. With DAO (resp. handling), calling any of the operations captures a prefix of the stack up to the operation definition (resp. handler invocation) and replaces that prefix by a parametrised computation given in the operation definition (resp. the handling site). Operation definitions and handlers can be seen as follows: they defunctionalise the set of replacement computations, couple them together with the prompt, and call them operations.

However, there is one big difference between delimited continuations on the one side and defined/handled operations on the other side: the latter have a direct-style denotational semantics (namely, trees) which the former seem to lack. As Filinski [24] has shown, delimited continuations have a compositional translation to a combination of state and undelimited continuations, which of course has a CPS semantics, although the denotational semantics of delimited continuations that we obtain using this construction is perhaps not as insightful.

The reader is reminded that some of the literature uses the phrase "composable continuations" for delimited continuations, and "uncomposable" or "ordinary" continuations for undelimited ones.

2.3 The origin of operation names

As we mentioned on page 30, the operation names in DAO are *bound* by the $\mathrm{wherealg}$ construct: if there is a $\mathrm{wherealg}$ nested in another $\mathrm{wherealg}$, then by definition those occurrences of $\mathrm{wherealg}$ handle different operations. The names of operations are inessential.

Binding as the origin of operation names is unconventional as of present; instead, the literature tends to use either *dynamically generated* operation names, or a fixed *global set* of operation names. Dynamically generated operation names can help avoid unintentional name clashes between modules of the same program, and so implementations such as Eff [8] employ them as the origin of operation names.[2] But dynamically generated operations do not seem an easy combination with effect typing, as evidenced by a lack of papers that study languages with generated operations and effect-and-type systems. (For an introduction to effect types, see for instance Kammar [39].) The complicated thing here seems to be that to typecheck effect types with generated operations, we must refer to the *value result* of a computation (generating a new operation) in the *type* of another, which is not so conventional in type systems. This is not a problem if we *bind* operation names rather than generating them: we simply have another zone in the context with operation names that we may use in types. This happens in both our Chapter 6 on defined algebraic operations, and our Chapter 5 on generalised labelled iteration.

The language of Zhang and Myers [71] also binds names, and their language has a very significant philosophical overlap with the *defined algebraic operations* in this thesis. We note that the language of Zhang and Myers is primarily based on delimited continuations with prompts (which they call *labels*), rather than a defunctionalisation (recall above). Their ⇧ syntax both binds a new prompt name and sets it on the stack, and we believe that this is an innovation that Zhang and Myers undersell, as it seems to enable for them an uncommon combination of effect typing with a non-static set of names (prompts, in their case). The author is not aware of another calculus that features effect typing with a non-static set of names for operations, apart from the ones in Chapters 5 and 6 of this thesis. Zhang and Myers proceed to introduce a way to bundle a prompt together with a "meaning", and call the resulting thing a "handler", although it is much more similar to the *defined operations* of this thesis.

The language of Zhang and Myers [71] additionally supports binding the defined operations to new names, and then using those names in the effect typing. Their notion of function can accept these "handlers" as parameters, which allows them to complete the programming

[2] Actually, in Eff operations are merely parts of a *signature* for an *effect*, and the language allows the generation of new effect *instances*. We deem the distinction between operation names and effect instances inessential.

narrative better than we could in this thesis. They argue (as we do) that (most) good uses of actual handlers are still possible in our systems; they furthermore argue that the passing of operation names as arguments eliminates a class of bugs, compared to normal handlers. Finally, their functions can be parametrised over "effect parameters", to complete the usual story of higher-order functions that can pass through the ambient effects. Another term for effect parameter in the literature is *effect row*.[3]

2.4 Macro-expressibility

We conjecture that our DAO language is macro-expressible in other languages, so one can simply macro-expand a DAO program and run it in the other implementation. In a language with global operation names like Koka [47], one could declare statically a new operation for every operation that occurs in the DAO source code, and handle that operation in exactly one place in the source code. As the type system in this thesis is rather limited — specifically, there are no effect rows as in [35] or [12] — we suspect that the type system unintentionally rules out all situations where substitution needs to be capture-avoiding for operation names in order to preserve the correct binding structure. We expect the resulting simulation to be faithful.

In a language with dynamically generated operation names like Eff [8], one could generate a new operation at load-time for every occurrence in the source code. Of course, if one executes DAO programs by macroexpansion, then the type checking must be done by hand.

If such presumed macro translations indeed exist, then the reader may wonder if *defined algebraic operations* is not really a "subset" of *handlers for algebraic operations*. The author conjectures that if additional suitable modularity features are mixed in, such as effect rows,

[3] We wish to bring to the attention of the reader that Zhang and Myers [71] write *binding* to mean two distinct concepts: (1) the introduction of a new statically-scoped prompt name, (2) the temporary assignment of a prompt name to a location on the stack, shadowing any location that the prompt name used to refer to previously. When we write *binding* in this thesis, we refer exclusively to meaning (1). The word *capability* in [71] must also not be mistaken for the security concept (namely a first-class token value that is a witness of a communication capability and that can be passed around arbitrarily); rather, a *capability* in [71] may refer to any of the three mechanisms for referring to effectful things in their language: namely prompt labels, handler variables, or effect variables.

then this macroexpansion stops being faithful, and it starts to become important to perform actual capture-avoiding substitution to calculate in the language.

The author wishes to point out that the main goal for DAO is not macro-expressivity — rather, it is finding a language that is better for writing software.

2.5 Reasoning

There has been some earlier work towards reasoning principles for handlers for algebraic effects. In some of the early work on handlers, Plotkin and Pretnar [62] explore a handling language for a global set of algebraic effects with a fixed equational theory.[4] This makes it easy to reason about code that uses operations, and the "interface" is built into the assumed global theory. The flip side of this approach is that not every handler is "correct", meaning that it cannot be given a suitable denotation that satisfies the theory. They resolve this by assigning a denotation of $\bot$ to some handlers, computation, and values. Their denotational semantics works, but is mildly unsatisfying because (as the authors explain in §6) in general it is undecidable whether a handler is correct. Any handling of a computation by an incorrect handler has semantics $\bot$, and so the presence of some incorrect handlers in a program can have quite a dramatic effect on its denotational semantics. This in turn creates a tension between denotational and operational semantics, because operationally it does not matter whether a handler is correct, and so an operational semantics would give answers in many cases where the denotational semantics does not.

After [62], the literature on handling shifts to signatures without an inherent equational theory. Some reasoning results are still obtained. Bauer and Pretnar [7] introduce an induction principle on computations, which says that inductively, every computation M in a given environment is *essentially* equal to a finite computation built from either (1) the divergent computation $\bot$, or (2) a returned value $\texttt{val}\ V$, or (3) an operation call $\alpha\, V\, (x.\, -)$ containing such a finite

[4] In our *defined algebraic operations* language, inside the scope of an operation definition, a stronger logic applies if the algebra validates extra equations (Definition 152 on page 186).

computation.[5] For details, see [7] §6.2. In their example in [7] §7.2, Bauer and Pretnar give a situation where two computations can be shown not to interfere, and show using this induction principle that the two computations commute. We do not consider such reasoning principles in this thesis; we conjecture that the analogous principle also holds in DAO.

Biernacki et al. [12] give a language with algebraic effects, handlers, and effect rows, not unlike Hillerström and Lindley [35]. Biernacki et al. introduce a form of relational parametricity for the effect rows, with which they can prove a wild variety of program equivalences, and they give a number of interesting examples. The examples show how in a sense, the effect rows are opaque, much like how values of a universally quantified type in System F are opaque; see Pitts [57].

Although Bauer and Pretnar [7] and Biernacki et al. [12] show how their techniques can be used to obtain strong program equivalences, there still seems to be considerable effort and creativity involved in setting up the proofs using their techniques. In the opinion of the author of the thesis before you, the techniques presented here for obtaining program equivalences are more pedestrian (and therefore easier) to apply.

There is some work on the combination of handling and dependent types, in particular Brady [16] in the programming language Idris [14], and Ahman [4] on the theoretical side. Building on top of this, Brady [15] uses GADTs [22] to devise a notion of operation whose invocation causes the effect signature to change, in a similar spirit to the typestate of Aldrich et al. [5] or the parametrised monads of Atkey [6]. It would be interesting to see a theoretical account of these in the context of either handlers or DAO, or an account of the interaction of DAO with dependent types.

[5] Actually, the syntax in Eff is $\iota\#\mathtt{op}\,V\,(x.\,M)$ for an operation op on an instance ι. Their syntax $\mathtt{val}\ V$ corresponds to our syntax $\mathrm{return}\ V$.

Chapter 3

ITERATION AND FIRST-ORDER LABELLED ITERATION

The material in this chapter is based on joint work with the author's supervisor, Paul Levy, and is a close adaptation of our paper "Iteration and Labelled Iteration" *published in the proceedings of MFPS'16 [30].*

We do not consider coherence of the denotational semantics in this chapter. However, in §5.5, we prove the coherence of a the denotational semantics of a larger language, which implies the coherence of the denotational semantics of §3.3. The language in §3.2 is merely a sublanguage of this (as we show in §3.3.4), and therefore its denotational semantics is also coherent.

Goncharov et al. [32] follow the approach presented in this chapter in giving a metalanguage for guarded iteration, and proceed to use that to study the semantics of guarded iteration.

The language in this chapter is extended in §5, which adds effectful function types.

3.1 Introduction

3.1.1 Overview

Iteration is an important programming language feature.

- In imperative languages, it is best known in `for` and `while` loops. The meaning of such a loop is to iterate code until some condition is met, or if the condition is never met, the loop diverges. Such loops are often supplemented by `break` and `continue`.
- It has also been studied in the lambda calculus setting [37, 45, 49].
- In the categorical setting, iteration corresponds to complete Elgot monads [31]. They descend from iterative, iteration, and Elgot theories, and their algebras and monads [23, 1, 2, 3, 53], which study variants of the sum-based iteration $-^\dagger$. This field is related to Kleene monads [33, 43, 44].

Iteration can be implemented using recursion, but it is simpler: semantics of recursion require a least fixpoint, where iteration has a simple set-based semantics. Also from the programmer's perspective, iteration and recursion are different: a program using a `for` or `while` loop can sometimes be clearer than the same program using recursion.

3.1.2 The sum-based representation of iteration

We study two representations of iteration. First, the classical sum-based construct $-^\dagger$ that turns a computation $\Gamma, A \vdash M : A + B$ into a computation $\Gamma, A \vdash M^\dagger : B$. Categorically, this representation of iteration corresponds to complete Elgot monads [31]. To understand the correspondence better, we introduce a term constructor iter for $-^\dagger$. (Details are in Section 3.2.)

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad \Gamma, x{:}A \vdash_{\mathrm{c}} M : A + B}{\Gamma \vdash_{\mathrm{c}} \mathrm{iter}\ V,\ x.\ M : B}$$

Imperative programs with `for` and `while` can now be encoded using iter. As an example, the program on the left corresponds to the term on the right:

imperative	λ-calculus-like
$x := V;$	
while $(p(x))$ {	iter $V, x.$
$\quad x := f(x);$	$\quad$ if $\;p(x)$
}	$\quad$ then return inl $\;f(x)$
return $g(x);$	$\quad$ else $\;$ return inr $g(x)$

This works as follows. The iter construct introduces a new variable x, which starts at V. The body is evaluated. If the body evaluates to $\mathrm{inr}\ W$, then the loop is finished and its result is W. If the body evaluates to $\mathrm{inl}\ V'$, then we set x to V', and keep on evaluating the body until it evaluates to some $\mathrm{inr}\ W$.

3.1.3 The "De Bruijn index" awkwardness with the sum-based representation

Programmers using imperative languages regularly use nested loops, as well their associated `break` and `continue` statements, which may be labelled. Such statements are not essential for programming, and code using `break` or `continue` can be rewritten so it does not use either statement, but this usually comes at a price in readability. There is usually a labelled and an unlabelled form of `break` and `continue`.

On the left side of Figure 3.1, we show an program in a Java-like language with nested labelled loops, and labelled `continue` statements. The colours can be ignored for now. The program computes the formula $\sum_{\substack{0\leq i\leq 8 \\ \wedge\, a[i][0]\neq 5}} \prod_{\substack{0\leq j\leq 8 \\ \wedge\, a[i][j] \text{ even}}} a[i][j]$, although the specific formula is not important for the example. Recall from Java that "$\mathrm{continue\ inner}$" aborts the current iteration of the inner loop, and continues with a fresh iteration of the inner while loop. Statement "$\mathrm{continue\ outer}$" does the analogous thing but for the outer loop. It will abort the inner loop implicitly.

```
int sum = 0;
outer: for (int i = 0; i ≤ 8; i++){
    if (a[i][0] == 5)
        continue outer;
    int prod = 1;
inner: for (int j = 0; j ≤ 8; j++){
        if (¬isEven(a[i][j]))
            continue inner;
        if (a[i][j] == 0)
            // product will be 0.
            continue outer;
        prod = prod * a[i][j];
    }
    sum = sum + prod;
}
int result = sum;
```

```
iter ⟨0, 0⟩, ℓ1.  let ⟨sum, i⟩ = ℓ1 in
  if i ≥ 9 then
    return inr sum
  else if a[i][0] == 5 then
    return inl ⟨sum, i + 1⟩
  else iter ⟨1, 0⟩, ℓ2.  let ⟨prod, j⟩ = ℓ2 in
    if j ≥ 9 then
      return inr inl ⟨sum + prod, i + 1⟩
    else if ¬isEven a[i][j] then
      return inl ⟨prod, j + 1⟩
    else if a[i][j] == 0 then
      return inr inl ⟨sum, i + 1⟩
    else
      return inl ⟨prod * a[i][j], j + 1⟩
```

Figure 3.1: Two programs programs that compute the same formula. The left hand program is written in Java; the right hand program uses fine-grain call-by-value with *sum-based* iteration. Related fragments have the same colour. Both programs compute $\sum_{\substack{0\leq i\leq 8 \\ \wedge\, a[i][0]\neq 5}} \prod_{\substack{0\leq j\leq 8 \\ \wedge\, a[i][j] \text{ even}}} a[i][j]$.

On the right side, we have a similar program to compute the same formula, but using sum-based iteration.

We use colour to indicate fragments that are intuitively related, because control flows to the same place after those fragments:

- After continue inner and the two occurrences of $\mathrm{return\ inl}\ \langle \cdots, j+1 \rangle$, control flows to the beginning of the inner loop. We have drawn a solid purple box around those fragments. The assignment to $prod$ on the left also precedes the beginning of the inner loop, and we have coloured it purple.

- After both occurrences of continue outer, control flows to the beginning of the outer for loop. Similarly, after $\mathrm{return\ inl}\ \langle sum, i+1 \rangle$ and after the two occurrences of $\mathrm{return\ inr\ inl}\ \langle \cdots \rangle$, control flows to the beginning of the outer iter. We have drawn a thick dashed red box around those fragments. The assignment to sum on the left also precedes the beginning of the outer loop, and we have coloured it red.

Note that in the left (Java) program, both statements in red boxes are written the same: "continue outer".

Both programs work, but the syntax of the right hand program has two awkwardnesses for programmers:

1. Continuing to the outer loop (red) is written $\mathrm{return\ inl}\ \langle \cdots \rangle$ in one case, and $\mathrm{return\ inr\ inl}\ \langle \cdots \rangle$ in the other cases. The same "control fragment" is written differently depending on where it occurs. This makes moving code into and out of the inner loop error-prone.

2. $\mathrm{return\ inl}\ \langle \cdots \rangle$ is used to resume both the inner and the outer iteration. To find out where control resumes, a reader of the program must carefully look up the innermost enclosing iteration. In contrast, in the Java program there can be no mistake about where control resumes after continue outer.

This awkwardness is exacerbated when there are three or more nested loops with the same structure: on the right hand side,

- $\mathrm{return\ inl}\ (\cdots)$ would continue the *innermost enclosing* iteration
- $\mathrm{return\ inr\ inl}\ (\cdots)$ would continue the *second-innermost enclosing* iteration
- $\mathrm{return\ inr\ inr\ inl}\ (\cdots)$ would continue the *third-innermost enclosing* iteration, which would always be the outer iteration.

We call this the "De Bruijn index awkwardness", because De Bruijn's indices [20] for variables in λ calculus also work by counting intermediate binders, and they have similar disadvantages for programmers. Indeed, De Bruijn [20] claims his notation to be good for a number of things, but does not claim that it is "easy to write and easy to read for the human reader". For a brief introduction to De Bruijn indices, we refer to [38]; the same issue has also been studied from a different angle in [11, 52].

3.1.4 The solution: Labelled iteration

We solve the De Bruijn index awkwardness with a second iteration construct, which we call labelled iteration and which we will also spell iter. It binds a name $x : A$ with a dual purpose:

- It holds a *value* of type A.
- It serves as a label for *restarting the loop*, upon which a new value of type A must be supplied.

In Figure 3.2, we have put the same Java program side-by-side side with an implementation using labelled iteration.

Like sum-based iteration, labelled iteration has a set-based semantics, but the type system is more involved. We explain labelled iteration in more detail in Section 3.3. We chose the spelling raise because there is a similarity with raising an exception; see also the discussion in Section 3.4.

```
int sum = 0;
outer: for (int i = 0; i ≤ 8; i++){
    if (a[i][0] == 5)
        continue outer;
    int prod = 1;
    inner: for (int j = 0; j ≤ 8; j++){
        if (¬isEven(a[i][j]))
            continue inner;
        if (a[i][j] == 0)
            // product will be 0.
            continue outer;
        prod = prod * a[i][j];
    }
    sum = sum + prod;
}
int result = sum;
```

```
iter ⟨0, 0⟩, ℓ1.  let ⟨sum, i⟩ = ℓ1 in
    if i ≥ 9 then
        return sum
    else if a[i][0] == 5 then
        raise_ℓ1 ⟨sum, i + 1⟩
    else iter ⟨1, 0⟩, ℓ2.  let ⟨prod, j⟩ = ℓ2 in
        if j ≥ 9 then
            raise_ℓ1 ⟨sum + prod, i + 1⟩
        else if ¬isEven a[i][j] then
            raise_ℓ2 ⟨prod, j + 1⟩
        else if a[i][j] == 0 then
            raise_ℓ1 ⟨sum, i + 1⟩
        else
            raise_ℓ2 ⟨prod * a[i][j], j + 1⟩
```

Figure 3.2: Two programs programs that compute the same formula. The left hand program is written in Java; the right hand program uses fine-grain call-by-value with *labelled* iteration. Related fragments have the same colour. Both programs compute $\sum_{\substack{0 \le i \le 8 \\ \wedge\, a[i][0] \neq 5}} \prod_{\substack{0 \le j \le 8 \\ \wedge\, a[i][j] \text{ even}}} a[i][j]$.

3.1.5 Chapter summary

We define both languages: we give a type system, denotational semantics, big-step operational semantics, and an adequacy theorem for both languages. We explain the De Bruijn index awkwardness with the first language, and give a realistic example. We show that the first construct can be macro-expressed in terms of the second construct.

For both types of iteration, we study only loops with `continue`: we omit `break` because we believe it is a straightforward extension.

We define the language with sum-based iteration in Section 3.2, and the language with labelled iteration in Section 3.3. The labelled iteration language in this chapter does not supports iteration across function call boundaries; this will be added in Chapter 5.

3.2 Sum-based iteration

We define both our constructs in terms of fine-grain call-by-value or FGCBV [48], which is a variant of call-by-value lambda calculus that has a syntactic separation between values and computations, and in which the evaluation order is made explicit.

We explain FGCBV and sum-based iteration here. The syntax and type system of FGCBV is given in Figure 3.3. We give a simple set-based semantics with divergence:

$$\begin{aligned} [\![1]\!] &= \{\star\} \\ [\![\mathrm{nat}]\!] &= \mathbb{N} \\ [\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![A]\!] \to ([\![B]\!] + \{\bot\}) \end{aligned}$$

$$[\![\Gamma]\!] \;=\; \prod\nolimits_{(x:A)\in\Gamma} [\![A]\!]$$

$$[\![\Gamma \vdash_{\mathrm{v}} V : A]\!] \in [\![\Gamma]\!] \to [\![A]\!]$$

$$[\![\Gamma \vdash_{\mathrm{c}} M : A]\!] \in [\![\Gamma]\!] \to ([\![A]\!] + \{\bot\})$$

$$\begin{array}{lll}
\text{values} & V, W & ::= x \mid \langle\rangle \mid 0 \mid \text{succ}\ V \mid \text{inl}\ V \mid \text{inr}\ V \mid \langle V, W\rangle \mid \lambda x.\ M \\
\text{computations} & M, N & ::= \text{return}\ V \mid \text{let}\ V\ \text{be}\ x.\ M \mid M\ \text{to}\ x.\ N \\
& & \quad \mid V\ W \mid \text{case}\ V\ \text{of}\ \{0.\ M;\ \text{succ}\ x.\ N\} \\
& & \quad \mid \text{case}\ V\ \text{of}\ \{\text{inl}\ x.\ M;\ \text{inr}\ y.\ N\} \mid \text{case}\ V\ \text{of}\ \langle x, y\rangle.\ M \\
\text{types} & A, B, C & ::= 1 \mid \text{nat} \mid A + B \mid A \times B \mid A \to B
\end{array}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\text{v}} x : A} \qquad \frac{}{\Gamma \vdash_{\text{v}} \langle\rangle : 1} \qquad \frac{}{\Gamma \vdash_{\text{v}} 0 : \text{nat}} \qquad \frac{\Gamma \vdash_{\text{v}} V : \text{nat}}{\Gamma \vdash_{\text{v}} \text{succ}\ V : \text{nat}}$$

$$\frac{\Gamma \vdash_{\text{v}} V : A}{\Gamma \vdash_{\text{v}} \text{inl}\ V : A + B} \qquad \frac{\Gamma \vdash_{\text{v}} V : B}{\Gamma \vdash_{\text{v}} \text{inr}\ V : A + B} \qquad \frac{\Gamma \vdash_{\text{v}} V : A \qquad \Gamma \vdash_{\text{v}} W : B}{\Gamma \vdash_{\text{v}} \langle V, W\rangle : A \times B}$$

$$\frac{\Gamma \vdash_{\text{v}} V : A}{\Gamma \vdash_{\text{c}} \text{return}\ V : A} \qquad \frac{\Gamma \vdash_{\text{v}} V : A \qquad \Gamma, x : A \vdash_{\text{c}} M : B}{\Gamma \vdash_{\text{c}} \text{let}\ V\ \text{be}\ x.\ M : B} \qquad \frac{\Gamma \vdash_{\text{c}} M : A \qquad \Gamma, x : A \vdash_{\text{c}} N : B}{\Gamma \vdash_{\text{c}} M\ \text{to}\ x.\ N : B}$$

$$\frac{\Gamma, x : A \vdash_{\text{c}} M : B}{\Gamma \vdash_{\text{v}} \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash_{\text{v}} V : A \to B \qquad \Gamma \vdash_{\text{v}} W : A}{\Gamma \vdash_{\text{c}} V\ W : B}$$

$$\frac{\Gamma \vdash_{\text{v}} V : \text{nat} \qquad \Gamma \vdash_{\text{c}} M : C \qquad \Gamma, x : \text{nat} \vdash_{\text{c}} N : C}{\Gamma \vdash_{\text{c}} \text{case}\ V\ \text{of}\ \{0.\ M;\ \text{succ}\ x.\ N\} : C}$$

$$\frac{\Gamma \vdash_{\text{v}} V : A + B \qquad \Gamma, x : A \vdash_{\text{c}} M : C \qquad \Gamma, y : B \vdash_{\text{c}} N : C}{\Gamma \vdash_{\text{c}} \text{case}\ V\ \text{of}\ \{\text{inl}\ x.\ M;\ \text{inr}\ y.\ N\} : C}$$

$$\frac{\Gamma \vdash_{\text{v}} V : A \times B \qquad \Gamma, x : A, y : B \vdash_{\text{c}} M : C}{\Gamma \vdash_{\text{c}} \text{case}\ V\ \text{of}\ \langle x, y\rangle.\ M : C}$$

Addition for sum-based iteration

$$\frac{\Gamma \vdash_{\text{v}} V : A \qquad \Gamma, x : A \vdash_{\text{c}} M : A + B}{\Gamma \vdash_{\text{c}} \text{iter}\ V,\ x.\ M : B}$$

Figure 3.3: Above: syntax of plain fine-grain call-by-value. Sum-based iteration adds only one term construct and no types; the typing rule for this term is given below.

Fine-grain call-by-value

$$[\![\langle V, W\rangle]\!]_\rho = \langle [\![V]\!]_\rho, [\![W]\!]_\rho\rangle$$

$$[\![x]\!]_\rho = \rho(x) \qquad [\![\lambda x.\ M]\!]_\rho = \lambda(a{\in}[\![A]\!]).[\![M]\!]_{(\rho,x\mapsto a)}$$

$$[\![\langle\rangle]\!]_\rho = \langle\rangle \qquad [\![\text{return } V]\!]_\rho = \text{inl } [\![V]\!]_\rho$$

$$[\![0]\!]_\rho = 0 \qquad [\![\text{let } V \text{ be } x.\ M]\!]_\rho = [\![M]\!]_{(\rho,x\mapsto[\![V]\!]_\rho)}$$

$$[\![\text{succ } V]\!]_\rho = 1 + [\![V]\!]_\rho \qquad [\![M \text{ to } x.\ N]\!]_\rho = \begin{cases} [\![N]\!]_{(\rho,x\mapsto v)} & \text{if } [\![M]\!]_\rho = \text{inl } v \\ \text{inr } \bot & \text{if } [\![M]\!]_\rho = \text{inr } \bot \end{cases}$$

$$[\![\text{inl } V]\!]_\rho = \text{inl } [\![V]\!]_\rho \qquad [\![V\ W]\!]_\rho = [\![V]\!]_\rho\ [\![W]\!]_\rho$$

$$[\![\text{inr } V]\!]_\rho = \text{inr } [\![V]\!]_\rho$$

$$[\![\text{case } V \text{ of } \{0.\ M;\ \text{succ } x.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_\rho & \text{if } [\![V]\!]_\rho = 0 \\ [\![N]\!]_{(\rho,x\mapsto n)} & \text{if } [\![V]\!]_\rho = 1+n \end{cases}$$

$$[\![\text{case } V \text{ of } \{\text{inl } x.\ M;\ \text{inr } y.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_{(\rho,x\mapsto a)} & \text{if } [\![V]\!]_\rho = \text{inl } a \\ [\![N]\!]_{(\rho,y\mapsto b)} & \text{if } [\![V]\!]_\rho = \text{inr } b \end{cases}$$

$$[\![\text{case } V \text{ of } \{\langle x, y\rangle.\ M\}]\!]_\rho = [\![M]\!]_{(\rho,x\mapsto a,y\mapsto b)} \qquad \text{if } [\![V]\!]_\rho = \langle a, b\rangle$$

Addition for sum-based iteration

$$[\![\text{iter } V, x.\ M]\!]_\rho = \begin{cases} \text{inl } w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho,x\mapsto v_i)} = \text{inl inl } v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho,x\mapsto v_k)} = \text{inl inr } w \\ \text{inr } \bot & \text{if no such } v_{0..k} \text{ exists} \end{cases}$$

Figure 3.4: Denotational semantics of values and computations in fine-grain call-by-value, and semantics of the sum-based iteration construct.

The semantics of plain FGCBV and FGCBV with sum-based iteration are the same, except of course that the latter has an extra construct. We give big-step operational semantics for both languages in Figure 3.5. The adequacy statements are simple:

Proposition 1 (adequacy).

1. For each closed computation M of plain FGCBV without iteration, there is a unique V such that $M \Downarrow \mathrm{return}\ V$, and $[\![M]\!]_\varnothing = \mathrm{inl}\ [\![V]\!]_\varnothing$.

2. For each closed computation M of FGCBV with sum-based iteration, either

   - there is a unique V such that $M \Downarrow \mathrm{return}\ V$, and $[\![M]\!]_\varnothing = \mathrm{inl}\ [\![V]\!]_\varnothing$, or
   - M does not reduce to a terminal, and $[\![M]\!]_\varnothing = \mathrm{inr}\ \bot$.

3.3 Labelled iteration with pure function types

3.3.1 Introduction

To fix the De Bruijn index awkwardness indicated in Section 3.1.3, we now give a language that has an effectful "labelled iteration" construct instead. Labels can be propagated across sequencing, but not across function boundaries — functions can still only return a value: they are "pure". In Chapter 5, we extend this language to allow labels to be propagated across function boundaries.

The judgements in the language of this section are

$$\Delta; \Gamma \vdash_{\mathrm{c}} M : A \qquad \text{for computations}$$

$$\Gamma \vdash_{\mathrm{v}} V : A \qquad \text{for values}$$

We give the typing rules in Figure 3.6. *Value context* Γ is a context of variables bound to values, as usual. *Label context* Δ exists only for computations; it is a context of *typed labels*. Computations may raise any label in their label context Δ, together with a value of the associated type. Values cannot raise any labels; see Chapter 5 for a generalisation that allows raising labels across function boundaries.

Fine-grain call-by-value

$T ::= \text{return } V$

$$\frac{}{\text{return } V \Downarrow \text{return } V} \qquad \frac{M[V/x] \Downarrow T}{\text{let } V \text{ be } x.\ M \Downarrow T} \qquad \frac{M \Downarrow \text{return } V \qquad N[V/x] \Downarrow T}{M \text{ to } x.\ N \Downarrow T}$$

$$\frac{M[W/x] \Downarrow T}{(\lambda x.\ M)\ W \Downarrow T} \qquad \frac{M[V/x, W/y] \Downarrow T}{\text{case } \langle V, W\rangle \text{ of } \{\langle x, y\rangle.\ M\} \Downarrow T}$$

$$\frac{M_0 \Downarrow T}{\text{case } 0 \text{ of } \{0.\ M_0;\ \text{succ } x.\ M_{\text{succ}}\} \Downarrow T} \qquad \frac{M_{\text{succ}}[V/x] \Downarrow T}{\text{case } (\text{succ } V) \text{ of } \{0.\ M_0;\ \text{succ } x.\ M_{\text{succ}}\} \Downarrow T}$$

$$\frac{M_{\text{inl}}[V/x] \Downarrow T}{\text{case } (\text{inl } V) \text{ of } \{\text{inl } x.\ M_{\text{inl}}; \text{inr } x.\ M_{\text{inr}}\} \Downarrow T} \qquad \frac{M_{\text{inr}}[V/x] \Downarrow T}{\text{case } (\text{inr } V) \text{ of } \{\text{inl } x.\ M_{\text{inl}}; \text{inr } x.\ M_{\text{inr}}\} \Downarrow T}$$

Addition for sum-based iteration

$T ::= \text{return } V$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \text{return inl } V_i \qquad M[V_k/x] \Downarrow \text{return inr } Z}{\text{iter } V_0,\ x.\ M \Downarrow \text{return } Z}$$

Figure 3.5: Big-step operational semantics of plain fine-grain call-by-value and of sum-based iteration.

In our operational semantics, closed computations reduce to "terminal" computations of the same type, or they do not reduce at all. We use metavariable T for terminals. For FGCBV and its extension with sum-based iteration, terminal computations are always of the form $\text{return } V$. Introducing a separate notion of terminals might seem odd for now, but in Figure 3.8 we extend the rules for FGCBV and add another form of terminal. So T above may come to stand for something other than $\text{return } V$ further in this chapter.

Values and types are the same as in fine-grain call-by-value in Figure 3.3 on page 59.

computations $\quad M, N \ ::= \ \cdots \mid \mathrm{iter}\ V, x.\ M \mid \mathrm{raise}_x\ V$

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash_{\mathrm{v}} x : A} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{let}\ V\ \mathrm{be}\ x.\ M : B}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{return}\ V : A} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{c}} M : A \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} N : B}{\Delta; \Gamma \vdash_{\mathrm{c}} M\ \mathrm{to}\ x.\ N : B}$$

$$\frac{\cdot\ ;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Gamma \vdash_{\mathrm{v}} \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A \to B \qquad \Gamma \vdash_{\mathrm{v}} W : A}{\Delta; \Gamma \vdash_{\mathrm{c}} V\ W : B}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Gamma \vdash_{\mathrm{v}} \mathrm{inl}\ V : A + B} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : B}{\Gamma \vdash_{\mathrm{v}} \mathrm{inr}\ V : A + B}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : \mathrm{nat} \qquad \Delta; \Gamma \vdash_{\mathrm{c}} M : C \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} N : C}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{case}\ V\ \mathrm{of}\ \{0.\ M;\ \mathrm{succ}\ x.\ N\} : C}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A + B \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : C \qquad \Delta;\ \Gamma, y{:}B \vdash_{\mathrm{c}} N : C}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{case}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\ M;\ \mathrm{inr}\ y.\ N\} : C}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A \times B \qquad \Delta;\ \Gamma, x{:}A, y{:}B \vdash_{\mathrm{c}} M : C}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{case}\ V\ \mathrm{of}\ \langle x, y\rangle.\ M : C}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta, x{:}A;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{iter}\ V, x.\ M : B} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad (x{:}A) \in \Delta}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{raise}_x\ V : B}$$

Figure 3.6: Syntax of labelled iteration.

Denotations of judgements are

$$[\![\Delta; \Gamma \vdash_{\mathrm{c}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to \Big(\sum_{(y:C)\in\Delta} [\![C]\!] + [\![A]\!] + \{\bot\}\Big)$$

$$[\![\Gamma \vdash_{\mathrm{v}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to [\![A]\!] \,.$$

Value context Γ is used to form values.

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash_{\mathrm{v}} x : A}$$

Label context Δ is used to form computations, much like raising an exception. However, conventionally, exception names come from a global set. Our "exception names", which we call labels, will be bound in the same way that variables are bound by λ.

Furthermore, when a label is raised, it must be parametrised by a value of the corresponding type. The typing rule is as follows:

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad (x{:}A) \in \Delta}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{raise}_x\ V : B}$$

We thus have these judgements.

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash_{\mathrm{c}} \mathrm{raise}_x\ \langle 3, \mathrm{true}\rangle : \mathrm{string}$$

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash_{\mathrm{c}} \mathrm{raise}_x\ \langle y, z\rangle : 0$$

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash_{\mathrm{c}} \mathrm{return}\ y : \mathrm{nat}$$

But we cannot raise variables:

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \not\vdash_{\mathrm{c}} \mathrm{raise}_y\ 3$$

And we can also not use labels for their value:

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \not\vdash_{\mathrm{c}} \mathrm{return}\ x : \mathrm{nat} \times \mathrm{bool}$$

Indeed, the typing rule of return (see Figure 3.6 on the previous page) shows that x is not available in the context of the argument to return:

$$\frac{y{:}\mathrm{nat}, z{:}\mathrm{bool} \ \vdash_{\mathrm{v}}\ V : \mathrm{nat} \times \mathrm{bool}}{(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \ \vdash_{\mathrm{c}}\ \mathrm{return}\ V : \mathrm{nat} \times \mathrm{bool}}$$

Our use of a syntactically separate kind of names bears resemblance to the use of function names by Kennedy [42] for control.

Labelled iteration

We now wish to use labels to generalise the $\mathrm{iter}\ V, x.\ M$ from last section. Remember that previously when M reduces to

- $\mathrm{return}\ \mathrm{inl}\ V'$, then the loop should be re-tried with value V',
- $\mathrm{return}\ \mathrm{inr}\ W$, then the result of the loop is W.

Our new notation will *also* be $\mathrm{iter}\ V, x.\ M$. However, here x is *both* a variable and a label:

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta, x{:}A;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{iter}\ V, x.\ M : B}$$

Now similarly when writing $\mathrm{iter}\ V, x.\ M$, when M reduces to

- $\mathrm{raise}_x\ V'$, then the loop should be re-tried with value V',
- $\mathrm{raise}_y\ W,\ (y \neq x)$ then the loop should be aborted and loop y should be re-tried with value W,
- $\mathrm{return}\ W$, then the result of the loop is W.

We wish to repeat that the same name x can appear in *both* Δ and Γ. We pose no general syntactic restriction on $(x{:}A) \in \Delta$ and $(x{:}B) \in \Gamma$ to have the same type. However, to be able to form $\mathrm{iter}\ V, x.\ M$, we must have x in both Δ and Γ of the same type.

We also wish to note at this point that we define the semantics of our language on the *binding diagrams*[25], that is, on the abstract syntax modulo α-equivalence.

Labelled iteration and λ

Now that contexts for computations are different from contexts for values, the conventional fine-grain call-by-value judgements have to be tweaked to work in this setting. The typing rule for return in Figure 3.6 is simple: when we move upwards from a computation to a value judgement we just forget about Δ.

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{return}\ V : A}$$

But reversely, for λ, we have a choice: what should Δ be? We take what seems to be a reasonable choice: to reset Δ to the empty context, $\cdot$.

$$\frac{\cdot\,;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Gamma \vdash_{\mathrm{v}} \lambda x.\ M : A \to B}$$

Java agrees with this choice: it is a syntax error to write a labelled `continue` or `break` with a label outside of the current method [34]. From a programmer's perspective, this means that all functions are pure. Later in Chapter 5, we will explore a generalisation where functions do not have to be pure and which allows Δ to propagate upwards across λ.

3.3.2 Denotational semantics

Recall that the semantics of computation and value judgements is as follows.

$$[\![\Delta; \Gamma \vdash_{\mathrm{c}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to \Big(\sum_{(y:C)\in\Delta} [\![C]\!] + [\![A]\!] + \{\bot\}\Big)$$

$$[\![\Gamma \vdash_{\mathrm{v}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to [\![A]\!]$$

The denotation of types is as follows.

$$[\![1]\!] = \{\star\}$$

$$[\![\mathrm{nat}]\!] = \mathbb{N}$$

$$[\![A + B]\!] = [\![A]\!] + [\![B]\!]$$

$$[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \to B]\!] = [\![A]\!] \to ([\![B]\!] + \{\bot\})$$

We give the semantics of computations and values in Figure 3.7. We use the following notation for elements of the ternary sum $(\sum_{(x:B)\in\Delta} [\![B]\!] + [\![A]\!] + \{\bot\})$:

1. $\mathit{return}\ a$ (for $a \in [\![A]\!]$) (compare to the term notation: $\mathrm{return}\ V$),
2. $\mathit{raise}_x\ b$ (for $b \in [\![B]\!]$) (compare to the term notation: $\mathrm{raise}_x\ V$),
3. $\bot$.

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\langle\rangle]\!]_\rho &= \langle\rangle \\
[\![0]\!]_\rho &= 0 \\
[\![\text{succ } V]\!]_\rho &= 1 + [\![V]\!]_\rho \\
[\![\text{inl } V]\!]_\rho &= \text{inl } [\![V]\!]_\rho \\
[\![\text{inr } V]\!]_\rho &= \text{inr } [\![V]\!]_\rho \\
[\![\langle V, W\rangle]\!]_\rho &= \langle [\![V]\!]_\rho, [\![W]\!]_\rho\rangle \\
[\![\lambda x.\ M]\!]_\rho &= \lambda(a{\in}[\![A]\!]).[\![M]\!]_{(\rho, x\mapsto a)}
\end{aligned}$$

$$\begin{aligned}
[\![\text{return } V]\!]_\rho &= \mathit{return}\ [\![V]\!]_\rho \\
[\![\text{raise}_x\ V]\!]_\rho &= \mathit{raise}_x\ [\![V]\!]_\rho \\
[\![\text{let } V \text{ be } x.\ M]\!]_\rho &= [\![M]\!]_{(\rho, x\mapsto [\![V]\!]_\rho)} \\
[\![M \text{ to } x.\ N]\!]_\rho &= \begin{cases} [\![N]\!]_{(\rho, x\mapsto v)} & \text{if } [\![M]\!]_\rho = \mathit{return}\ v \\ \mathit{raise}_y\ w & \text{if } [\![M]\!]_\rho = \mathit{raise}_y\ w \\ \bot & \text{if } [\![M]\!]_\rho = \bot \end{cases} \\
[\![V\ W]\!]_\rho &= [\![V]\!]_\rho\ [\![W]\!]_\rho
\end{aligned}$$

$$[\![\text{case } V \text{ of } \{0.\ M;\ \text{succ } x.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_\rho & \text{if } [\![V]\!]_\rho = 0 \\ [\![N]\!]_{(\rho, x\mapsto n)} & \text{if } [\![V]\!]_\rho = 1 + n \end{cases}$$

$$[\![\text{case } V \text{ of } \{\text{inl } x.\ M;\ \text{inr } y.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_{(\rho, x\mapsto a)} & \text{if } [\![V]\!]_\rho = \text{inl } a \\ [\![N]\!]_{(\rho, y\mapsto b)} & \text{if } [\![V]\!]_\rho = \text{inr } b \end{cases}$$

$$[\![\text{case } V \text{ of } \{\langle x, y\rangle.\ M\}]\!]_\rho = [\![M]\!]_{(\rho, x\mapsto a, y\mapsto b)} \qquad \text{if } [\![V]\!]_\rho = \langle a, b\rangle$$

$$[\![\text{iter } V, x.\ M]\!]_\rho = \begin{cases} \mathit{return}\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_x\ v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{return}\ w \\ \mathit{raise}_y\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_x\ v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{raise}_y\ w \\ \bot & \text{if no other case matches} \end{cases}$$

Figure 3.7: Denotational semantics of terms and values of the language with labelled iteration. See also §3.3.2.

Definition 2 (weakening)**.** We say that $\Delta'; \Gamma'$ is *stronger* than $\Delta; \Gamma$ when $\Delta' \subseteq \Delta$ and $\Gamma' \subseteq \Gamma$. Alternatively, we say that $\Delta; \Gamma$ is *weaker* than $\Delta'; \Gamma'$.

A term in a context is also a term in a weaker context, with the same derivation.

Definition 3 (closedness)**.**

1. When $\cdot \vdash_{\mathrm{v}} V : A$, then we say that V is *closed*.
2. When $\Delta; \cdot \vdash_{\mathrm{c}} M : A$, then we say that M is *closed*.

Definition 4. A *substitution* (between two-zone contexts) $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ consists of two parts,

- for every label $(x : A) \in \Delta'$, a *label* $\sigma_{\mathrm{lab}}(x)$ of type A in Δ, and
- for every variable $(x : A) \in \Gamma'$, a *value* $\sigma_{\mathrm{id}}(x) \quad (\Gamma \vdash_{\mathrm{v}} \sigma_{\mathrm{id}}(x) : A)$.

Remark. From a two-zone substitution $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ we can trivially obtain a one-zone substitution $\Gamma' \to \Gamma$. By abuse of notation, we also write σ for this obtained substitution on one-zone contexts. Similarly, from a one-zone substitution $\sigma : \Gamma' \to \Gamma$, we obtain trivially a two-zone substitution $\cdot\,; \Gamma' \to \cdot\,; \Gamma$, for which we also write σ.

We can use a substitution $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ as follows on computations. Given a computation $\Delta'; \Gamma' \vdash_{\mathrm{c}} M : A$, we obtain the computation $\Delta; \Gamma \vdash_{\mathrm{c}} M\sigma : A$ by

- for any $x \in \Delta$, replacing all occurrences of $\mathrm{raise}_x\ V$ (where x is free) by $\mathrm{raise}_{\sigma_{\mathrm{lab}}(x)}\ (V\sigma)$, where $V\sigma$ is given similarly by induction. And
- for any $x \in \Gamma$, replacing all value occurrences of variables by $\sigma_{\mathrm{id}}(x)$.

For one-zone contexts Γ we have the usual notion of substitution $\sigma : \Gamma' \to \Gamma$ that assigns a value (over Γ) to each variable of Γ'. And given $\Gamma' \vdash_{\mathrm{v}} V : A$, we obtain similarly $\Gamma \vdash_{\mathrm{v}} V\sigma : A$.

Two-zone contexts and their substitutions form a category, and one-zone contexts and their substitutions form another category. That is, substitutions can be composed associatively and composition has an identity.

$$T \ ::= \ \mathrm{return}\ V \mid \mathrm{raise}_x\ V$$

$$\frac{M \Downarrow \mathrm{raise}_x\ V}{M\ \mathrm{to}\ x.\ N \Downarrow \mathrm{raise}_x\ V}$$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \mathrm{raise}_x\ V_i \qquad M[V_k/x] \Downarrow \mathrm{return}\ Z}{\mathrm{iter}\ V_0,\ x.\ M \Downarrow \mathrm{return}\ Z}$$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \mathrm{raise}_x\ V_i \qquad M[V_k/x] \Downarrow \mathrm{raise}_y\ Z}{\mathrm{iter}\ V_0,\ x.\ M \Downarrow \mathrm{raise}_y\ Z}\ (x \neq y)$$

Figure 3.8: Big-step operational semantics for labelled iteration. This figure extends Figure 3.5. Namely, we add rules, and we add a new form of terminal: $\mathrm{raise}_x\ V$.

Lemma 5 (substitution lemma).

1. Let one-zone substitution $\sigma : \Gamma' \to \Gamma$ be given. If $\Gamma' \vdash_{\mathrm{v}} V : A$, then
   $[\![V\sigma]\!]_\rho = [\![V]\!]_{(x \mapsto [\![\sigma(x)]\!]_\rho)_{x \in \Gamma'}}$.

2. Let two-zone substitution $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ be given.
   If $\Delta'; \Gamma' \vdash_{\mathrm{c}} M : A$, then $[\![M\sigma]\!]_\rho = f([\![M]\!]_{(x \mapsto [\![\sigma_{\mathrm{id}}(x)]\!]_\rho)_{x \in \Gamma'}})$,
   where f maps $raise_x\ v$ to $raise_{\sigma_{\mathrm{lab}}(x)}\ v$.

3.3.3 Operational semantics

We define a big-step "reduction" relation $M \Downarrow T$ between closed computations $\Delta;\ \cdot\ \vdash_{\mathrm{c}} M : A$ and (closed) terminals $\Delta;\ \cdot\ \vdash_{\mathrm{c}} T : A$ of the same type, such that for every such M either

1. $M \Downarrow T = \mathrm{return}\ V$, or
2. $M \Downarrow T = \mathrm{raise}_x\ V$, $x \in \mathrm{dom}\,\Delta$, or
3. M does not reduce.

Derivation rules are given in Figure 3.8, and the reduction relation is defined as their least fixed point.

Theorem 6 (adequacy)**.**

1. If $M \Downarrow \mathrm{return}\ V$, then $[\![M]\!]_\varnothing = \mathit{return}\ [\![V]\!]_\varnothing$.
2. If $M \Downarrow \mathrm{raise}_x\ V$, then $[\![M]\!]_\varnothing = \mathit{raise}_x\ [\![V]\!]_\varnothing$.
3. If M does not reduce, then $[\![M]\!]_\varnothing = \bot$.

3.3.4 Translation from sum-based iteration

Let $\Gamma \vdash_{\mathrm{c}} M : A$ or $\Gamma \vdash_{\mathrm{v}} V : A$ be a computation or value in the language with sum-based iteration. We define a *translation* $\mathrm{translate}(M)$, $\mathrm{translate}(V)$ from sum-based iteration, such that $\cdot\ ;\ \Gamma \vdash_{\mathrm{c}} \mathrm{translate}(M) : A$ or $\Gamma \vdash_{\mathrm{v}} \mathrm{translate}(V) : A$, respectively, in the language with labelled iteration. The translation macro-expands sum-based iter as follows. The other constructs are left unchanged.

$$\mathrm{translate}(\mathrm{iter}\ V, x.\ M) = \mathrm{iter}\ V, x.\ \big(\mathrm{translate}(M)\ \mathrm{to}\ \mathit{result}.\ \mathrm{case}\ \mathit{result}\ \mathrm{of}\ \{\mathrm{inl}\ y.\ \mathrm{raise}_x\ y; \mathrm{inr}\ x'.\ \mathrm{return}\ x'\}\big)$$

where $\mathrm{translate}(M)$ is implicitly weakened by adding x to Δ.

Note that in this translation, we only need to use labels directly under iter, and labels never propagate over longer distances.

Theorem 7 (translation preserves semantics)**.**

1. Let $\Gamma \vdash_{\mathrm{c}} M : A$ be a computation of the language with sum-based iteration, and $\rho \in [\![\Gamma]\!]$. Then $[\![M]\!]_\rho = [\![\mathrm{translate}(M)]\!]_\rho$.
2. Let $\Gamma \vdash_{\mathrm{v}} V : A$ be a value of the language with sum-based iteration, and $\rho \in [\![\Gamma]\!]$. Then $[\![V]\!]_\rho = [\![\mathrm{translate}(V)]\!]_\rho$.

Corollary 8. If $M \Downarrow T$ in the language with sum-based iteration, then there is T' such that $\mathrm{translate}(M) \Downarrow T'$ in the language with labelled iteration, and $[\![T]\!]_\varnothing = [\![T']\!]_\varnothing$. And if M does not reduce to a terminal, then $\mathrm{translate}(M)$ does not reduce to a terminal.

3.4 Discussion and related work

In our presentation of labelled iteration, we have chosen to only consider pure functions. In Chapter 5, we proceed to generalise this language to allow for effectful function types.

We have noticed the *De Bruijn index awkwardness* in settings other than iteration. For instance, it is customary in functional languages such as Haskell to use monad transformers [36] to embed imperative programs with multiple side-effects, but they suffer from a similar De Bruijn index awkwardness: the i^{th} monad transformer is addressed by writing "$\text{lift}^i\ \mathit{effect}$". This issue and proposed solutions have been studied in the literature [40, 55, 15], but addressing effects using labels seems yet unexplored. Imperative languages address mutable cells using variables, and it is possible that addressing effects with labels might benefit the readability of similar functional programs as well.

Many programming languages have not just unlabelled and labelled `continue`, after which we have modelled our combination of iter and raise, but also unlabelled and labelled `break`. It should be straightforward to introduce a construct that binds a label like iter, but when the label is raised with parameter a, the result of that construct is a, so that raise of that label resembles `break`. Such a construct, together with the raise we used in this chapter, resembles an intra-procedural form of exception handling. If we wrap an iter inside this new construct and use one label for breaking and one for continuing, we can "`break`" and "`continue`" from this combination of constructs, to deepen the resemblence with Java-style loops.

3.5 Chapter conclusion

In this chapter we summarized the essence of the sum-based representation of iteration, and evaluated it from a programming perspective. Although it might work well for a semantics standpoint, it is inadequate for programmers to program in. We proposed an alternative representation of iteration that is suitable for programmers, but still has relatively clean semantics.

3.6 Proofs

We first prove adequacy of fine-grain call-by-value without iteration. The adequacy of FGCBV + sum-based iteration and the adequacy of the language with labelled iteration are then minor modifications. All our adequacy proofs are in the style of Tait [70].

We use the following substitution lemma for both plain FGCBV and FGCBV with sum-based iteration.

Lemma 9. Assume a substitution $\sigma : \Gamma' \to \Gamma$ and an environment $\rho \in [\![\Gamma]\!]$.

1. Let $\Gamma' \vdash_{\mathrm{v}} V : A$ be a value. Then $[\![V]\!]_{(x \mapsto [\![\sigma_x]\!]_\rho)_{x \in \Gamma'}} = [\![V\sigma]\!]_\rho$.

2. Let $\Gamma' \vdash_{\mathrm{c}} M : A$ be a computation. Then $[\![M]\!]_{(x \mapsto [\![\sigma_x]\!]_\rho)_{x \in \Gamma'}} = [\![M\sigma]\!]_\rho$.

The proofs of both substitution lemmas, Lemma 9 and Lemma 5, are routine and we omit them.

3.6.1 Adequacy of FGCBV without iteration

We prove adequacy with the help of the following type-indexed predicate on closed terms.

Definition 10. By mutual induction on the type of V and M, respectively.

when $\vdash_{\mathrm{v}} V : 1$: $\quad P(V) \equiv \mathrm{true}$

when $\vdash_{\mathrm{v}} V : \mathrm{nat}$: $\quad P(V) \equiv \mathrm{true}$

when $\vdash_{\mathrm{v}} V : A + B$: $\quad P(\mathrm{inl}\ V) \equiv P(V)$

$\qquad\qquad\qquad\qquad\qquad\quad P(\mathrm{inr}\ V) \equiv P(V)$

when $\vdash_{\mathrm{v}} V : A \times B$: $\quad P(\langle V, W\rangle) \equiv P(V) \wedge P(W)$

when $\vdash_{\mathrm{v}} V : A \to B$: $\quad P(\lambda x.\ M) \equiv \forall(\vdash_{\mathrm{v}} W : A) : P(W) \Rightarrow P(M[W/x])$

when $\vdash_{\mathrm{c}} M : A$: $\quad P(M) \equiv \exists(\vdash_{\mathrm{v}} V : A) : \Big(P(V)\ \wedge\ M \Downarrow \mathrm{return}\ V\ \wedge\ [\![M]\!]_\varnothing = \mathrm{inl}\ [\![V]\!]_\varnothing\Big)$

Observe that $P(M)$ implies adequacy of M.

Proposition 11.

1. If $\Gamma \vdash_{\mathrm{v}} V : A$, and if for all $(x{:}B) \in \Gamma$ we have a closed $\vdash_{\mathrm{v}} \sigma_x : B$ satisfying $P(\sigma_x)$, then $P(V\sigma)$.

2. If $\Gamma \vdash_{\mathrm{c}} M : A$, and if for all $(x{:}B) \in \Gamma$ we have a closed $\vdash_{\mathrm{v}} \sigma_x : B$ satisfying $P(\sigma_x)$, then $P(M\sigma)$.

Proof. By induction on the term. Here are some interesting and less interesting cases.

$V = x$) Then $V\sigma = \sigma_x$, which was assumed to satisfy P.

$M = \mathrm{return}\ V$) Trivially by induction.

$V = \lambda y.\ M$) We have to show that if $\vdash_{\mathrm{v}} W : A$ satisfies P, then $M[\sigma, W/y]$ satisfies P. By induction.

$M = \mathrm{let}\ V\ \mathrm{be}\ x.\ N$) We are allowed to assume $P(V\sigma)$, so the induction hypothesis gives us $P(N[\sigma, (V\sigma)/x])$. We know that $M\sigma$ and $N[\sigma, (V\sigma)/x]$ reduce to the same terminal. We know $[\![M\sigma]\!]_\varnothing = [\![N\sigma]\!]_{x \mapsto [\![V\sigma]\!]_\varnothing}$, which we know is equal to $[\![N[\sigma, (V\sigma)/x]]\!]_\varnothing$ by the substitution lemma. Now $P(N[\sigma, (V\sigma)/x])$ implies $P(M\sigma)$.

$M = V\ W$) Similarly.

$M = M'\ \mathrm{to}\ x.\ N$) From the induction, we get V such that $P(V)$ and $M'\sigma \Downarrow \mathrm{return}\ V$ and $[\![M'\sigma]\!]_\varnothing = \mathrm{inl}\ [\![V]\!]_\varnothing$. From the derivation rule and the induction, we get V' such that $P(V')$ and $N[\sigma, V/x] \Downarrow \mathrm{return}\ V'$, and $[\![N[\sigma, V/x]]\!]_\varnothing = \mathrm{inl}\ [\![V']\!]_\varnothing$.

By the substitution lemma, $[\![N\sigma[V/x]]\!]_\varnothing = [\![N\sigma]\!]_{x \mapsto [\![V]\!]_\varnothing}$, and because we know $[\![M'\sigma]\!]_\varnothing = \mathrm{inl}\ [\![V]\!]_\varnothing$, we know that by definition

$$[\![(M'\sigma)\ \mathrm{to}\ x.\ (N\sigma)]\!]_\varnothing = [\![N\sigma]\!]_{x \mapsto [\![V]\!]_\varnothing}\ .$$

This completes the proof for this case.

$M = \mathrm{case}\ V\ \mathrm{of}\ \cdots$) Depending on the type of V, but for every type trivially by case analysis on $V\sigma$. □

Corollary 12. All closed terms satisfy P.

Adequacy directly follows from this.

Observe that the only cases in which we essentially looked at the normal form of $M\sigma$ are $\mathrm{return}\ V$ and $M\ \mathrm{to}\ x.\ N$. Specifically, we did not use the normal form of $M\sigma$ in the let case. This means that we can reuse most of the proof for FGCBV with sum-based iteration.

3.6.2 Adequacy of FGCBV + sum-based iteration

Similar structure. We redefine $P(M)$:

$$P(\vdash_{\mathrm{c}} M : A) = \Big(\exists(\vdash_{\mathrm{v}} V : A) : (P(V) \ \wedge\ M \Downarrow \mathrm{return}\ V \ \wedge\ [\![M]\!]_{\varnothing} = \mathrm{inl}\ [\![V]\!]_{\varnothing}) \\ \vee\ (M \not\Downarrow \ \wedge\ [\![M]\!]_{\varnothing} = \mathrm{inr}\ \bot)\Big)$$

We have the same proposition as Proposition 11 in this case:

- The case $M = \mathrm{return}\ V$ is still trivial.
- For $M = M'\ \mathrm{to}\ x.\ N$, we have to consider the alternative case that $M'\sigma \not\Downarrow$ and $[\![M'\sigma]\!]_{\varnothing} = \mathrm{inr}\ \bot$. This case is trivial.
- For iter, observe that every sequence $V_1, \cdots, V_k$ in the operational semantics corresponds uniquely to a sequence

  $$v_0 = [\![V\sigma]\!]_{\varnothing}, v_1 = [\![V_1]\!]_{\varnothing}, v_2 = [\![V_2]\!]_{\varnothing}, \cdots, v_k = [\![V_k]\!]_{\varnothing}$$

  for the denotational semantics, and the proof in that case is analogous to the proof for let.

  To prove that non-existence of a valid sequence $V_1, \cdots, V_k$ for the operational semantics implies the non-existence of a valid sequence $v_0, \cdots, v_k$, we instead prove the contrapositive. Indeed, we have our initial $V\sigma$ already, and by induction on a valid sequence $v_0, \cdots, v_k$ together with the induction hypothesis, we obtain step by step our sequence $V_1, \cdots, V_k$. So now we also know that $\mathrm{iter}\ V, x.M \not\Downarrow$ implies $[\![\mathrm{iter}\ V, x.M]\!]_{\varnothing} = \mathrm{inr}\ \bot$.

3.6.3 Adequacy of the language with labelled iteration

Similar structure. We redefine $P(M)$ again. Recall that M closed means that $\Delta;\ \cdot\ \vdash_{\mathrm{c}} M : A$.

$$P(M) \equiv \Big(\Big(\exists(\vdash_{\mathrm{v}} V : A) : \Big(P(V)\ \wedge\ M \Downarrow \mathrm{return}\ V\ \wedge\ [\![M]\!]_{\varnothing} = \mathit{return}\ [\![V]\!]_{\varnothing}\Big)\Big)$$
$$\vee \Big(\exists((x{:}B) \in \Delta) : \exists(\vdash_{\mathrm{v}} V : B) : \big(P(V)\ \wedge\ M \Downarrow \mathrm{raise}_x\ V\ \wedge\ [\![M]\!]_{\varnothing} = \mathit{raise}_x\ [\![V]\!]_{\varnothing}\big)\Big)$$
$$\vee \big(M \not\Downarrow\ \wedge\ [\![M]\!]_{\varnothing} = \bot\big)\ \Big)$$

We have a proposition analogous to Proposition 11.

Proposition 13.

1. If $\Gamma \vdash_{\mathrm{v}} V : A$, and if for all $(x{:}B) \in \Gamma$ we have a closed $\vdash_{\mathrm{v}} \sigma_x : B$ satisfying $P(\sigma_x)$, then $P(V\sigma)$.

2. If $\Delta; \Gamma \vdash_{\mathrm{c}} M : A$, and if we have a substitution $\sigma : \Delta'; \Gamma' \to \Delta;\ \cdot$ such that $P(\sigma_{\mathrm{id}}(x))$ on all variables, then $P(M\sigma)$.

- The additional case for sequencing is trivial.
- The case $P(\mathrm{raise}_x\ V)$ is trivial.
- The additional case for iteration is analogous.

Chapter 4

HETEROGENEOUS LOGICAL RELATIONS: A TECHNIQUE FOR THE COHERENCE OF DENOTATIONAL SEMANTICS

4.1 Introduction

This chapter is about the following problem:

> In many lambda calculi, the same typing judgement follows from multiple derivations. Yet the denotational semantics is defined by induction on a term's derivation. How do we know whether the semantics of a term is well-defined?

Non-unique derivations in lambda calculi are more common than one might think. It may surprise the reader to learn that it is hard to avoid non-unique derivations in a type system, unless every raw term can only be assigned one typing judgement, which is rather rare. Indeed, what is the type of the following term?

$$\lambda x.\, x$$

Is it a function from bool to bool? Or one from nat to nat? What is its context? And in type-and-effect systems, what is its effect set? These questions do not have easy answers in

"Curry-style" lambda calculi, where λ is not annotated with the type of its argument.

So we accept that terms may not have unique types. To find a term without a unique derivation, we readily construct a counterexample that uses let to bind a variable that then remains unused:

$$\mathrm{let}\ f = (\lambda x.\, x)\ \mathrm{in}\ 5 \quad : \quad \mathrm{nat}$$

Every type for f gives rise to a different typing derivation for this term.

The reader might ponder that to avoid the coherence problem, we merely need to annotate lambdas in order to regain unique derivations. But lambdas are not the only construct that would require such care. Also of concern are sum types: $\mathrm{inl}\ 3$ may have a variety of types, including $\mathrm{nat} + \mathrm{nat}$ and $\mathrm{nat} + \mathrm{string}$. One could imagine disambiguating by writing $\mathrm{inl}_{\mathrm{string}}\, 3$, but that is rather ugly. Then there are effect types: does the type of $(\lambda x.\, x)$ specify that it is a pure function, or is its type ambiguous about that?

We prefer our language to be slightly implicit over such verbosity, and we deal with the coherence problem.

Semantic coherence is clearly a rather universal problem for lambda calculi, and we are looking for a generically applicable solution. Many specific approaches are already discussed in the literature; here is a brief overview. Papers such as [17, 18] give a coherence result for coercions, for a language where lambdas are annotated with the type of the argument. Ohori [54] requires the language to be strongly normalising, which excludes Turing-complete languages. Reynolds [65] presents a solution that works only because the semantics of everything is an element of a universal domain, and we do not want to restrict ourselves to domain-theoretic semantics either. Bidirectional typechecking such as in [21] assigns a unique derivation to many terms, but still requires type annotations on reducible expressions which is less general than we would like. An inquiry by the author's PhD advisor (Paul Blain Levy) to the TYPES mailing list on 3 February 2017 showed — to the author's mild sorrow — that the community at large is only aware of such specific approaches.

The author did find relief encountering heterogeneous logical relations in Biernacki et al. [13]. This is a very versatile method which works both in sophisticated settings such as [13], and also in the simple setting of lambda calculus, as we will now show.

Remark. We use the following words with these meanings.

- *Typing derivation*: a specific application of the typing rules leading to a typing judgement. Typing derivations necessarily have a single statement at the bottom, which we call the *conclusion*. The typing rules are specific to the formal system that we are considering in a chapter.
- *Typing judgement*: the conclusion of any (valid) typing derivation, without regard for the derivation that generated it.
- *Raw term*: the abstract syntax tree of a term, without regard for whether or not the term can typecheck. We write metavariables for raw terms in `typewriter font`. We leave the tree structure implicit, and write just the syntax.
- *Typed term*, or simply *term*: a typing judgement, but in the presentation we leave everything except the raw term implicit for the reader. The complete typing judgement may be ambiguous or not, but if ambiguous then it is not important to the story.

In this chapter we use plain lambda calculus, so there will be no distinction between values and computations.

4.2 A lambda calculus with finite sums and products and subtyping

To present the heterogenous logical relations technique succinctly, we switch to the following plain lambda calculus, with some number of abstract ground types (denoted G), countable sum and product types, and a subtyping (see on page 81). The $01{+}{\times}$ types with the usual

constructors and pattern matching are subsumed by this presentation of countable product and sum types. We leave out a concrete syntax for the elements of ground types, representing it by ellipses. The pattern-matches are analogous to the sum and product pattern matches. We present the language as if all countable sum and product types are present, but nothing in this chapter depends on the existence of any sum or product types.

$$\begin{array}{lll} \text{types} & A, B & ::= G \mid A \to B \mid \sum_{i \in I} A_i \mid \prod_{i \in I} A_i \\ \text{terms} & M, N & ::= x \mid \text{let } M \text{ be } x.\, N \mid \lambda x.\, M \mid M\, N \mid \text{in}_i\ M \mid \left\langle \vec{M} \right\rangle \\ & & \quad \mid \text{case } M \text{ of } \{\text{in}_i\ x.\, N_i\}_{i \in I} \mid \text{case } M \text{ of } \langle \vec{x} \rangle\, .\, N \mid \cdots \end{array}$$

The semantics of types

This language has no effects, and indeed we choose a very simple denotational semantics of types:

$$\begin{aligned} [\![G]\!] &= \cdots \\ [\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \\ [\![\sum_{i \in I} A_i]\!] &= \sum_{i \in I} [\![A_i]\!] \\ [\![\prod_{i \in I} A_i]\!] &= \prod_{i \in I} [\![A_i]\!] \end{aligned}$$

The semantics of derivations

Notation. Suppose that we have some typing judgement, say $\Gamma \vdash M : A$, for some term M. We write $\boldsymbol{D :: \Gamma \vdash M : A}$ to express that D is a typing derivation with conclusion $\Gamma \vdash M : A$.

We use a very plain set-theoretic semantics. When $D :: \Gamma \vdash M : A$ then $[\![D]\!] \in \left(\prod_{x \in |\Gamma|} [\![\Gamma(x)]\!] \right) \to [\![A]\!]$. The semantics of *derivations* is the obvious one. The semantics of variable derivations is projection. The semantics of let derivations is found by extending the environment tuple. The semantics of abstraction, application, injection, tuple-forming, and pattern matching is simply abstraction, application, injection, tuple-forming, and pattern matching on the set theory level.

We want to define the semantics of a *term* as the semantics of one of its derivations; for that we need to check that all derivations have the same semantics. In this chapter, we call the resulting kind of thing a "semantic value" and indicate it with metavariables v, w.

Subtyping, coercion, and its semantics

As explained in §4.1, this language already has typed terms with multiple derivations. We introduce even more ambiguity with a subtyping on the sum, product, and function types as follows. (1) A sum type is a subtype of another sum type when the latter has all variants of the former (and maybe more), and the types match. (2) A product type is a subtype of another product type when the former one has all fields that the latter has (and maybe more), and the types match. (3) Sum and product types are covariant in its constituent types, and function types are contravariant on the left and covariant on the right.

$$\frac{}{G \leq G} \qquad \frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}$$

$$\frac{I \subseteq J \qquad \forall i \in I : A_i \leq B_i}{\sum_{i \in I} A_i \leq \sum_{j \in J} B_j} \qquad \frac{I \subseteq J \qquad \forall i \in I : A_i \leq B_i}{\prod_{j \in J} A_j \leq \prod_{i \in I} B_i}$$

Proposition 14. Given two types A, B, there is at most one derivation of $A \leq B$. The subtyping relation $\leq$ is a partial order.

Definition 15. We give a semantics to the subtyping relation. When $A \leq B$ then $[\![A \leq B]\!]$ is a function from $[\![A]\!]$ to $[\![B]\!]$, defined as follows:

$$\begin{aligned}
[\![G \leq G]\!](v) &= v \\
[\![A \to B \leq A' \to B']\!](f) &= [\![B \leq B']\!] \circ f \circ [\![A' \leq A]\!] \\
[\![\sum_{i \in I} A_i \leq \sum_{j \in J} B_j]\!](in_i\ v) &= in_i\ [\![A_i \leq B_i]\!](v) \\
[\![\prod_{j \in J} A_j \leq \prod_{i \in I} B_i]\!](\langle v_j \rangle_{j \in J}) &= \langle [\![A_i \leq B_i]\!](v_i) \rangle_{i \in I}
\end{aligned}$$

Proposition 16. $[\![-]\!]$ is a functor from the poset of types and subtypes to Set, that is: $[\![A \leq A]\!]$ is the identity, and the composition of $[\![A \leq B]\!]$ followed by $[\![B \leq C]\!]$ is always the same as $[\![A \leq C]\!]$.

When we have a typed term and that type has a supertype, then we can coerce:

$$\frac{A \leq B \qquad \Gamma \vdash M : A}{\Gamma \vdash M : B} \tag{4.1}$$

The semantics of a coerced term is found by postcomposition with $[\![A \leq B]\!]$. To keep the presentation uncomplicated, we do not have a rule for weakening or coercion in the context.

4.3 The logical relation

In this section, we define a relation $R[A; B]$ for every pair of types A, B,

$$R[A; B] \quad \subseteq \quad [\![A]\!] \times [\![B]\!] \quad .$$

The intuition is as follows. In order to define our logical relation, we make explicit that we have four kinds of types — functions, products, sums, and ground types — and our subtyping only relates types of the same kind: a function type can be a subtype of another but not a subtype of a product type. On the semantic side we have the same situation: we can only ever coerce (with $[\![A \leq B]\!]$) a function into another function, and never into a tuple. However, we may coerce a tuple into a smaller tuple, or coerce each element of a tuple.

Our logical relation will relate all pairs of semantic values that "do not have conflicting information":

- The source of conflict comes from values like $inl\ \ v$ versus $inr\ \ w$, which are never related. In general, *sum-injections* $in_i\ v, in_j\ w$ are never related if $i{\neq}j$. And $in_i\ v$ and $in_i\ w$ are related iff v and w are related.
- Two *tuples* are related precisely if all the common fields are related. Two tuples with no fields in common are always related, as there is no conflict at any field.
- *Functions* are related if all related values map to related values.

- We take the view that *different kinds of types* do not conflict: any sum-injection is related with any tuple, and any sum-injection or tuple is related with any function, and vice versa. The syntax as is does not allow any overlap between the different kinds of types, but we do not rule it out either for extended languages.

- If we have ground values, then they are related with themselves and to all non-ground values, but not to any other ground value.

We can summarily describe our logical relation as follows: **two semantic values are related if and only if all sum-injections in the common structure on both sides have the same indices**. An important consequence of this is that for a given type A, relation $R[A;A] \subseteq [\![A]\!]^2$ is the diagonal, relating values to themselves and to nothing else, as we will see below.

We define the relations R as follows:

$$
\begin{array}{rrrll}
v\, R\big[G;G\big]\, w & & \text{iff} \quad v = w & \text{for the same ground type } G \\
\neg\,(v\, R\big[G;G'\big]\, w) & & & \text{for two different ground types } G, G' \\
f\, R\big[A \to B; A' \to B'\big]\, g & & & \text{iff} \quad \forall (x\, R[A;A']\, x') : f x\, R[B;B']\, g x' \\
in_i\, v & R\big[\textstyle\sum_{i\in I} A_i\,;\sum_{i\in J} B_i\big] & in_i\, w & \text{iff} \quad i \in I \cap J \ \text{ and } \ v\, R[A_i\,;B_i]\, w \\
\neg\,(in_i\, v & R\big[\textstyle\sum_{i\in I} A_i\,;\sum_{i\in J} B_i\big] & in_j\, w\,) & \text{if} \quad i \neq j \\
\langle v_i\rangle_{i\in I} & R\big[\textstyle\prod_{i\in I} A_i\,;\prod_{i\in J} B_i\big] & \langle w_i\rangle_{i\in J} & \text{iff} \quad \forall i{\in}(I\cap J) :\ v\, R[A_i\,;B_i]\, w
\end{array}
$$

And relation R between all pairs of types that are of a different kind (ground, function, sum, product) is the "everything relation".

We also define R on environments, as long as the corresponding contexts have the same set of variables, say X:

$$
\begin{array}{rcl}
R[\Gamma;\Gamma'] & \subseteq & \textstyle\prod_{x\in X} [\![\Gamma(x)]\!] \times \prod_{x\in X} [\![\Gamma'(x)]\!] \\
\rho\, R[\Gamma;\Gamma']\, \rho' & \text{iff} & \forall x{\in}X :\ \rho(x)\, R[\Gamma(x)\,;\Gamma'(x)]\, \rho'(x)
\end{array}
$$

Furthermore, we define R on functions from environments to semantic values, as long as the again the contexts are on the same set of variables X:

$$\begin{array}{rcl} R[\Gamma \vdash A; \Gamma' \vdash A'] & \subseteq & \left(\left(\prod_{x \in X} [\![\Gamma(x)]\!]\right) \to [\![A]\!]\right) \times \left(\left(\prod_{x \in X} [\![\Gamma'(x)]\!]\right) \to [\![A']\!]\right) \\ f\ R[\Gamma \vdash A; \Gamma' \vdash A']\ g & \text{iff} & \forall \rho\, R[\Gamma; \Gamma']\, \rho' :\ f(\rho)\, R[A; A']\, g(\rho') \end{array}$$

Proposition 17.

1. Let A be a type. Then $R[A; A]$ is the diagonal.
2. Let Γ be a context. Then $R[\Gamma; \Gamma]$ is the diagonal.
3. Let A be a type and let Γ be a context. Then $R[\Gamma \vdash A; \Gamma \vdash A]$ is the diagonal.

Proof. For (1), by induction on A. Parts (2) and (3) are then trivial. □

We now wish to prove that R relates every pair of derivations of the same raw term. Below in Lemma 24, we will apply induction on the sum of the sizes of the pair of derivations. Observe that the structure of all derivations of any raw term is the same, modulo applications of coercion and the types mentioned in the judgements.

We prove some lemmas that relate R to the gadgets that we used in our semantics. The proofs of Lemmas 18,19,20,21 are all trivial.

Lemma 18 (coherence of λ). Let $\mathtt{M}$ be a raw term, and let

$$D :: \Gamma,\ x{:}A \vdash \mathtt{M} : B$$
$$D' :: \Gamma', x{:}A' \vdash \mathtt{M} : B'$$

for two contexts with the same set of variables $|\Gamma| = |\Gamma'|$. Suppose furthermore that

$$[\![D]\!]\ \ R\big[(\Gamma, x{:}A) \vdash B\,;\,(\Gamma', x{:}A') \vdash B'\big]\ \ [\![D']\!] \quad .$$

Write $\lambda x.\, D$ (resp. $\lambda x.\, D'$) for the derivation obtained by adding an abstraction rule at the bottom. Then

$$[\![\lambda x.\, D]\!]\ \ R\big[\Gamma \vdash (A \to B)\,;\,\Gamma' \vdash (A' \to B')\big]\ \ [\![\lambda x.\, D']\!] \quad .$$

For brevity, we leave the indices to R implicit in the future.

Lemma 19 (coherence of let). Let $\mathtt{M}, \mathtt{N}$ be two raw terms, and let

$$\begin{aligned} D &:: \Gamma &&\vdash \mathtt{M} : A \\ D' &:: \Gamma' &&\vdash \mathtt{M} : A' \\ E &:: \Gamma,\ x{:}A &&\vdash \mathtt{N} : B \\ E' &:: \Gamma', x{:}A' &&\vdash \mathtt{N} : B' \end{aligned}$$

such that $|\Gamma| = |\Gamma'|$. Suppose furthermore that $[\![D]\!]\, R\, [\![D']\!]$ and $[\![E]\!]\, R\, [\![E']\!]$. Write $\mathrm{let}\ D\ \mathrm{be}\ x.\, E$ (resp. $\mathrm{let}\ D'\ \mathrm{be}\ x.\, E$) for the derivation that combines the two. Then we have $[\![\mathrm{let}\ D\ \mathrm{be}\ x.\, E]\!]\ R\ [\![\mathrm{let}\ D'\ \mathrm{be}\ x.\, E']\!]$.

Lemma 20 (coherence of in_i). Let D, D' be two derivations of $\mathtt{M}$ using contexts over the same set of variables. Let E, E' be two derivations that are obtained by applying in_i below D (resp. D'). That is, we use the same index i on both sides, but we require nothing about the resulting type of $in_i\ \mathtt{M}$. If $[\![D]\!]\, R\, [\![D']\!]$, then we have $[\![E]\!]\, R\, [\![E']\!]$.

Lemma 21 (coherence of tuple-forming). Let I be an index set. Let $<\mathtt{M}_i>_{i\in I}$ be a raw tuple expression with two derivations D, D' using contexts over the same set of variables. Write $(D_i)_{i\in I}, (D'_i)_{i\in I}$ for the derivations of the parts. If $\forall i:\ [\![D_i]\!]\, R\, [\![D'_i]\!]$, then we have $[\![D]\!]\, R\, [\![D']\!]$.

The lemmas for application and for pattern-matching on sums and products are analogous and also trivial.

For coercion, we first need the following lemma. (Recall that we defined $[\![A \le B]\!]$ in Def. 15.)

Lemma 22 (R preserves semantic coercion). Let A, B, C be three types with $B \le C$, and let

$$\begin{aligned} v &\in [\![A]\!] \\ \text{and} \qquad w &\in [\![B]\!] \\ \text{so that} \quad [\![B \le C]\!](w) &\in [\![C]\!] \quad . \end{aligned}$$

1. If vRw, then $v\ R\ [\![B{\le}C]\!](w)$.
2. If wRv, then $[\![B{\le}C]\!](w)\ R\ v$.

Proof. From the definition of $R[-\,;-]$ it is clear that for types X,Y, relation $R[X;Y]$ is the transpose of $R[Y;X]$, so the two cases are symmetric. We prove only the first part.

By induction on the proof of $B \leq C$.

- Consider the case $G \leq G$. Then $[\![G \leq G]\!]$ is the identity so this is trivial.
- Consider the case $B_1 \to B_2 \leq C_1 \to C_2$, implying that $C_1 \leq B_1$ and $B_2 \leq C_2$. Then A is either a function type or it is not, but if it is not then already trivially $v\ R[A; C_1 \to C_2]\ [\![B \leq C]\!] w$. So let us assume that A is some function type $A_1 \to A_2$.

  We know from Def. 15 that $[\![B \leq C]\!](w) = [\![B_2 \leq C_2]\!] \circ w \circ [\![C_1 \leq B_1]\!]$. We also know from assumption vRw that

  $$\forall (x\ R[A_1; B_1]\ x') : \quad vx\ R\Big[A_2; B_2\Big]\ wx'$$

  and we wish to prove that

  $$\forall (x\ R[A_1; C_1]\ x') : \quad vx\ R\Big[A_2; C_2\Big]\ [\![B_2 \leq C_2]\!]\Big(w\ [\![C_1 \leq B_1]\!](x')\Big) \quad .$$

  So assume such $x\ R[A_1; C_1]\ x'$. We know that $C_1 \leq B_1$, so by induction:

  $$x\ R\Big[A_1; B_1\Big]\ [\![C_1 \leq B_1]\!](x')$$

  By assumption, we then know

  $$vx\ R\Big[A_2; B_2\Big]\ \big(w\ [\![C_1 \leq B_1]\!](x')\big)$$

  We also know that $B_2 \leq C_2$, so by induction we get

  $$vx\ R\Big[A_2; C_2\Big]\ [\![B_2 \leq C_2]\!]\Big(w\ [\![C_1 \leq B_1]\!](x')\Big)$$

  as required.

- Consider the case $\sum_{j \in J} B_j \leq \sum_{k \in K} C_k$, implying that $J \subseteq K$ and each B_j is below the corresponding C_j. Then either A is a sum type or it isn't, but if it isn't then already trivially $v\ R[A; \sum_{j \in J} B_j]\ [\![B \leq C]\!] w$. So let us assume that A is some sum type $\sum_{i \in I} A_i$.

Then by vRw there must also be some i, v', w',

$$\begin{aligned} i &\in I \cap J \\ v &= in_i\ v' \\ w &= in_i\ w' \end{aligned}$$

such that $v'\ R[A_i; B_i]\ w'$. By induction, we also know that $v'\ R[A_i; C_i]\ [\![B_i \leq C_i]\!](w')$. Therefore, as required,

$$in_i\ v' \qquad R\big[\textstyle\sum \vec{A};\ \sum \vec{C}\big] \qquad in_i\ [\![B_i \leq C_i]\!](w') \quad .$$

- Consider the case $\prod_{j\in J} B_j \leq \prod_{k\in K} C_k$, implying that $J \supseteq K$ and each C_k is above the corresponding B_k. Then either A is a product type or it isn't, but if it isn't then already trivially $v\ R[A; \prod_{j\in J} B_j]\ [\![B \leq C]\!]w$. So let us assume that A is some product type $\prod_{i\in I} A_i$. Then by vRw, both v and w must be tuples and $\forall (i \in I \cap J) :\ v_i\ R[A_i; B_i]\ w_i$. Then certainly we must have $\forall (i \in I \cap K) :\ v_i\ R[A_i; B_i]\ w_i$. We use induction on each element of the restricted $\vec{w}$ to find that

  $$\forall (i \in I \cap K) :\ v_i\ R[A_i; C_i]\ [\![B_i \leq C_i]\!](w_i)$$

  and thus $(v_i)_{i\in I}\ \ R\big[\sum_{i\in I} A_i;\ \sum_{k\in K} C_k\big]\ \ \big([\![B_k \leq C_k]\!](w_k)\big)_{k\in K}$ as required. □

We can now give the lemma for coercion. As coercion does not show up in raw term syntax, let us write $(A \leq B)\ D$ for a derivation D extended with a coercion from A to B at the bottom.

Lemma 23 (coherence of coercion). Let $\mathtt{M}$ be a raw term, and let

$$\begin{aligned} D &:: \Gamma \vdash \mathtt{M} : A \\ D' &:: \Gamma' \vdash \mathtt{M} : A' \end{aligned}$$

such that $|\Gamma| = |\Gamma'|$, and let $[\![D]\!]\ R\ [\![D']\!]$.

1. If $A \leq B$, then $[\![(A \leq B)\ D]\!]\ R\ [\![D']\!]$.
2. If $A' \leq B'$, then $[\![D]\!]\ R\ [\![(A' \leq B')\ D']\!]$.

Proof. Recall from page 82 that we defined $[\![(A \leq B)\ D]\!] = [\![A \leq B]\!] \circ [\![D]\!]$. Apply Lemma 22. □

Lemma 24. Let D and D' be two derivations of the same judgement. Then $[\![D]\!]\ R\ [\![D']\!]$.

Proof. By induction on the sum of the sizes of D and D'. If one of the derivations ends in a coercion then apply Lemma 23. Otherwise, both derivations must end in applications of the same syntax-generating rule, and we can apply the appropriate lemma for this rule. □

Theorem 25. Let D and D' be two derivations of the same judgement. Then $[\![D]\!] = [\![D']\!]$.

Proof. We know from Lemma 24 that $[\![D]\!]\ R[\Gamma \vdash A; \Gamma \vdash A]\ [\![D']\!]$, and in Proposition 17 we saw that $R[\Gamma \vdash A; \Gamma \vdash A]$ is the diagonal. □

4.4 Conclusion

Coherence of a denotational semantics is essential if we define a semantics by induction on a typing derivation. We can use a heterogeneous logical relation to prove this with relative ease for a relatively simple lambda calculus with sums, products, subtyping, and a set-based semantics. The task consists of the following steps.

- We divide up types into a number of classes (functions, tuples, sum-injections, ground values).
- For any two types A, B we define the relation $R[A; B]$: for any two types within the same class, we related semantic values that are "free of conflict all the way down within the common structure", where conflicts are only caused by two values that are both elements of a sum type, but which are different branches. (Recall page 83 for the precise definition.) If A and B are of different kinds, then $R[A; B]$ is the everything relation, as there is no common structure where there can be a conflict.

  On the same type, $R[A; A]$ is the diagonal.

- We show that the relation is "preserved by semantic coercion": if we have two related semantic values vRw, then the relation still holds after applying any coercion to v or w. (Lemma 22)

- We define a relation on semantics of derivations $[\![D]\!]$. Assume that two contexts Γ and Γ' have the same variable names; we relate two semantic derivations ($R[\Gamma \vdash A; \Gamma' \vdash A']$) when for pointwise related environments, the semantic derivation returns related semantic values.

  Again, $R[\Gamma \vdash A; \Gamma \vdash A]$ is the diagonal.

- For two derivations whose semantics are related, the relation still holds after applying coercion on the bottom of one derivation. (Lemma 23)

- If two derivations of the same raw term end in the same syntax-generating rule, then the semantics of the two derivations are related if the semantics of the premise derivations of each of the branches are related. (Lemmas 18, 19, 20, 21, etc.)

- Given a raw term, there is only a single syntax-generating rule that can be at the bottom of its derivations. (That is, we do not count coercion rules, which do not show up in term syntax.)

- We deduce by induction that any two derivations of the same raw term have related semantics (Lemma 24). And all derivations of the same *judgement* have *equal* semantics (Theorem 25).

As discussed in §4.1, the approach for semantic coherence presented here avoids many particularities of the language/semantics in question. In particular, this approach does not require much at all from the type system — in particular it does not require minimal types — nor does it seem very important that the denotational semantics is set-based: the technique seems to work for any semantics based on a concrete cartesian closed category.

Additional simplicity comes from our simple notion of coherence — merely equality as elements of sets — in contrast to Biernacki et al. [13] which uses a CPS translation semantics,

and where a notion of coherence defined indirectly in terms of operational semantics is more appropriate.

The approach presented here is fairly extensible. For instance, we can add top and bottom types (with singleton and empty semantics, respectively). The singleton element $*$ of $[\![\top]\!]$ would be related to itself and to everything else. It remains to show — analogously to Lemma 22 — that if vRw then $*Rw$, which is trivial, and if the nonexistent element of $[\![\bot]\!]$ is related to w then anything must be related to w, which is also trivial.

In Lemma 22, we assumed that our ground types do not have a subtyping between the ground types. Indeed, a more interesting subtyping relation and coercion may easily cause the semantics to be incoherent. The crucial bits to look at in our approach are the logical relation R, and more work needs to be done specifically for the coercion in Lemma 22. If there are any new syntactic constructs that work at multiple types, then lemmas like Lemma 18 need to be proven.

We feel that the brevity of the current approach is due in part to not having type variables and substitution. It would be interesting to see whether this can be added to the current approach.

Chapter 5

EFFECTFUL FUNCTION TYPES FOR LABELLED ITERATION

5.1 Introduction

We extend the labelled iteration language of Chapter 3 with a type-and-effect system and effectful functions. We have to take some care to make sure that the denotational semantics is well-defined. In Chapter 9 on page 208, we sketch how this chapter might form a blueprint for a more general notion of defined algebraic operations.

In Chapter 3, we described a form of non-global "exceptions" which are bound rather than handled, so that unhandled exceptions are not only prevented but ungrammatical. These exceptions occur naturally in Java-like languages: they are bound by `label: while (...) {...}` or `label: for (...;...;...) {...}`, are raised by `break label;` or `continue label;`, and are considered up to alpha renaming of labels. It is a syntax error to use a label that has not been bound, in the same way that accessing undefined variables is a syntax error. Using an undefined variable or undefined label is not a runtime error but simply *ungrammatical*.[1]

[1] As in Chapter 3, we only consider `continue` using iter and raise. A `break` statement would be completely

Inputs: $x_{11}, \cdots, x_{33} : \mathrm{nat}$

// We use iteration as a "trick": as a way to easily break out of code, like an early return.

$\mathrm{iter}\ 0, \mathit{result}.$

$\quad \mathrm{if}\ \ \mathit{result} \neq 0$

$\quad \mathrm{then}\quad$ // Nonzero iteration variable means: the result has been found.

$\qquad \mathrm{return}\ \mathit{result}$

$\quad \mathrm{else}\quad$ // Define a helper function that may invoke the label,

$\qquad \mathrm{let}\ \mathit{check3} = \Big(\ \lambda(\mathit{triple} : \mathrm{nat} \times (\mathrm{nat} \times \mathrm{nat})).$

$\qquad\quad \mathrm{case}\ \mathit{triple}\ \mathrm{of}\ (a, \mathit{pair}).\ \mathrm{case}\ \mathit{pair}\ \mathrm{of}\ (b, c).$

$\qquad\quad \mathrm{if}\qquad a = 1 \text{ and } b = 1 \text{ and } c = 1 \quad \mathrm{then}\,\mathrm{raise}_{\mathit{result}}\ 1$

$\qquad\quad \mathrm{else\,if}\quad a = 2 \text{ and } b = 2 \text{ and } c = 2 \quad \mathrm{then}\,\mathrm{raise}_{\mathit{result}}\ 2$

$\qquad\quad \mathrm{else}\,\mathrm{return}\ \langle\rangle\ \Big) \quad : \quad (\mathrm{nat} \times (\mathrm{nat} \times \mathrm{nat})) \to 1\,!\,\{\mathit{result}\}$

$\qquad \mathrm{in}\quad$ // and try every triple with it.

$\qquad\quad \mathit{check3}\ \langle x_{11}, \langle x_{12}, x_{13}\rangle\rangle\ \mathrm{to}\ \_.\ \mathit{check3}\ \langle x_{21}, \langle x_{22}, x_{23}\rangle\rangle\ \mathrm{to}\ \_.\ \mathit{check3}\ \langle x_{31}, \langle x_{32}, x_{33}\rangle\rangle\ \mathrm{to}\ \_.$

$\qquad\quad \mathit{check3}\ \langle x_{11}, \langle x_{21}, x_{31}\rangle\rangle\ \mathrm{to}\ \_.\ \mathit{check3}\ \langle x_{12}, \langle x_{22}, x_{32}\rangle\rangle\ \mathrm{to}\ \_.\ \mathit{check3}\ \langle x_{13}, \langle x_{23}, x_{33}\rangle\rangle\ \mathrm{to}\ \_.$

$\qquad\quad \mathit{check3}\ \langle x_{11}, \langle x_{22}, x_{33}\rangle\rangle\ \mathrm{to}\ \_.\ \mathit{check3}\ \langle x_{13}, \langle x_{22}, x_{31}\rangle\rangle\ \mathrm{to}\ \_.$

$\qquad\quad \mathrm{raise}_{\mathit{result}}\ 3$

Figure 5.1: A program of type $\mathrm{nat}\,!\,\varnothing$ that determines whether a game of noughts-and-crosses (tic-tac-toe) has been determined. This code does not type check with the language of §3.3 — because *check3* may invoke the label — but it does type check with the language in this chapter. The input is given in 9 numbers $x_{11}, \cdots, x_{33}$, which must be 1 for a nought, 2 for a cross, and 0 for an empty space. The program returns 1, 2, or 3. It returns 1 only if there are three noughts in a row; 2 only if there are three crosses in a row; 3 iff the game is undetermined.

We use iteration to simulate early return, but note that `return` in conventional languages would not work for this purpose here. We assume the existence of a boolean type, if..then..else, and suitable equality and conjunction operators. Similar code can also be written using pattern matching. We write _ for an unused variable.

The language in §3.3 had a significant limitation: it was not possible to invoke a label inside a function, when that label was bound outside the function. We fix that here, and allow label invocation across function boundaries as long as the function is not returned out of the scope of the labels it uses. In Figure 5.1 on the preceding page, we show an example that did not type check with the language of §3.3, but it does type check with the language in this chapter.

Recap

The typing judgements in Chapter 3 were:

$$\Gamma \vdash_{\mathrm{v}} V : A$$

$$\Delta; \Gamma \vdash_{\mathrm{c}} M : A$$

So values have a value context Γ available with typed variables; computations additionally have a label context Δ of labels that can be invoked, together with the type of value needed to invoke that label. A computation judgement in Chapter 3 may be, for instance,

$$\underline{x} : \mathrm{nat}\ ;\ y : \mathrm{nat} \vdash_{\mathrm{c}} \mathrm{raise}_{\underline{x}}\ y\ :\ \mathrm{nat}$$

This computation uses the value of y from the value context, and invokes label $\underline{x}$ with it.[2] So here,

$$\text{label context}\quad \Delta = \{\underline{x} : \mathrm{nat}\}$$

$$\text{value context}\quad \Gamma = \{y : \mathrm{nat}\}\ .$$

Recall from Chapter 3 that in practice, we tend to use the same letter for both a label and a variable, as this makes intuitive sense if the label models `continue` behaviour for a loop, spelled iter in our languages. Recall from §3.3 that there is no essential relationship between Δ and Γ.

---

analogous.

[2] In Chapter 3, labels were not underlined. We underline labels in this chapter to make the difference with variables more pronounced.

As pointed out at the end of §3.3.1 on page 65, the function types had to be pure, because the typing rule for λ cannot "see" the labels available to the computation that uses the value containing the λ:

$$\frac{\cdot\,;\ \Gamma, x{:}A \vdash_{\mathrm{c}} \quad M \quad : B}{\Gamma \vdash_{\mathrm{v}} \quad \lambda x.\ M \quad : A \to B}$$

Regardless of typing judgements, it is unclear how in Chapter 3 we could define the semantics of the function type $[\![A \to B]\!]$ appropriately, other than the set of functions that are pure apart from potential divergence.

The new type system

Here's how we fix this, and make programs like the one in Figure 5.1 legal. We keep the value context Γ. We also keep the label context Δ, but now it merely signifies the labels that we are *aware of* in the context; the presence of a label in Δ does not mean that we are allowed to use it. Indeed, our new value judgement shows that value are aware of the label bindings surrounding it. Of course, values are still inert and do not themselves invoke labels. The value judgement is:

$$\boxed{\Delta; \Gamma \vdash_{\mathrm{v}} V : A}$$

Computations are also aware of Δ. We also include a Gifford-style effect system [50]: our computation judgement states that $\theta \subseteq \mathrm{dom}(\Delta)$ (the "effect set") is an upper bound for the labels that the computation may invoke. The computation judgement is as follows; we use θ to be able to give a compositional semantics.

$$\boxed{\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta}$$

For types, we move to a notion of types *over a set of labels*. The types over $\mathrm{dom}(\Delta)$ are as follows. The effect set θ shows up in the function type, which is now an effectful function type $A \to B \,!\, \theta$. To make our language definition more modular, we call the "$B \,!\, \theta$" part a "computation type". Note that this language is still fine-grain call-by-value, not call-by-push-

value [48], as our function types are value types not computation types.

$$\begin{array}{lll} \text{value types} & A, B & ::= \; 1 \mid \mathrm{nat} \mid A + B \mid A \times B \mid A \to \underline{B} \\ \text{computation types} & \underline{A}, \underline{B} & ::= \; A \,!\, \theta \qquad (\theta \subseteq \mathrm{dom}(\Delta)) \end{array}$$

The label context Δ maps labels to types over $\mathrm{dom}(\Delta)$, and the value context Γ maps variables to types over $\mathrm{dom}(\Delta)$.

Let us move to the semantics. Recall that in Chapter 3, the semantics of a computation of type A in an environment was either a semantic value of type A, or a label together with a value of the corresponding type, or divergence ($\bot$). We correspondingly give semantics to the function type $A \to B \,!\, \theta$ here as follows. Note that the semantics of a type now depends on Δ, and so we write $[\![\Delta \vdash A]\!]$ (resp. $[\![\Delta \vdash \underline{A}]\!]$) for the semantics of a value (resp. computation) type.

$$\begin{aligned} [\![\Delta \vdash A \to B \,!\, \theta]\!] &= [\![\Delta \vdash A]\!] \to [\![\Delta \vdash B \,!\, \theta]\!] \\ [\![\Delta \vdash \quad B \,!\, \theta]\!] &= [\![\Delta \vdash B]\!] + \sum_{\underline{x} \in \theta} [\![\Delta \vdash \Delta(\underline{x})]\!] + \{\bot\} \end{aligned}$$

We arrive at the main complication in this chapter: Unfortunately, we cannot simply use this as an inductive definition, because we don't know that $\Delta(\underline{x})$ is a smaller type than $A \to B \,!\, \theta$. In turn, type $\Delta(\underline{x})$ may mention $\underline{y}$, and $\Delta(\underline{y})$ may be a larger type even. The solution we present in this chapter is to postulate that Δ is "acyclic", which makes sure that the semantics of types is well-defined.

Overview of this chapter. We extend the language from Chapter 3 to the language we just sketched with effectful function types. We give a denotational semantics, and use the method of Chapter 4 to prove it coherent. This implies coherence of the labelled iteration language from §3.3. We do not give an operational semantics; it is completely routine.

5.2 Type syntax and label contexts

Definition 26 (types, labels)**.** Let L be a finite set of "labels". **Types over** L are given inductively by the following grammar. We use A, B to denote a value type, and $\underline{A}, \underline{B}$ to denote a

computation type.

$$\begin{array}{lll} \text{value types} & A, B & ::= 1 \mid \mathrm{nat} \mid A + B \mid A \times B \mid A \to \underline{B} \\ \text{computation types} & \underline{A}, \underline{B} & ::= A \,!\, \theta \qquad (\theta \subseteq L) \end{array}$$

The computation type $A \,!\, \theta$ represents computations that may either return a value of type A, or invoke one of the exception labels in θ. We will later associate a type to each label, and exception invocations must be accompanied by a value of the respective type.

We will write labels as $\underline{x}, \underline{y}, \underline{z}$. In this chapter, L is always a finite set.

Proposition 27. Any type over L is also a type over a superset of L. In fact, any type over L is a type over any superset of the set of labels mentioned in that type.

Definition 28. A **label precontext** Δ on a finite set L consists of for each label $\underline{x} \in L$ a type $\Delta(\underline{x})$ over L. So $\mathrm{dom}(\Delta) = L$.

Acyclicity

Definition 29. Let Δ be a label precontext on a finite set L.

1. Let $\underline{x} \in L$ be a label. The **support of $\underline{x}$** with respect to Δ, notation $\mathbf{support}_{\Delta}(\underline{x})$, is the downset of $\underline{x}$ under the transitive closure of $\prec_{\Delta}$, defined as

$$\underline{y} \prec_{\Delta} \underline{x} \quad \iff \quad \Delta(\underline{x}) \text{ mentions } \underline{y} \quad .$$

2. Let A be a type over L. The **support of A** with respect to Δ, notation $\mathbf{support}_{\Delta}(A)$, is the union of the supports of all labels mentioned in A.

Equivalently: the support of A with respect to Δ is the least subset $U \subseteq L$ such that U contains all labels mentioned in A, and for each $\underline{x} \in U$, we have that U also contains all labels mentioned in $\Delta(\underline{x})$.

Example. A value type $A \to B \,!\, \theta$ can mention labels in A, in B, and in θ.

Definition 30. Let Δ be a label precontext. We call Δ a **label context** if the support of each label does not contain that label. Or equivalently, when the graph of labels with $\prec_{\Delta}$ is acyclic.

Proposition 31. Let Δ be a label precontext on L. Then Δ is a label context iff there is a total ordering on $L = \{x_1, \ldots, x_{|L|}\}$ such that for each i, $\Delta(x_i)$ may mention only $x_1, \ldots, x_{i-1}$.

Examples.

1. $\Delta = \{\underline{x} : \mathrm{nat}, \underline{y} : \mathrm{nat}\}$ is a label context.

2. $\Delta = \{\underline{x} : \mathrm{nat} \to (\mathrm{nat} \to \mathrm{nat}\,!\,\underline{y})\,!\,\varnothing,\ \underline{y} : \mathrm{nat}\}$ is a label context with $\underline{x} \succ_\Delta \underline{y}$; we write $\underline{y}$ for the one-element set $\{\underline{y}\}$.

3. $\Delta = \{\underline{x} : \mathrm{nat}, \underline{y} : \mathrm{nat} \to \mathrm{nat}\,!\,\underline{y}\}$ *is not* a label context, because of the cycle $\underline{y} \prec_\Delta \underline{y}$.

4. $\Delta = \{\underline{x} : \mathrm{nat} \to \mathrm{nat}\,!\,\underline{y},\ \underline{y} : \mathrm{nat} \to \mathrm{nat}\,!\,\underline{x}\}$ *is not* a label context, because of the cycle $\underline{x} \prec_\Delta \underline{y} \prec_\Delta \underline{x}$.

Convention. We will sometimes write $\Delta = \{\underline{x}_1 : A_1,\ \underline{x}_2 : A_2,\ \cdots,\ \underline{x}_n : A_n\}$ for the label context on $\{\underline{x}_1, \cdots, \underline{x}_n\}$.

Definition 32 (extended label context)**.** Let Δ be a label context on L, A be a value type over L, and let $\underline{x}$ be a fresh label $\notin \mathrm{dom}(\Delta)$. Then $(\Delta, \underline{x}{:}A) = \Delta \cup \{\underline{x} \mapsto A\}$ is a label context on $L \cup \{\underline{x}\}$.

Self-contained subsets

Definition 33. A **self-contained subset** K of a label precontext Δ on L is a subset $K \subseteq L$ such that Δ assigns to every label in K a value type over K.

Proposition 34. Let Δ be a label precontext on a finite set L. A subset $K \subseteq L$ is a self-contained subset of Δ iff $\Delta|_K$ is also a label precontext. If additionally Δ is a label context, then $\Delta|_K$ is also a label context. The whole of L is always self-contained, and so is the empty set.

Proposition 35. Let Δ be a label precontext. The intersection of two self-contained subsets of Δ is again a self-contained subset of Δ.

Proposition 36. Let Δ be a label context on L, and let A be a type over L. Then $\mathrm{support}_\Delta(A)$ is a self-contained subset of Δ, and A is a type over $\mathrm{support}_\Delta(A)$. In fact, $\mathrm{support}_\Delta(A)$ is the smallest such set that is both (1) self-contained and (2) large enough that A is a type over it.

Semantics of types

Definition 37 (semantics of types in ordered label context)**.** Let Δ be an ordered label context on L. Then there is a unique function $[\![\Delta \vdash -]\!]$ from types to sets satisfying the following equations.

$$\begin{aligned}
[\![\Delta \vdash 1]\!] &= \{\star\} \\
[\![\Delta \vdash \mathrm{nat}]\!] &= \mathbb{N} \\
[\![\Delta \vdash A + B]\!] &= [\![\Delta \vdash A]\!] + [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \times B]\!] &= [\![\Delta \vdash A]\!] \times [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \to \underline{B}\,]\!] &= [\![\Delta \vdash A]\!] \to [\![\Delta \vdash \underline{B}\,]\!] \\
[\![\Delta \vdash B\,!\,\theta]\!] &= [\![\Delta \vdash B]\!] + \sum_{\underline{x} \in \theta} [\![\Delta \vdash \Delta(\underline{x})]\!] + \{\bot\}
\end{aligned} \tag{5.1}$$

The definition is inductive because in occurrences of $[\![\Delta \vdash A]\!]$ on the right hand side, either the support of A is strictly smaller, or support is the same and the size of A is strictly smaller.

We can slightly decompose this definition by defining the semantics of the subset of active labels $\theta \subseteq \mathrm{dom}(\Delta)$,

$$[\![\Delta \vdash \theta]\!] = \sum_{\underline{x} \in \theta} [\![\Delta \vdash \Delta(\underline{x})]\!]$$

so that we can rewrite the equation for $[\![\Delta \vdash B\,!\,\theta]\!]$ as simply:

$$[\![\Delta \vdash B\,!\,\theta]\!] = [\![\Delta \vdash B]\!] + [\![\Delta \vdash \theta]\!] + \{\bot\}$$

Remark. If we make all the θ in computation types equal to $\varnothing$, then this is exactly the semantics of Chapter 3.

The semantics of types depends not on the size of Δ, but merely on the type that it assigns to some labels:

Lemma 38. Let Δ be an label context, with self-contained subset K, and let A be a value type over K. Then $[\![\Delta|_K \vdash A]\!] = [\![\Delta \vdash A]\!]$. Similarly, for any computation type $\underline{A}$, we have $[\![\Delta|_K \vdash \underline{A}]\!] = [\![\Delta \vdash \underline{A}]\!]$.

For instance, we can take $\mathrm{support}_\Delta(A)$ or $\mathrm{support}_\Delta(\underline{A})$ for K (recall Prop. 36).

Subtyping

Our type system has the type $A \to B \,!\, \phi$ of functions that take arguments of type A and return a value of type B, while using labels in ϕ. But if $\phi \subseteq \theta$, then we want to easily and implicitly convert a function $f : A \to B \,!\, \phi$ to $f : A \to B \,!\, \theta$.

We introduce a subtype system that does exactly this, and nothing more. Furthermore, functions, products, and sums are contra/covariant in the usual way.

Definition 39. Let L be a finite set of labels. We define a partial order of *subtypes* over L:

$$\frac{}{1 \leq 1} \qquad \frac{}{\mathrm{nat} \leq \mathrm{nat}} \qquad \frac{A \leq A' \qquad B \leq B'}{A + B \;\leq\; A' + B'}$$

$$\frac{A \leq A' \qquad B \leq B'}{A \times B \;\leq\; A' \times B'} \qquad \frac{A' \leq A \qquad B \leq B' \qquad \phi \subseteq \theta}{A \to B \,!\, \phi \;\leq\; A' \to B' \,!\, \theta}$$

We omit L from the subtyping judgement, as the judgement is not conditional on it: two types over L are $\leq$ iff they are $\leq$ as types over *any* superset of the set of labels mentioned in the two types.

We will have the obvious typing rules for coercion of values and computations:

$$\frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A \qquad A \leq B}{\Delta; \Gamma \vdash_{\mathrm{v}} V : B} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \phi \qquad A \leq B \qquad \phi \subseteq \theta}{\Delta; \Gamma \vdash_{\mathrm{c}} M : B \,!\, \theta}$$

The following collection of maps are useful for the semantics of coercion.

Definition 40. For a label context Δ on L, and $A \leq A'$ over L, we define a "semantic coercion" map $[\![\Delta \vdash A \leq A']\!]$:

$$[\![A \quad \leq A']\!] \quad : [\![\Delta \vdash A]\!] \quad \to [\![\Delta \vdash A']\!]$$

And if additionally $\theta \subseteq \theta' \subseteq L$ then we also define a semantic coercion map $[\![\Delta \vdash A \,!\, \theta \leq A' \,!\, \theta']\!]$:

$$[\![A \,!\, \theta \leq A' \,!\, \theta']\!] \; : \; [\![\Delta \vdash A \,!\, \theta]\!] \; \to \; [\![\Delta \vdash A' \,!\, \theta']\!]$$

which is defined in the obvious way: by mutual induction on the derivation of $A \leq A'$:

$$\begin{array}{llll}
[\![\Delta \vdash 1 & \leq 1 &]\!](\star) & = \star \\
[\![\Delta \vdash \mathrm{nat} & \leq \mathrm{nat} &]\!](n) & = n \\
[\![\Delta \vdash A + B & \leq A' + B' &]\!] & = [\![\Delta \vdash A \leq A']\!] + [\![\Delta \vdash B \leq B']\!] \\
[\![\Delta \vdash A \times B & \leq A' \times B' &]\!] & = [\![\Delta \vdash A \leq A']\!] \times [\![\Delta \vdash B \leq B']\!] \\
[\![\Delta \vdash (A \,!\, \theta) & \leq (A' \,!\, \theta') &]\!] & = [\![\Delta \vdash A \leq A']\!] + (\hookrightarrow) + \mathrm{id} \\
[\![\Delta \vdash (A' \to \underline{B}) & \leq (A \to \underline{B}')]\!](f) & = [\![\Delta \vdash \underline{B} \leq \underline{B}']\!] \circ f \circ [\![\Delta \vdash A \leq A']\!]
\end{array}$$

where $+, \times, \hookrightarrow$ have the usual meaning[3], and id is the identity.

5.3 Terms

Definition 41 (context, variables)**.** Let L be a finite set of labels. A **context over** L, typically denoted Γ, is a finite set of *variables*, together with for each variable a type over L.

We use a fine-grain call-by-value language, like before.

Definition 42. Our **terms** are of the form

$\boxed{L; \Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta}$ for computations, and

$\boxed{L; \Delta; \Gamma \vdash_{\mathrm{v}} V : A}$ for values,

[3] Suppose that $f : W \to X,\ g : Y \to Z$. Then $f + g : W + Y \to X + Z$ is the function that uses f, g depending on the input; function $f \times g : W \times Y \to X \times Z$ uses f and g on each component. The inclusion map $\hookrightarrow : X \to X \cup Y$ maps each element in A to itself.

where Δ is a label context on L, Γ and A are a context and a type over L, and θ is a subset of L. Term are generated by the derivation rules below. We usually omit L for conciseness, but it can always be reconstructed because $L = \mathrm{dom}(\Delta)$.

We may sometimes write just M (resp. V) for the computation (resp. value), leaving the rest of the data implied. When we want to emphasize a computation or value in its full $\vdash$-form, we will talk about a *computation* or *value judgement*. A term may be the conclusion of multiple derivations; by definition it is the conclusion of at least one derivation.

Rules for fine-grain call-by-value plus effect typing

$$\frac{(x{:}A) \in \Gamma}{\Delta; \Gamma \vdash_{\mathrm{v}} x : A} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B \,!\, \theta}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{let}\ V\ \mathrm{be}\ x.\, M : B \,!\, \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{return}\ V : A \,!\, \theta} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} N : B \,!\, \theta}{\Delta; \Gamma \vdash_{\mathrm{c}} M\ \mathrm{to}\ x.\, N : B \,!\, \theta}$$

$$\frac{\Delta; \Gamma, x{:}A \vdash_{\mathrm{c}} M : B \,!\, \theta}{\Delta; \Gamma \vdash_{\mathrm{v}} \lambda x.\, M : A \to B \,!\, \theta} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A \to B \,!\, \theta \qquad \Delta; \Gamma \vdash_{\mathrm{v}} W : A}{\Delta; \Gamma \vdash_{\mathrm{c}} V\, W : B \,!\, \theta}$$

Rules for iteration

$$\frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta, \underline{x}{:}A;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : B \,!\, (\theta \cup \{\underline{x}\})}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{iter}\ V, x.\, M : B \,!\, \theta} \qquad \frac{\underline{x} \in \theta \qquad \Delta; \Gamma \vdash_{\mathrm{v}} V : \Delta(\underline{x})}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathrm{raise}_{\underline{x}}\, V : B \,!\, \theta}$$

Rules for coercion

$$\frac{\Delta; \Gamma \vdash_{\mathrm{v}} V : A \qquad A \leq B}{\Delta; \Gamma \vdash_{\mathrm{v}} V : B} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \phi \qquad A \leq B \qquad \phi \subseteq \theta}{\Delta; \Gamma \vdash_{\mathrm{c}} M : B \,!\, \theta}$$

Rules for sum and product types

$$\frac{}{\Delta;\Gamma \vdash_{\mathrm{v}} \langle\rangle : 1} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A \quad \Delta;\Gamma \vdash_{\mathrm{v}} W : B}{\Delta;\Gamma \vdash_{\mathrm{v}} \langle V, W\rangle : A \times B} \qquad \frac{}{\Delta;\Gamma \vdash_{\mathrm{v}} 0 : \mathrm{nat}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : \mathrm{nat}}{\Delta;\Gamma \vdash_{\mathrm{v}} \mathrm{succ}\, V : \mathrm{nat}} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A}{\Delta;\Gamma \vdash_{\mathrm{v}} \mathrm{inl}\, V : A + B} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : B}{\Delta;\Gamma \vdash_{\mathrm{v}} \mathrm{inr}\, V : A + B}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : \mathrm{nat} \qquad \Delta;\Gamma \vdash_{\mathrm{c}} M : \underline{C} \qquad \Delta;\ \Gamma, x{:}\mathrm{nat} \vdash_{\mathrm{c}} N : \underline{C}}{\Delta;\Gamma \vdash_{\mathrm{c}} \mathrm{case}\, V \,\mathrm{of}\, \{0.\, M;\ \mathrm{succ}\, x.\, N\} \,!\, \theta : \underline{C}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A + B \qquad \Delta;\ \Gamma, x{:}A \vdash_{\mathrm{c}} M : \underline{C} \qquad \Delta;\ \Gamma, y{:}B \vdash_{\mathrm{c}} N : \underline{C}}{\Delta;\Gamma \vdash_{\mathrm{c}} \mathrm{case}\, V \,\mathrm{of}\, \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\} : \underline{C}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A \times B \qquad \Delta;\ \Gamma, x{:}A, y{:}B \vdash_{\mathrm{c}} M : \underline{C}}{\Delta;\Gamma \vdash_{\mathrm{c}} \mathrm{case}\, V \,\mathrm{of}\, \langle x, y\rangle.\, M : \underline{C}}$$

5.4 Semantics of term derivations

In this section, we define the semantics of computation and value *derivations*. Later, in Theorem 49, we show that all derivations of a term have the same semantics, so that in Definition 50 we can define the semantics of a term to be the semantics of any of its derivations.

We have the usual semantics of contexts Γ:

Definition 43. Let Δ be a label context. The semantics $[\![\Delta \vdash \Gamma]\!]$ of a context over L is the product of the semantics of the constituent types:

$$[\![\Delta \vdash \Gamma]\!] = \prod_{(x:A)\in\Gamma} [\![\Delta \vdash A]\!]$$

Our semantics will be of the following shape. For a derivation D of $\Delta;\Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$ we will have:

$$[\![\Delta;\Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta]\!] \ :\ [\![\Delta \vdash \Gamma]\!] \longrightarrow [\![\Delta \vdash A \,!\, \theta]\!]$$

And for a derivation D of $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$ we will have:

$$[\![\Delta; \Gamma \vdash_{\mathrm{c}} V : A]\!] \ : \ [\![\Delta \vdash \Gamma]\!] \longrightarrow [\![\Delta \vdash A]\!]$$

To indicate that D such a derivation, we may write $D :: \Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$ or $D :: \Delta; \Gamma \vdash_{\mathrm{v}} V : A$.

The semantics of term derivations is by induction on the derivation. It happens to agree with the semantics of Chapter 3 when all θ in function types are empty, and all θ in term judgements are L.

We will now proceed to define $[\![D]\!]$. We do a case distinction on the rule at the bottom of the derivation.

Semantics of coercion rules

Suppose that the last rule in a derivation D is an application of value coercion:

$$\frac{D' :: \Delta; \Gamma \vdash_{\mathrm{v}} V : A \qquad A \leq A'}{D :: \Delta; \Gamma \vdash_{\mathrm{v}} V : A'}$$

We define $[\![D]\!]$ as the composition $[\![\Delta \vdash \Gamma]\!] \xrightarrow{[\![D']\!]} [\![\Delta \vdash A]\!] \xrightarrow{[\![\Delta \vdash A \leq A']\!]} [\![\Delta \vdash A']\!]$. Similarly, the semantics of a computation coercion is postcomposition with $[\![\Delta \vdash A \,!\, \theta \leq A' \,!\, \theta']\!]$.

Semantics of non-coercion rules

We will present the semantics of these derivations as if they were the semantics of judgements. For instance, the semantics of let is as follows:

> *Suppose a derivation D ends with the rule for* let*, and D_V and D_M are the derivations for V and M. Then the semantics of D are* $[\![D]\!]_\rho = [\![D_M]\!]_{(\rho, x \mapsto [\![D_V]\!]_\rho)}$.

However, for the definition we will only write: $[\![\mathrm{let}\ V\ \mathrm{be}\ x.\ M]\!]_\rho = [\![M]\!]_{(\rho, x \mapsto [\![V]\!]_\rho)}$. This an abuse of notation for brevity. Recall $return, raise, \bot$ defined in Definition 40, which we use together with this slight abuse of notation to proceed to give semantics of derivations.

To make sense of these definitions, it is important to recall from Lemma 38 that the semantics of a type $[\![\Delta \vdash A]\!]$ (resp. $[\![\Delta \vdash \underline{A}]\!]$) does not depend on how big Δ is, but only on the types assigned to the labels in the support of A (resp. $\underline{A}$).

$$[\![x]\!]_\rho = \rho(x)$$

$$[\![\langle\rangle]\!]_\rho = \langle\rangle$$

$$[\![0]\!]_\rho = 0$$

$$[\![\text{succ } V]\!]_\rho = 1 + [\![V]\!]_\rho$$

$$[\![\text{inl } V]\!]_\rho = \text{inl } [\![V]\!]_\rho$$

$$[\![\text{inr } V]\!]_\rho = \text{inr } [\![V]\!]_\rho$$

$$[\![\langle V, W\rangle]\!]_\rho = \langle [\![V]\!]_\rho, [\![W]\!]_\rho\rangle$$

$$[\![\lambda x.\ M]\!]_\rho = \lambda(a{\in}[\![A]\!]).[\![M]\!]_{(\rho, x\mapsto a)}$$

$$[\![\text{return } V]\!]_\rho = \mathit{return}\ [\![V]\!]_\rho$$

$$[\![\text{raise}_{\underline{x}}\ V]\!]_\rho = \mathit{raise}_{\underline{x}}\ [\![V]\!]_\rho$$

$$[\![\text{let } V \text{ be } x.\ M]\!]_\rho = [\![M]\!]_{(\rho, x\mapsto [\![V]\!]_\rho)}$$

$$[\![M \text{ to } x.\ N]\!]_\rho = \begin{cases} [\![N]\!]_{(\rho, x\mapsto v)} & \text{if } [\![M]\!]_\rho = \mathit{return}\ v \\ \mathit{raise}_{\underline{y}}\ w & \text{if } [\![M]\!]_\rho = \mathit{raise}_{\underline{y}}\ w \\ \bot & \text{if } [\![M]\!]_\rho = \ \bot \end{cases}$$

$$[\![V\ W]\!]_\rho = [\![V]\!]_\rho\ [\![W]\!]_\rho$$

$$[\![\text{case } V \text{ of } \{0.\ M;\ \text{succ } x.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_\rho & \text{if } [\![V]\!]_\rho = 0 \\ [\![N]\!]_{(\rho, x\mapsto n)} & \text{if } [\![V]\!]_\rho = 1 + n \end{cases}$$

$$[\![\text{case } V \text{ of } \{\text{inl } x.\ M;\ \text{inr } y.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_{(\rho, x\mapsto a)} & \text{if } [\![V]\!]_\rho = \text{inl } a \\ [\![N]\!]_{(\rho, y\mapsto b)} & \text{if } [\![V]\!]_\rho = \text{inr } b \end{cases}$$

$$[\![\text{case } V \text{ of } \{\langle x, y\rangle.\ M\}]\!]_\rho = [\![M]\!]_{(\rho, x\mapsto a, y\mapsto b)} \qquad \text{if } [\![V]\!]_\rho = \langle a, b\rangle$$

$$[\![\text{iter } V, x.\ M]\!]_\rho = \begin{cases} \mathit{return}\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge\ \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_{\underline{x}}\ v_{i+1} \\ & \quad \wedge\ [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{return}\ w \\ \mathit{raise}_{\underline{y}}\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge\ \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_{\underline{x}}\ v_{i+1} \\ & \quad \wedge\ [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{raise}_{\underline{y}}\ w \\ & \quad \wedge\ \underline{y} \text{ is not the same label as } \underline{x} \\ \bot & \text{if no other case matches} \end{cases}$$

This completes the definition of the semantics of derivations.

5.5 Coherence

In this section, we prove that all derivations of the same typing judgement have the same semantics. This means that we now have a semantics of *typing judgements* ("semantics of terms"), rather than merely a semantics of derivations. We follow the approach from Chapter 4, until in Definition 50 we can define the semantics of terms as the semantics of one of its derivations.

Let L be a finite set, and let Δ, Δ' be two label contexts on L. We define a relation R for every pair of value types over L, and every pair of computation types over L.

$$\begin{array}{rcl} R[\Delta \vdash A; \quad \Delta' \vdash A' \quad] & \subseteq & [\![\Delta \vdash A]\!] \quad \times \; [\![\Delta' \vdash A']\!] \\ R[\Delta \vdash A \,!\, \theta; \Delta' \vdash A' \,!\, \theta'] & \subseteq & [\![\Delta \vdash A \,!\, \theta]\!] \; \times \; [\![\Delta' \vdash A' \,!\, \theta']\!] \end{array}$$

We employ six kinds of value types: natural numbers, nullary sums (0), binary sums ($A + B$), nullary products (1), binary products ($A \times B$), and function types ($A \to \underline{B}$). When A and A' are different kinds of value types, then $R[\Delta \vdash A; \Delta' \vdash A']$ is the complete relation. Otherwise, we define R as follows. Similar to Def. 37, we use induction first on the support of the type, and then on the type itself.

$$\begin{array}{rcl} m \; R\big[\Delta \vdash \mathrm{nat}; \quad \Delta' \vdash \mathrm{nat}\big] \; n & & \text{iff} \quad m = n \\ R\big[\Delta \vdash 0; \quad \Delta' \vdash 0\big] & = & \varnothing \\ (inl \; v) \; R\big[\Delta \vdash A{+}B; \; \Delta' \vdash A'{+}B'\big] \; (inl \; w) & & \text{iff} \quad v \, R\big[\Delta \vdash A; \Delta' \vdash A'\big] \, w \\ (inr \; v) \; R\big[\Delta \vdash A{+}B; \; \Delta' \vdash A'{+}B'\big] \; (inr \; w) & & \text{iff} \quad v \, R\big[\Delta \vdash B; \Delta' \vdash B'\big] \, w \\ \neg \Big(v \, R\big[\Delta \vdash A{+}B; \; \Delta' \vdash A'{+}B'\big] \; w\Big) & & \text{otherwise} \\ \star \, R\big[\Delta \vdash 1; \quad \Delta' \vdash 1\big] \, \star & & \\ \langle v_1, v_2 \rangle \; R\big[\Delta \vdash A{\times}B; \; \Delta' \vdash A'{\times}B'\big] \; \langle w_1, w_2 \rangle & & \text{iff} \quad v_1 \, R\big[\Delta \vdash A; \Delta' \vdash A'\big] \, w_1 \\ & & \quad \text{and} \;\; v_2 \, R\big[\Delta \vdash B; \Delta' \vdash B'\big] \, w_2 \end{array}$$

$$f\ R\big[\Delta \vdash A{\to}\underline{B};\, \Delta' \vdash A'{\to}\underline{B'}\big]\ g \qquad \text{iff} \quad \forall\,\big(x\ R\big[\Delta{\vdash}A;\, \Delta'{\vdash}A'\big]\ x'\big) : (fx)\ R\big[\Delta{\vdash}\underline{B};\, \Delta'{\vdash}\underline{B'}\big]\ (gx')$$

$$(\mathit{return}\ v)\ R\big[\Delta \vdash B\,!\,\theta;\ \ \Delta' \vdash B'\,!\,\theta'\big]\ (\mathit{return}\ w) \qquad \text{iff} \quad v\ R\big[\Delta \vdash B;\, \Delta' \vdash B'\big]\ w$$

$$(\mathit{raise}_{\underline{x}}\ v)\ R\big[\Delta \vdash B\,!\,\theta;\ \ \Delta' \vdash B'\,!\,\theta'\big]\ (\mathit{raise}_{\underline{x}}\ w) \qquad \text{iff} \quad v\ R\big[\Delta \vdash \Delta(\underline{x});\, \Delta' \vdash \Delta'(\underline{x})\big]\ w$$

$$\neg\Big((\mathit{raise}_{\underline{x}}\ v)\ R\big[\Delta \vdash B\,!\,\theta;\ \ \Delta' \vdash B'\,!\,\theta'\big]\ (\mathit{raise}_{\underline{y}}\ w)\Big) \qquad \text{when} \quad \underline{x} \neq \underline{y}$$

$$\bot\ R\big[\Delta \vdash B\,!\,\theta;\ \ \Delta' \vdash B'\,!\,\theta'\big]\ \bot$$

We also define R on environments, as long as the corresponding contexts have the same set of variables, say X:

$$R[\Delta \vdash \Gamma;\, \Delta' \vdash \Gamma'] \quad \subseteq \quad \textstyle\prod_{x\in X}[\![\Delta \vdash \Gamma(x)]\!] \times \prod_{x\in X}[\![\Delta' \vdash \Gamma'(x)]\!]$$

$$\rho\ R[\Delta \vdash \Gamma;\, \Delta' \vdash \Gamma']\ \rho' \qquad \text{iff} \quad \forall x{\in}X :\ \rho(x)\ R[\Delta \vdash \Gamma(x)\,;\, \Delta' \vdash \Gamma'(x)]\ \rho'(x)$$

Furthermore, we define R on functions from environments to semantic values and computations ("semantic judgements"), as long as again the contexts are on the same set of variables X:

$$R[\Delta,\Gamma{\vdash}A;\, \Delta',\Gamma'{\vdash}A'] \ \subseteq\ \big((\textstyle\prod_{x\in X}[\![\Delta{\vdash}\Gamma(x)]\!]) \to [\![\Delta{\vdash}A]\!]\big) \times \big((\prod_{x\in X}[\![\Delta'{\vdash}\Gamma'(x)]\!]) \to [\![\Delta'{\vdash}A']\!]\big)$$

$$f\ R[\Delta,\Gamma{\vdash}A;\, \Delta',\Gamma'{\vdash}A']\ g \qquad \text{iff} \quad \forall\rho\ R[\Delta{\vdash}\Gamma;\, \Delta'{\vdash}\Gamma']\ \rho' :\ f(\rho)\ R[\Delta{\vdash}A;\, \Delta'{\vdash}A']\ g(\rho')$$

$$R[\Delta,\Gamma{\vdash}\underline{A};\, \Delta',\Gamma'{\vdash}\underline{A'}] \ \subseteq\ \big((\textstyle\prod_{x\in X}[\![\Delta{\vdash}\Gamma(x)]\!]) \to [\![\Delta{\vdash}\underline{A}]\!]\big) \times \big((\prod_{x\in X}[\![\Delta'{\vdash}\Gamma'(x)]\!]) \to [\![\Delta'{\vdash}\underline{A'}]\!]\big)$$

$$c\ R[\Delta,\Gamma{\vdash}\underline{A};\, \Delta',\Gamma'{\vdash}\underline{A'}]\ d \qquad \text{iff} \quad \forall\rho\ R[\Delta{\vdash}\Gamma;\, \Delta'{\vdash}\Gamma']\ \rho' :\ c(\rho)\ R[\Delta{\vdash}\underline{A};\, \Delta'{\vdash}\underline{A'}]\ d(\rho')$$

Proposition 44. Let L be a finite set and Δ be a label context on L. Then for every value type A, $R[\Delta \vdash A;\, \Delta \vdash A]$ is the diagonal. And for every computation type $\underline{A}$, $R[\Delta \vdash \underline{A};\, \Delta \vdash \underline{A}]$ is the diagonal. Analogously for contexts and judgements.

Proposition 45 (symmetry). Let L be a finite set, and let Δ, Δ' be two label contexts on L.

- For value types A, B over L, $v\ R[\Delta \vdash A;\, \Delta' \vdash B]\ w \ \Leftrightarrow\ w\ R[\Delta \vdash B;\, \Delta' \vdash A]\ v$.
- For computation types $\underline{A}, \underline{B}$ over some finite set L, $c\ R[\Delta \vdash \underline{A};\, \Delta' \vdash \underline{B}]\ d \ \Leftrightarrow\ d\ R[\Delta \vdash \underline{B};\, \Delta' \vdash \underline{A}]\ c$.

- The analogous bi-implications hold for contexts, value judgements, and computation judgements.

The following shows compatibility with semantic coercion $[\![\Delta \vdash A \leq B]\!], [\![\Delta \vdash \underline{A} \leq \underline{B}]\!]$, defined in Definition 40.

Lemma 46 (semantic coercion preserves R). Let L be a finite set, Δ, Δ' be label contexts on L, and A, A', B be three types over L with $A' \leq B$.

1. If $v\, R\big[\Delta \vdash A; \Delta' \vdash A'\big]\, w$, then also $v\, R\big[\Delta \vdash A; \Delta' \vdash B\big]\, [\![\Delta' \vdash A' \leq B]\!](w)$.

Let also $\theta \subseteq \psi \subseteq L$ and $\theta' \subseteq \psi' \subseteq L$.

2. If $\theta \subseteq L$ and $\theta' \subseteq \psi \subseteq L$ and $c\, R\big[\Delta \vdash A!\theta; \Delta' \vdash A'!\theta'\big]\, d$,
   then also $c\, R\big[\Delta \vdash A!\theta; \Delta' \vdash B!\psi\big]\, [\![\Delta' \vdash A'!\theta' \leq B!\psi]\!](d)$.

The proof is by induction on the derivation of $A \leq B$. By symmetry, we also know that semantic coercion on the left preserves R.

Notation. Given a value derivation D, write $(A \leq B)\, D$ for D extended with a coercion from A to B at the bottom. Analogously, we write $(A!\theta \leq B!\psi)\, D$ for the coerced computation derivation.

Lemma 47. Let L be a finite set, let Δ, Δ' be two label contexts on L, and let Γ, Γ' be two value contexts on L on the same set of variables $|\Gamma| = |\Gamma'|$. Let A, A', B be three value types on L with $A' \leq B$.

- Let D, D' be two derivations of the same raw value $\mathtt{V}$,

$$D :: \Delta\,; \Gamma \vdash_{\mathrm{v}} \mathtt{V} : A$$
$$D' :: \Delta'; \Gamma' \vdash_{\mathrm{v}} \mathtt{V} : A' \ .$$

  If $\quad [\![D]\!]\, R\Big[\Delta; \Gamma \vdash A;\ \Delta'; \Gamma' \vdash A'\Big]\, [\![D']\!]$,
  then $[\![D]\!]\, R\Big[\Delta; \Gamma \vdash A;\ \Delta'; \Gamma' \vdash B\ \Big]\, [\![(A' \leq B)\, D']\!]$.

- Let $\theta \subseteq L$ and $\theta' \subseteq \psi \subseteq L$, and let D, D' be two derivations of the same raw computation $\mathtt{M}$,

$$D :: \Delta\,;\Gamma \vdash_{\mathrm{c}} \mathtt{M} : A\,!\,\theta$$
$$D' :: \Delta';\Gamma' \vdash_{\mathrm{c}} \mathtt{M} : A'\,!\,\theta' \quad .$$

  If $\quad [\![D]\!]\, R\Big[\Delta;\Gamma\vdash A!\theta;\ \Delta';\Gamma'\vdash A'!\theta'\Big]\, [\![D']\!]$,
  then $[\![D]\!]\, R\Big[\Delta;\Gamma\vdash A!\theta;\ \Delta';\Gamma'\vdash B!\psi\Big]\, [\![(A'!\theta' \leq B!\psi)\, D']\!]$.

- Syntactic coercion on the left preserves R analogously.

Proof. Straightforward using Lemma 46. □

Lemma 48 (compatibility with syntactic constructs). Let L be a finite set, let Δ, Δ' be two label contexts on L, and let Γ, Γ' be two value contexts on L on the same set of variables $|\Gamma| = |\Gamma'|$. Let A, A' be two value types on L.

- Let D, D' be two derivations of the same raw value $\mathtt{V}$,

$$D :: \Delta\,;\Gamma \vdash_{\mathrm{v}} \mathtt{V} : A$$
$$D' :: \Delta';\Gamma' \vdash_{\mathrm{v}} \mathtt{V} : A' \quad .$$

  Then $[\![D]\!]\, R\, [\![D']\!]$.

- Let $\theta \subseteq L$ and $\theta' \subseteq L$, and let D, D' be two derivations of the same raw computation $\mathtt{M}$,

$$D :: \Delta\,;\Gamma \vdash_{\mathrm{c}} \mathtt{M} : A\,!\,\theta$$
$$D' :: \Delta';\Gamma' \vdash_{\mathrm{c}} \mathtt{M} : A'\,!\,\theta' \quad .$$

  Then $[\![D]\!]\, R\, [\![D']\!]$.

Proof. By induction on the combined size of the derivations. If one of the derivations ends in a coercion, then apply Lemma 47. Otherwise, the derivations end in the same syntactic rule. Straightforward in every case. For $\mathrm{iter}\ V, x.\, M$, if $[\![D]\!] \neq \bot$, then by induction on the number of steps to get to return or $\mathrm{raise}_{\underline{y}}$ (for $\underline{x} \neq \underline{y}$). □

Recall from Proposition 44 that R on the same judgement is the diagonal. Then we automatically get:

Theorem 49. Let D, D' be two derivations of the same computation $\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$. Then $[\![D]\!] = [\![D']\!]$.

And similarly, let D, D' be two derivations of the same value $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$. Then $[\![D]\!] = [\![D']\!]$.

Definition 50. The semantics $[\![M]\!]$ of a term M is the semantics of one of the derivations of M. Similarly, the semantics $[\![V]\!]$ of a value V is the semantics of one of the derivations of V.

Proposition 51. The labelled iteration language from §3.3 is a sub-language of the language of this chapter, just by taking $\theta = \varnothing$ everywhere. The denotational semantics coincide.

Corollary 52. The semantics of the labelled iteration language with pure function types from §3.3 is coherent.

Chapter 6

DEFINED ALGEBRAIC OPERATIONS

In this chapter, we proceed to formally define our defined algebraic operation (DAO) language. We gave an introduction to the language in §1.6, and we give further examples in Chapter 8, so here we jump right in.

Remark. As explained in §1.6, we present the language as a fine-grain call-by-value language. For clarity of presentation we only include binary sum and product types; it would be trivial to go further and add arbitrary ground types, or sums or products of arbitrary families of value types.

We present the language with finitely many operations of finite arity, but these restrictions are again purely for presentational reasons: we can generalise the formalisation to use arbitrary sets for the arity of operations, of which there may be arbitrarily many. Similarly, we present value contexts as a finite sequence of variables with types, but this is merely for clarity of presentation: nothing in our formalisation requires value contexts to be ordered or finite.

6.1 Types, arity assignments

Convention. We let L stand for a finite set of operation names in context. We use underlined roman font for its particular elements ($\underline{\text{error}}$, $\underline{\text{read}}$) and α, β, γ as metavariables for the elements of L.

$$\frac{A' \leq A \qquad B \leq B' \qquad \theta \subseteq \theta'}{A \to B \,!\, \theta \;\leq\; A' \to B' \,!\, \theta'}$$

$$\frac{}{0 \leq 0} \qquad \frac{}{1 \leq 1} \qquad \frac{A \leq A' \qquad B \leq B'}{A + B \;\leq\; A' + B'} \qquad \frac{A \leq A' \qquad B \leq B'}{A \times B \;\leq\; A' \times B'}$$

Figure 6.1: Subtyping rules for fine-grain call-by-value with defined operations: the function type rule, and the rules that make $\leq$ into a precongruence.

Definition 53. Let L be a finite set of operation names in context. The **value** and **computation types on** L are generated by the following grammar. We underline computation types, and we have added the empty type 0 and the unit type 1.

value types	$A, B \;::=\; A \to \underline{B} \mid 0 \mid 1 \mid A + B \mid A \times B$
computation types	$\underline{A}, \underline{B} \;::=\; B \,!\, \theta \qquad (\theta \subseteq L)$

We show the subtyping relation in Figure 6.1: arrow is covariant in θ and in the right hand type, and contravariant in the left hand type, and $+$ and $\times$ are both covariant in both sides.

Proposition 54. Given two types A, B over a set L, there is at most one derivation of $A \leq B$. The subtyping relation $\leq$ is a partial order.

Value contexts are as usual:

Definition 55. A **value context over L** — typically denoted Γ — is a finite set of (distinct) variable names, each associated with a type over L.

Value contexts are written $x_1{:}A_1, x_2{:}A_2, \cdots, x_n{:}A_n$. Given a value context Γ, the extension $\Gamma, y{:}B$ is only valid notation if y is not already in Γ.

The width of the operations is specified in the arity assignment:

Definition 56. An **arity assignment on L** (typically denoted by Δ) is a function from L to natural numbers.

For instance, if we have an error $\underline{\text{error}}$ and a binary $\underline{\text{read}}$ operation, then we might have:

$$L = \{\underline{\text{error}}, \underline{\text{read}}\}$$

$$\Delta = \{\underline{\text{error}} : 0, \underline{\text{read}} : 2\}$$

The *arity*, indicated after the colon, is the number of ways in which execution may be resumed after the operation's completion.

Given an arity assignment Δ on L, note that we can reobtain L as the domain of Δ. We will use this fact to be more concise, and commonly merely speak of "an arity assignment Δ", "a value context over $\mathrm{dom}(\Delta)$", and "a type over $\mathrm{dom}(\Delta)$".

For convenience, we call a **typing context** a pair $\Delta; \Gamma$, where Δ is an arity assignment and Γ is a value context over $\mathrm{dom}(\Delta)$. As neither operations nor variables have a globally intrinsic arity/type, we put this information in the typing context so that we can type-check terms, and determine the semantics of them. In particular, Δ is needed to find the semantics of types.

For the semantics of types, we must first give some basic definitions for trees with multiple node types such as above: $\underline{\text{error}}$ nodes are nullary while $\underline{\text{read}}$ nodes are binary. In this thesis, we only deal with terminating programs, so well-founded trees will suffice.

6.2 Well-founded trees over a signature

Definition 57. A **signature** $\mathcal{S}$ is a set $|\mathcal{S}|$, whose elements are called "operations", together with an "arity" set $\mathrm{arity}_{\mathcal{S}}(s)$ for each operation $s \in |\mathcal{S}|$. We also call $|\mathcal{S}|$ the **underlying set**.

We can take the union of two signatures with disjoint underlying sets:

Definition 58. Suppose we have two signatures $\mathcal{S}$ and $\mathcal{S}'$ with disjoint underlying sets. Then $\boldsymbol{\mathcal{S} \cup \mathcal{S}'}$ is the signature with underlying set $|\mathcal{S} \cup \mathcal{S}'| = |\mathcal{S}| \cup |\mathcal{S}'|$, and

$$\mathrm{arity}_{\mathcal{S}\cup\mathcal{S}'}(s) = \mathrm{arity}_{\mathcal{S}}(s) \qquad \text{when } \; s \in |\mathcal{S}|$$

$$\mathrm{arity}_{\mathcal{S}\cup\mathcal{S}'}(s) = \mathrm{arity}_{\mathcal{S}'}(s) \qquad \text{when } \; s \in |\mathcal{S}'|$$

We write $\varnothing$ for the empty signature.

One level of trees over the signature is captured by the functor $F_{\mathcal{S}}$:

Definition 59. The endofunctor $F_{\mathcal{S}} : \mathrm{Set} \to \mathrm{Set}$ of "$\mathcal{S}$-trees of exactly one level" maps X to $F_{\mathcal{S}}(X) = \sum_{s \in |\mathcal{S}|} X^{\mathrm{arity}_{\mathcal{S}}(s)}$.

Definition 60. Let $\mathcal{S}$ be a signature, and X be a set. We inductively define a (well-founded) $\mathcal{S}$-tree over X to be either

- a "leaf" containing a value in X, or
- a "node" with a choice of operation $s \in |\mathcal{S}|$ and an $\mathrm{arity}_{\mathcal{S}}(s)$-collection of $\mathcal{S}$-trees over X.

We write $\mathcal{S}\text{-tree}(X)$ for the set of such well-founded trees. The trees in either case are denoted $\mathrm{leaf}(x)$ or $\boldsymbol{s}(\vec{\kappa})$: when we create a proper tree, we tend to write the operation on top in **bold**.

Proposition 61. Let $\mathcal{S}$ be a signature. Then $\mathcal{S}\text{-tree}(-)$ is the free monad over $F_{\mathcal{S}}$.

We will use the following definition for the semantics of operation definitions.

Definition 62. Let $\mathcal{S}$ and $\mathcal{S}'$ be two signatures with disjoint underlying sets. Let X be a set, and $f : F_{\mathcal{S}'}(\mathcal{S}\text{-tree}(X)) \to \mathcal{S}\text{-tree}(X)$. Then we define the *extended initial algebra morphism*

$$f^{\mathrm{ext}} \ : \ (\mathcal{S} \cup \mathcal{S}')\text{-tree}(X) \to \mathcal{S}\text{-tree}(X)$$

as

$$\begin{aligned} f^{\mathrm{ext}}(\mathrm{leaf}(x)) &= \mathrm{leaf}(x) \\ f^{\mathrm{ext}}(\boldsymbol{s}(\vec{\kappa})) &= \boldsymbol{s}\Big(\overrightarrow{f^{\mathrm{ext}}(\kappa)}\Big) \qquad \text{if } \ s \in |\mathcal{S}| \\ f^{\mathrm{ext}}(\boldsymbol{s}(\vec{\kappa})) &= \boldsymbol{f}_s\Big(\overrightarrow{f^{\mathrm{ext}}(\kappa)}\Big) \qquad \text{if } \ s \in |\mathcal{S}'| \quad . \end{aligned}$$

For a set Γ and a binary function $f : \Big(\Gamma \times F_{\mathcal{S}'}(\mathcal{S}\text{-tree}(X))\Big) \to \mathcal{S}\text{-tree}(X)$, we conveniently also write f^{ext} for the function $(\rho \mapsto (f\,\langle \rho, - \rangle)^{\mathrm{ext}}) \ : \ \Gamma \to \Big((\mathcal{S} \cup \mathcal{S}')\text{-tree}(X) \to \mathcal{S}\text{-tree}(X)\Big)$, or also for the uncurried variant $\Big(\Gamma \times (\mathcal{S} \cup \mathcal{S}')\text{-tree}(X)\Big) \to \mathcal{S}\text{-tree}(X)$.

Definition 63. A signature $\mathcal{S}$ is a **subsignature** of another signature $\mathcal{S}'$ (notation: $\boldsymbol{\mathcal{S} \subseteq \mathcal{S}'}$) when $|\mathcal{S}| \subseteq |\mathcal{S}'|$ and the arities of the common operations are the same.

If $\mathcal{S}$ and $\mathcal{S}'$ have disjoint underlying sets, then $\mathcal{S}$ is a subsignature of $\mathcal{S} \cup \mathcal{S}'$. Conversely, if $\mathcal{S} \subseteq \mathcal{S}'$, then $|\mathcal{S}| \cap |\mathcal{S}' \setminus \mathcal{S}| = \varnothing$ and $\mathcal{S}'$ arises uniquely as a union of $\mathcal{S}$ with a signature with underlying set $|\mathcal{S}'| \setminus |\mathcal{S}|$.

Definition 64. Suppose we have three signatures $\mathcal{S}$, $\mathcal{S}'$, and $\mathcal{S}''$ with pairwise disjoint underlying sets, two sets X, Y, and a function $f : (\mathcal{S} \cup \mathcal{S}')\text{-tree}(X) \longrightarrow (\mathcal{S} \cup \mathcal{S}'')\text{-tree}(Y)$. We say that $\boldsymbol{f}$ **preserves the** $\boldsymbol{\mathcal{S}}$**-algebra** when for all $s \in |\mathcal{S}|$ and $\vec{\kappa} \in (\mathcal{S} \cup \mathcal{S}')\text{-tree}(X)^{\text{arity}_{\mathcal{S}}(s)}$, the following two trees are the same:

$$f\big(\boldsymbol{s}(\vec{\kappa})\big) \;\; = \;\; \boldsymbol{s}\Big(\overrightarrow{f(\kappa)}\Big)$$

The following is trivial by definition.

Proposition 65. Suppose that we have two signatures $\mathcal{S} \cup \mathcal{S}'$ with disjoint underlying sets, a set X, and a function $f : F_{\mathcal{S}'}(\mathcal{S}\text{-tree}(X)) \to \mathcal{S}\text{-tree}(X)$. Then f^{ext} preserves the $\mathcal{S}$-algebra.

6.3 Semantics of types and value contexts

The semantics of an arity assignment Δ is a signature, namely

$$[\![\text{dom}(\Delta); \Delta]\!] = \text{dom}(\Delta) \text{ with } \text{arity}(\alpha) = \{1, .., \Delta(\alpha)\} \quad .$$

We can also talk about the **semantics of a restriction** $\boldsymbol{\theta}$ **of** $\boldsymbol{\Delta}$, which is simply

$$[\![\theta; \Delta]\!] = \theta \text{ with } \text{arity}(\alpha) = \{1, .., \Delta(\alpha)\} \quad .$$

It is a subsignature of $[\![\text{dom}(\Delta); \Delta]\!]$.

Definition 66. Let L be a finite set. The **semantics of a type** over L is a set that depends on the arity assignment. Let Δ be an arity assignment on L, then

$$\begin{aligned}
[\![\Delta \vdash 0]\!] &= \varnothing \\
[\![\Delta \vdash 1]\!] &= \{*\} \\
[\![\Delta \vdash A + B]\!] &= [\![\Delta \vdash A]\!] + [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \times B]\!] &= [\![\Delta \vdash A]\!] \times [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \to \underline{B}]\!] &= [\![\Delta \vdash A]\!] \to [\![\Delta \vdash \underline{B}]\!] \\
[\![\Delta \vdash B\,!\,\theta]\!] &= [\![\theta; \Delta]\!]\text{-tree}([\![\Delta \vdash B]\!])
\end{aligned}$$

Given Δ, the semantics of a value context $\Gamma = (x : \Gamma(x))_{x \in |\Gamma|}$ is the product of the semantics of types:

$$[\![\Delta \vdash \Gamma]\!] = \prod_{x \in |\Gamma|} [\![\Delta \vdash \Gamma(x)]\!] \ .$$

Proposition 67. Let L be a finite set with subsets $\theta \subseteq \psi \subseteq L$. Let $\Delta : L \to \mathbb{N}$, and let A be a type over L. Then $[\![\Delta \vdash A\,!\,\theta]\!] \subseteq [\![\Delta \vdash A\,!\,\psi]\!]$. Namely, the latter are trees over signature $[\![\psi; \Delta]\!]$, and the former are trees over subsignature $[\![\theta; \Delta]\!]$.

It turns out that semantics of a type does not depend on L, as long as L mentions all the operations in the type, and only depends on the values of Δ for those operations:

Proposition 68. Let L, K be two disjoint finite sets. Let $\Delta : L \to \mathbb{N}$ and $\Delta' : K \to \mathbb{N}$.

- If $\theta \subseteq L$, then $[\![\theta; \Delta,\Delta']\!] = [\![\theta; \Delta]\!]$.
- Let A (resp. $\underline{A}$) be a value (resp. computation) type over L. Then

$$\begin{aligned}
[\![\Delta, \Delta' \vdash A]\!] &= [\![\Delta \vdash A]\!] \\
\text{and} \quad [\![\Delta, \Delta' \vdash \underline{A}]\!] &= [\![\Delta \vdash \underline{A}]\!] \ .
\end{aligned}$$

- Let Γ be a context over L. Then

$$[\![\Delta, \Delta' \vdash \Gamma]\!] = [\![\Delta \vdash \Gamma]\!] \ .$$

The proof is trivial, with the second part by induction on the type.

$$
\begin{array}{lrcl}
\text{computations} & M, N, P & ::= & \text{return } V \mid \alpha\,\vec{M} \mid M \text{ wherealg } \mathcal{A} \\
& & & \mid \text{let } V \text{ be } x.\, M \mid M \text{ to } x.\ N \mid V\,W \\
& & & \mid \text{case } V \text{ of } \{\} \mid \text{case } V \text{ of } \{\langle\rangle.\, M\} \\
& & & \mid \text{case } V \text{ of } \{\text{inl } x.\, M;\ \text{inr } y.\, N\} \mid \text{case } V \text{ of } \{\langle x, y\rangle.\, M\} \\
\text{values} & V, W & ::= & x \mid \lambda x.\, M \mid \langle\rangle \mid \text{inl } V \mid \text{inr } V \mid \langle V, W\rangle \\
\text{syntactic algebras} & \mathcal{A}, \mathcal{B} & ::= & (\alpha\,\vec{k} = N_\alpha)_\alpha
\end{array}
$$

Figure 6.2a: Grammar for fine-grain call-by-value with defined operations. There is a new kind of identifier, called an *operation*. We write abstract operations as $\alpha, \beta, \cdots$. We underline specific ones, e.g: $\underline{\text{either}}$. The let construct is syntactic sugar for other combinations, for instance $\text{return } V \text{ to } x.\, M$. Compared to Figure 1.3 on page 24, we separated $-$ wherealg $(\alpha\,\vec{k} = -)_\alpha$ into an operation definition $-$ wherealg $-$ and a separate syntactic algebra $(\alpha\,\vec{k} = -)_\alpha$. This is to make the technical development clearer.

6.4 Terms

Our terms are given by the grammar and typing rules in Figure 6.2 from page 117. We have three judgements:

$$
\begin{array}{ll}
\text{value} & \Delta; \Gamma \vdash_{\text{v}} V : A \\
\text{computation} & \Delta; \Gamma \vdash_{\text{c}} M : A\,!\,\theta \\
\text{syntactic algebra} & \Delta; \Gamma \vdash_{\text{alg}} \mathcal{A} : \Delta' \Rightarrow A\,!\,\theta
\end{array}
$$

Values and *computations* are like in fine-grain call-by-value ("FGCBV") lambda calculus [48] with binary and nullary sum and product types. Values simply "are" and do not cause side-effects. During evaluations, the variables (from value contexts) are bound to evaluated values.

Computations *can* perform side-effects. The universe of side-effects in this language are *operations*, namely the ones declared in the arity assignment Δ. For example, if we have a binary operation $(\underline{\text{read}} : 2) \in \Delta$, then given two computations M_1, M_2 of the same type, we can construct computation $\underline{\text{read}}(M_1, M_2)$ which perhaps reads user input, and proceeds

Judgements: $\boxed{\Delta;\Gamma \vdash_{\mathrm{v}} V : A}$

$\boxed{\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\theta}$ with $\theta \subseteq \mathrm{dom}(\Delta)$

$\boxed{\Delta;\Gamma \vdash_{\mathrm{alg}} \mathcal{A} : \Delta' \Rightarrow A\,!\,\theta}$ with $\theta \subseteq \mathrm{dom}(\Delta)$ and $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \varnothing$

$$\frac{(x : A) \in \Gamma}{\Delta;\Gamma \vdash_{\mathrm{v}} x : A} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A \qquad \Delta;\Gamma, x{:}A \vdash_{\mathrm{c}} M : B\,!\,\theta}{\Delta;\Gamma \vdash_{\mathrm{c}} \text{let } V \text{ be } x.\, M : B\,!\,\theta}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A}{\Delta;\Gamma \vdash_{\mathrm{c}} \text{return } V : A\,!\,\theta} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\theta \qquad \Delta;\Gamma, x{:}A \vdash_{\mathrm{c}} N : B\,!\,\theta}{\Delta;\Gamma \vdash_{\mathrm{c}} M \text{ to } x.\, N : B\,!\,\theta}$$

$$\frac{(\alpha : n) \in \Delta \qquad \alpha \in \theta \qquad \forall i \in \{1,..,n\} : \Delta;\Gamma \vdash_{\mathrm{c}} M_i : A\,!\,\theta}{\Delta;\Gamma \vdash_{\mathrm{c}} \alpha\,\vec{M} : A\,!\,\theta}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \varnothing \qquad \mathrm{dom}(\Delta) \vdash \Gamma\ \mathsf{ctx} \qquad \mathrm{dom}(\Delta) \vdash A\ \mathsf{type} \qquad \theta \subseteq \mathrm{dom}(\Delta) \\ \forall \alpha \in \mathrm{dom}(\Delta') : \quad \Delta;\ \Gamma, (k_i{:}1 \to A!\theta)_{i \in \{1,..,\Delta'(\alpha)\}} \vdash_{\mathrm{c}} N_\alpha : A\,!\,\theta\end{array}}{\Delta;\Gamma \vdash_{\mathrm{alg}} (\alpha\,\vec{k} = N_\alpha)_{\alpha \in \mathrm{dom}(\Delta')} : \Delta' \Rightarrow A\,!\,\theta}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \varnothing \qquad \mathrm{dom}(\Delta) \vdash \Gamma\ \mathsf{ctx} \qquad \mathrm{dom}(\Delta) \vdash A\ \mathsf{type} \qquad \theta \subseteq \mathrm{dom}(\Delta) \\ \Delta, \Delta';\ \Gamma \vdash_{\mathrm{c}} M : A\,!\,(\theta \cup \mathrm{dom}(\Delta')) \\ \Delta;\Gamma \vdash_{\mathrm{alg}} \mathcal{A} : \Delta' \Rightarrow A\,!\,\theta\end{array}}{\Delta;\Gamma \vdash_{\mathrm{c}} M \text{ wherealg } \mathcal{A} : A\,!\,\theta}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A \to B\,!\,\theta \qquad \Delta;\Gamma \vdash_{\mathrm{v}} W : A}{\Delta;\Gamma \vdash_{\mathrm{c}} V\,W : B\,!\,\theta} \qquad \frac{\Delta;\Gamma, x{:}A \vdash_{\mathrm{c}} M : B\,!\,\theta}{\Delta;\Gamma \vdash_{\mathrm{v}} \lambda x.\, M : A \to B\,!\,\theta}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\theta \qquad A \leq A' \qquad \theta \subseteq \theta'}{\Delta;\Gamma \vdash_{\mathrm{c}} M : A'\,!\,\theta'} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{v}} V : A \qquad A \leq A'}{\Delta;\Gamma \vdash_{\mathrm{v}} V : A'}$$

In the top line of the premises of the rules for operation definition (wherealg) *and syntactic algebra* ($(\alpha\,\vec{k} = N_\alpha)_{\alpha \in K}$)*, we give explicitly some conditions that are needed to make the judgement well-formed. Elsewhere, we leave these conditions implicit.*

Figure 6.2b: Typing rules for fine-grain call-by-value with defined operations, first part.

$$\frac{}{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; \langle\rangle : 1} \qquad \frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : A \qquad \Delta;\Gamma \;\vdash_{\mathrm{v}}\; W : B}{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; \langle V, W\rangle : A \times B}$$

$$\frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : A}{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; \mathrm{inl}\; V : A + B} \qquad \frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; W : B}{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; \mathrm{inr}\; W : A + B}$$

$$\frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : 0}{\Delta;\Gamma \;\vdash_{\mathrm{c}}\; \mathrm{case}\; V \;\mathrm{of}\; \{\} : \underline{C}} \qquad \frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : 1 \qquad \Delta;\Gamma \;\vdash_{\mathrm{c}}\; M : \underline{C}}{\Delta;\Gamma \;\vdash_{\mathrm{c}}\; \mathrm{case}\; V \;\mathrm{of}\; \{\langle\rangle.\, M\} : \underline{C}}$$

$$\frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : A + B \qquad \Delta;\Gamma, x{:}A \;\vdash_{\mathrm{c}}\; M : \underline{C} \qquad \Delta;\Gamma, y{:}B \;\vdash_{\mathrm{c}}\; N : \underline{C}}{\Delta;\Gamma \;\vdash_{\mathrm{c}}\; \mathrm{case}\; V \;\mathrm{of}\; \{\mathrm{inl}\; x.\, M;\; \mathrm{inr}\; y.\, N\} : \underline{C}}$$

$$\frac{\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V : A \times B \qquad \Delta;\Gamma, x{:}A, y{:}B \;\vdash_{\mathrm{c}}\; M : \underline{C}}{\Delta;\Gamma \;\vdash_{\mathrm{c}}\; \mathrm{case}\; V \;\mathrm{of}\; \{\langle x, y\rangle.\, M\} : \underline{C}}$$

Figure 6.2c: Typing rules for fine-grain call-by-value with defined operations, continued.

in either M_1 or M_2 depending on what the user input was. The type of $\underline{\mathrm{read}}(M_1, M_2)$ is the same as the type of M_1 and M_2, so that evaluation preserves type:

$$\frac{(\alpha : n) \in \Delta \qquad \alpha \in \theta \qquad \forall i \in \{1, .., n\} :\; \Delta;\Gamma \;\vdash_{\mathrm{c}}\; M_i : A \,!\, \theta}{\Delta;\Gamma \;\vdash_{\mathrm{c}}\; \alpha\, \vec{M} : A \,!\, \theta}$$

Like in Chapter 5, the type of computations is $A \,!\, \theta$. This type expresses that when the computation may eventually return a value, it will be of type A. Additionally, the type system lets us restrict the operations that are legal to call to a subset $\theta \subseteq \mathrm{dom}(\Delta)$.

It is especially important that the type system lets us specify θ in function types $A \to B \,!\, \theta$, because that allowed us to give the semantics of function types that is stable under weakening (Proposition 68).

The last judgement, *syntactic algebras*, are best seen as a tool towards defining the novel con-

struct of our language, *operation definition*. Let us focus on operation definition now:

$$M \text{ wherealg } \left(\alpha\, \vec{k} = N_\alpha\right)_{\alpha \in \mathrm{dom}(\Delta')}$$

It runs computation M, letting through all operations outside $\mathrm{dom}(\Delta')$, but it does not let the operations in $\mathrm{dom}(\Delta')$ through. If an operation in $\mathrm{dom}(\Delta')$ is executed — say, a binary operation $\underline{\mathrm{read}}$ — then $N_{\underline{\mathrm{read}}}$ is executed instead, which can use functions that will execute the continuation of the left branch (k_1) or the rest of the right branch (k_2). Each k_i represents not merely what is textually in either branch, but also what happens "after the operation", all the way to the end of the $-$ wherealg $(\cdots)$. This is illustrated by the combined typing rule:

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \varnothing \qquad \theta \subseteq \mathrm{dom}(\Delta) \\ \mathrm{dom}(\Delta) \vdash \Gamma \ \mathsf{ctx} \qquad \mathrm{dom}(\Delta) \vdash A \ \mathsf{type} \\ \Delta, \Delta';\ \Gamma \ \vdash_{\mathrm{c}}\ M : A \,!\, (\theta \cup \mathrm{dom}(\Delta')) \\ \forall \alpha \in \mathrm{dom}(\Delta') : \quad \Delta;\ \Gamma, (k_i{:}1 \to A!\theta)_{i \in \{1,..,\Delta'(\alpha)\}} \ \vdash_{\mathrm{c}}\ N_\alpha : A \,!\, \theta\end{array}}{\Delta; \Gamma \ \vdash_{\mathrm{c}}\ M \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha \in \mathrm{dom}(\Delta')} \ :\ A \,!\, \theta}$$

Let us go through it step by step.

- *The conclusion* says that the combined computation may invoke operations $\theta \subseteq \mathrm{dom}(\Delta)$ of the arity specified in Δ. If the return value of type A contains functions, then all arities in Δ may also be relevant, not just the ones of θ.

- *The first two lines* of this rule are sanity checks that are already implied by the well-formedness of the judgements, but we spell them out anyway. For instance, any function types inside the return type A may only invoke operations that are still present in the environment, because only A can only mention operations $\mathrm{dom}(\Delta)$. The first line also reaffirms that we are defining new operation names.

- *The third line* says that M may use all operations in θ, and additionally the operations in Δ'. When M returns a value, then this value is returned from the composite computation verbatim — so the value must be of type A.

- *The fourth line* says what happens instead of passing an operation $\alpha \in \mathrm{dom}(\Delta')$ to the environment: the whole computation is replaced with N_α. In the case of a binary operation $\alpha(-,-)$, N_α additionally has access to

  - $k_1 \;=\; \lambda\langle\rangle.\ \big\langle\text{left continuation}\big\rangle$ wherealg $(\alpha\,\vec{k} = N_\alpha)_{\alpha\in\mathrm{dom}(\Delta')}$
  - $k_2 \;=\; \lambda\langle\rangle.\ \big\langle\text{right continuation}\big\rangle$ wherealg $(\alpha\,\vec{k} = N_\alpha)_{\alpha\in\mathrm{dom}(\Delta')}$

  Note that the $\langle$continuations$\rangle$ are of type $A\,!\,\theta$ — the same type as the whole thing — because it represents the behavior of the final part of M that runs all the way to the end of the operation definition, not merely to the end of the textual $\alpha(M_1, M_2)$.

  Also note that in the continuations, the "extra operations" $\in \mathrm{dom}(\Delta)$ are automatically still defined the same way. In the language of Kammar et al. [40], operation definition is like *deep handlers*.

For ease of definition and analysis, we have split up this typing rule into two on page 118: one for M wherealg $\mathcal{A}$, and one for $\mathcal{A} = (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}$.

Properties

Proposition 69. Weakening in Δ and Γ is admissible. That is, let Δ and Δ' be arity assignments with disjoint underlying sets. Let Γ be a value context over $\mathrm{dom}(\Delta)$, let Γ' be a value context over $\mathrm{dom}(\Delta) \cup \mathrm{dom}(\Delta')$, and let the value contexts be disjoint ($|\Gamma| \cap |\Gamma'| = \varnothing$). Let furthermore A be a type over $\mathrm{dom}(\Delta)$. Then:

1. Let $\Delta;\Gamma \vdash_{\mathrm{v}} V : A$. Then also $\Delta,\Delta';\Gamma,\Gamma' \vdash_{\mathrm{v}} V : A$.
2. Let additionally $\theta \subseteq \mathrm{dom}(\Delta)$, and let $\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\theta$.
   Then also $\Delta,\Delta';\Gamma,\Gamma' \vdash_{\mathrm{c}} M : A\,!\,\theta$.

Definition 70 (substitution)**.** Let $\Delta;\Gamma$ and $\Delta;\Gamma'$ be two typing contexts, and let σ be an $|\Gamma|$-ary family of value terms,

$$\forall x \in |\Gamma| : \quad \Delta;\Gamma' \vdash_{\mathrm{v}} \sigma(x) : \Gamma(x) \quad .$$

Then we call σ a **substitution** from Γ to Γ'. Notation: $\sigma : \Gamma \to \Gamma'$. Its semantics $[\![\sigma]\!]$ is a function from $[\![\Delta \vdash \Gamma']\!]$ to $[\![\Delta \vdash \Gamma]\!]$.

Definition 71. We can form substituted terms as follows:

1. Given a value $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$, we can obtain the substitution $\Delta; \Gamma' \vdash_{\mathrm{v}} V\sigma : A$.

2. Given a computation $\Delta; \Gamma \vdash_{\mathrm{c}} M : \underline{A}$, we can obtain the substitution $\Delta; \Gamma' \vdash_{\mathrm{c}} M\sigma : \underline{A}$.

We can define this substitution as usual: parts 1 and 2 simultaneously, by induction on the term, then $\forall\sigma$. In the λ, let, sequencing, and algebra cases, we must (as expected) weaken Γ and extend σ with the new variable. For $\mathrm{wherealg}$, we must additionally weaken each σ_x to Δ,Δ'.

6.5 Denotational semantics of terms

Definition on derivations

In Figure 6.3 on the next page, we give a denotational semantics of typing derivations. It is a rather conventional monadic semantics. Computations are denoted by trees over the signature corresponding to the arity assignment restricted to θ. Operation definition is denoted by an extended initial algebra morphism (Definition 62). Syntactic algebras are denoted by collecting the denotations of each N_α.

As in previous chapters, we must still show that this gives rise to a semantics of judgements: we must prove that any two derivations of the same judgement have the same semantics.

The logical relation

We straightforwardly adapt the approach for coherence from Chapter 4; for an overview see §4.4. As in §5.5, we employ five kinds of value types: nullary sums (0), binary sums ($A + B$), nullary products (1), binary products ($A \times B$), and function types ($A \to \underline{B}$). Additionally, we have the computation type $B\,!\,\theta$.

The semantics of a:		is a function:
subsumption	$A \leq B$	$[\![\Delta \vdash A \leq B]\!] \;:\; [\![\Delta \vdash A]\!] \;\to\; [\![\Delta \vdash B]\!]$
computation	$\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$	$[\![M]\!] : [\![\Delta \vdash \Gamma]\!] \to [\![\Delta \vdash A \,!\, \theta]\!]$
value	$\Delta; \Gamma \vdash_{\mathrm{v}} V : A$	$[\![V]\!] : [\![\Delta \vdash \Gamma]\!] \to [\![\Delta \vdash A]\!]$
syntactic alg.	$\Delta; \Gamma \vdash_{\mathrm{alg}} \mathcal{A} : \Delta' \Rightarrow A \,!\, \theta$	

$$[\![\mathcal{A}]\!] : [\![\Delta \vdash \Gamma]\!] \times \sum_{\alpha \in \mathrm{dom}(\Delta')} [\![\Delta \vdash A ! \theta]\!]^{\Delta'(\alpha)} \to [\![\Delta \vdash A ! \theta]\!]$$

Write ρ for the argument of type $[\![\Delta \vdash \Gamma]\!]$. The $\mathrm{wherealg}$ case is inductively defined. In its definition, for $[\![M]\!]_\rho = \boldsymbol{\alpha}(\vec{t})$ and $[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]_\rho$, we identify $[\![\Delta \vdash 1 \to A ! \theta]\!] = \{*\} \to [\![\Delta \vdash A ! \theta]\!]$ with $[\![\Delta \vdash A ! \theta]\!]$.

Recall from Proposition 67 that $[\![\Delta \vdash A \,!\, \theta]\!] \subseteq [\![\Delta \vdash A \,!\, (\theta \cup \psi)]\!]$. The definition of semantics of subsumption $[\![\Delta \vdash A \leq B]\!]$ is the obvious one. The semantics of values and computations is as follows:

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\mathrm{return}\ V]\!]_\rho &= \mathrm{leaf}([\![V]\!]_\rho) \\
[\![\alpha\, \vec{M}]\!]_\rho &= \boldsymbol{\alpha}(\overrightarrow{[\![M]\!]_\rho}) \\
[\![\mathrm{let}\ V\ \mathrm{be}\ x.\ M]\!]_\rho &= [\![M]\!]_{\rho, [\![V]\!]_\rho} \\
[\![M\ \mathrm{to}\ x.\ N]\!]_\rho &= (x \mapsto [\![N]\!]_{\rho,x})^* [\![M]\!]_\rho \quad \text{where } -^* \text{ denotes Kleisli extension} \\
[\![\lambda x.\ M]\!]_\rho(v) &= [\![M]\!]_{\rho,v} \\
[\![V\, W]\!]_\rho &= [\![V]\!]_\rho([\![W]\!]_\rho) \\
[\![M\ \mathrm{wherealg}\ \mathcal{A}]\!]_\rho &= ([\![\mathcal{A}]\!]_\rho)^{\mathrm{ext}}([\![M]\!]_\rho)
\end{aligned}$$

The semantics of coercion is given by postcomposition with $[\![\Delta \vdash A \leq B]\!]$ for values, and by applying $[\![\Delta \vdash A \leq B]\!]$ on the leaves for computations; recall also Prop. 67.

Figure 6.3a: Set-valued denotational semantics for fine-grain call-by-value with defined operations, first part. Recall the denotation of types from page 116. Recall the definition of $-^{\mathrm{ext}}$ from Definition 62 on page 114.

Let $K = \mathrm{dom}(\Delta')$.

The semantics of a syntactic algebra $\Delta; \Gamma \ \vdash_{\mathrm{alg}} \ (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \ : \ \Delta' \Rightarrow A \,!\, \theta$:

$$[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]_\rho \left\langle \beta, \vec{\kappa} \right\rangle \ = \ [\![N_\beta]\!]_{\rho, \vec{\kappa}}$$

so that

$$[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]_\rho^{\ \mathrm{ext}} \ : [\![\theta \cup K; \Delta, \Delta']\!]\text{-tree}\Big([\![\Delta \vdash A]\!]\Big) \ \longrightarrow \ [\![\theta; \Delta]\!]\text{-tree}\Big([\![\Delta \vdash A]\!]\Big)$$

or, synonymously:

$$[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]_\rho^{\ \mathrm{ext}} \quad : \qquad [\![\Delta, \Delta' \vdash A \,!\, (\theta \cup K)]\!] \quad \longrightarrow \quad [\![\Delta \vdash A \,!\, \theta]\!]$$

Semantics of values and pattern matching:

$$\begin{aligned}
&[\![\langle\rangle]\!]_\rho && = * \\
&[\![\mathrm{inl}\ V]\!]_\rho && = \mathit{inl}\ [\![V]\!]_\rho \\
&[\![\mathrm{inr}\ V]\!]_\rho && = \mathit{inr}\ [\![V]\!]_\rho \\
&[\![\langle V, W\rangle]\!]_\rho && = \langle [\![V]\!]_\rho, [\![W]\!]_\rho \rangle \\
&[\![\mathrm{case}\ V\ \mathrm{of}\ \{\}]\!]_\rho && = \text{vacuous, as } [\![V]\!]_\rho \in \varnothing \\
&[\![\mathrm{case}\ V\ \mathrm{of}\ \{\langle\rangle.\, M\}]\!]_\rho && = [\![M]\!]_\rho \\
&[\![\mathrm{case}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\}]\!]_\rho && = \begin{cases} [\![M]\!]_{\rho, v} & \text{if } [\![V]\!]_\rho = \mathit{inl}\ v \\ [\![N]\!]_{\rho, w} & \text{if } [\![V]\!]_\rho = \mathit{inr}\ w \end{cases} \\
&[\![\mathrm{case}\ V\ \mathrm{of}\ \{\langle x, y\rangle.\, M\}]\!]_\rho && = [\![M]\!]_{\rho, v, w} \quad \text{if } [\![V]\!]_\rho = \langle v, w\rangle
\end{aligned}$$

Figure 6.3b: Set-valued denotational semantics for fine-grain call-by-value with defined operations, continued.

Let Δ be an arity assignment. In contrast to §5.5, here we can get away with using the same Δ on both sides. We give heterogeneous logical relations

$$\begin{aligned} R[\Delta \vdash A; B] &\subseteq [\![\Delta \vdash A]\!] \times [\![\Delta \vdash B]\!] \\ R[\Delta \vdash \underline{A}; \underline{B}] &\subseteq [\![\Delta \vdash \underline{A}]\!] \times [\![\Delta \vdash \underline{B}]\!] \ . \end{aligned}$$

Namely, we define $R[\Delta \vdash A; B]$ as the complete relation when A, B are value types of different classes; otherwise we define $R[\Delta \vdash A; B]$ and $R[\Delta \vdash \underline{A}; \underline{B}]$ as follows:

$$\begin{array}{lll} & R\big[\Delta \vdash 0;\quad 0\big] = \varnothing & \\ (inl\ v)\ & R\big[\Delta \vdash A{+}B;\ A'{+}B'\big]\ (inl\ w) & \text{iff}\ \ v\, R\big[\Delta \vdash A; A'\big]\, w \\ (inr\ v)\ & R\big[\Delta \vdash A{+}B;\ A'{+}B'\big]\ (inr\ w) & \text{iff}\ \ v\, R\big[\Delta \vdash B; B'\big]\, w \\ \neg\,(v\, & R\big[\Delta \vdash A{+}B;\ A'{+}B'\big]\ w) & \text{otherwise} \\ & \star\, R\big[\Delta \vdash 1;\quad 1\big]\, \star & \\ \langle v_1, v_2\rangle\ & R\big[\Delta \vdash A{\times}B;\ A'{\times}B'\big]\ \langle w_1, w_2\rangle & \text{iff}\ \ v_1\, R\big[\Delta \vdash A; A'\big]\, w_1 \\ & & \qquad \wedge\ \ v_2\, R\big[\Delta \vdash B; B'\big]\, w_2 \\ f\ & R\big[\Delta \vdash A{\to}\underline{B}; A'{\to}\underline{B'}\big]\ g & \text{iff}\ \ \forall (x\, R\big[\Delta \vdash A; A'\big]\, x') : \\ & & \qquad (fx)\, R\big[\Delta \vdash \underline{B}; \underline{B'}\big]\, (gx') \\ c\ & R\big[\Delta \vdash B\,!\,\theta;\ \ B'\,!\,\theta'\big]\ d & \text{iff}\ \ c \text{ and } d \text{ are the same up to leaves,} \\ & & \qquad \text{and the leaves are pairwise } R\big[\Delta \vdash B; B'\big]\ . \end{array}$$

As in Section 5.5, we also define R on environments that have the same set of variables, say X:

$$\begin{aligned} R[\Delta \vdash \Gamma; \Gamma'] &\subseteq \textstyle\prod_{x\in X} [\![\Delta{\vdash}\Gamma(x)]\!] \times \prod_{x\in X} [\![\Delta{\vdash}\Gamma'(x)]\!] \\ \rho\ R[\Delta \vdash \Gamma; \Gamma']\ \rho' &\quad \text{iff} \quad \forall x{\in}X :\ \rho(x)\, R[\Delta \vdash \Gamma(x)\,; \Gamma'(x)]\, \rho'(x) \end{aligned}$$

And we define R on functions from environments to semantic values and computations ("se-

mantic judgements"), as long as the contexts are again on the same set of variables X:

$$R[\Delta;\Gamma \vdash A;\Gamma' \vdash A'] \;\subseteq\; \big((\textstyle\prod_{x\in X}[\![\Delta\vdash\Gamma(x)]\!]\big)\to[\![\Delta\vdash A]\!]\big)\times\big((\textstyle\prod_{x\in X}[\![\Delta\vdash\Gamma'(x)]\!]\big)\to[\![\Delta\vdash A']\!]\big)$$

$$f\; R[\Delta;\Gamma \vdash A;\Gamma' \vdash A']\; g \qquad \text{iff} \quad \forall\rho\, R[\Delta\vdash\Gamma;\Gamma']\,\rho' :\; f(\rho)\, R[\Delta\vdash A;A']\, g(\rho')$$

$$R[\Delta;\Gamma \vdash \underline{A};\Gamma' \vdash \underline{A}'] \;\subseteq\; \big((\textstyle\prod_{x\in X}[\![\Delta\vdash\Gamma(x)]\!]\big)\to[\![\Delta\vdash\underline{A}]\!]\big)\times\big((\textstyle\prod_{x\in X}[\![\Delta\vdash\Gamma'(x)]\!]\big)\to[\![\Delta\vdash\underline{A}']\!]\big)$$

$$c\; R[\Delta;\Gamma \vdash \underline{A};\Gamma' \vdash \underline{A}']\; d \qquad \text{iff} \quad \forall\rho\, R[\Delta\vdash\Gamma;\Gamma']\,\rho' :\; c(\rho)\, R[\Delta\vdash\underline{A};\underline{A}']\, d(\rho')$$

Proposition 72. For an arity assignment Δ and a value type A over $\mathrm{dom}(\Delta)$, we have that $R[\Delta \vdash A; A]$ is the diagonal. For a computation type $\underline{A}$ over $\mathrm{dom}(\Delta)$, $R[\Delta \vdash \underline{A};\underline{A}]$ is the diagonal. Analogously for contexts and judgements.

Proposition 73 (symmetry). Let Δ be an arity assignment.

- For value types A, B over $\mathrm{dom}(\Delta)$, $v\, R[\Delta \vdash A; B]\, w \;\Leftrightarrow\; w\, R[\Delta \vdash B; A]\, v$.
- For computation types $\underline{A}, \underline{B}$ over some finite set $\mathrm{dom}(\Delta)$, $c\, R[\Delta \vdash \underline{A};\underline{B}]\, d \;\Leftrightarrow\; d\, R[\Delta \vdash \underline{B};\underline{A}]\, c$.
- The analogous bi-implications hold for contexts, value judgements, and computation judgements.

Recall that we defined a subtyping in Figure 6.1 on page 112. And recall that in Figure 6.3, we defined for every arity assignment Δ and

for value types A, B over $\mathrm{dom}(\Delta)$ a function $[\![\Delta \vdash A \le B]\!] : [\![\Delta \vdash A]\!] \to [\![\Delta \vdash B]\!]$,

for computation types $\underline{A}, \underline{B}$ over $\mathrm{dom}(\Delta)$ a function $[\![\Delta \vdash \underline{A} \le \underline{B}]\!] : [\![\Delta \vdash \underline{A}]\!] \to [\![\Delta \vdash \underline{B}]\!]$,

and recall that the definitions of these functions are the obvious ones. We leave Δ implicit where they can be deduced, and write just $[\![A \le B]\!]$ (respectively $[\![\underline{A} \le \underline{B}]\!]$).

Compatibility with semantic constructs

Lemma 74 (semantic coercion preserves the relation). Let Δ be an arity assignment.

- Let A, B, C be value types over $\mathrm{dom}(\Delta)$, with $B \le C$. If $v\, R[\Delta \vdash A; B]\, w$, then $v\, R[\Delta \vdash A; C]\, \big([\![B \le C]\!]w\big)$.

- Let $\underline{A}, \underline{B}, \underline{C}$ be computation types over $\mathrm{dom}(\Delta)$, with $\underline{B} \leq \underline{C}$. If $v\, R[\Delta \vdash \underline{A}; \underline{B}]\, w$, then $v\, R[\Delta \vdash \underline{A}; \underline{C}]\, \big([\![\underline{B} \leq \underline{C}]\!] w\big)$.

The proof is trivial.

We find that leaf, $\boldsymbol{\alpha}$, Kleisli extension and extended initial algebra morphism behave well with respect to R:

Lemma 75. Let Δ be an arity assignment. Let A, A' be value types over $\mathrm{dom}(\Delta)$, and let $\theta \subseteq \mathrm{dom}(\Delta)$, $\theta' \subseteq \mathrm{dom}(\Delta)$.

- If $v\, R\big[\Delta \vdash A; A'\big]\, w$, then $\mathrm{leaf}(v)\, R\big[\Delta \vdash A \,!\, \theta; A' \,!\, \theta'\big]\, \mathrm{leaf}(w)$.
- If $\forall i \in \{1, .., \Delta(\alpha)\} :\ m_i\, R\big[\Delta \vdash A \,!\, \theta; A' \,!\, \theta'\big]\, n_i$,
  then $\boldsymbol{\alpha}(\vec{m})\, R\big[\Delta \vdash A \,!\, \theta; A' \,!\, \theta'\big]\, \boldsymbol{\alpha}(\vec{n})$.
- Let additionally B, B' be value types over $\mathrm{dom}(\Delta)$. Let f, g be two functions

$$f \ :\ [\![\Delta \vdash A\,]\!] \longrightarrow [\![\Delta \vdash B\ !\,\theta\,]\!]$$
$$g \ :\ [\![\Delta \vdash A']\!] \longrightarrow [\![\Delta \vdash B' \,!\, \theta']\!]$$

  so that

$$f^* : [\![\Delta \vdash A\ !\,\theta\,]\!] \to [\![\Delta \vdash B\ !\,\theta\,]\!]$$
$$g^* : [\![\Delta \vdash A' \,!\, \theta']\!] \to [\![\Delta \vdash B' \,!\, \theta']\!]\ .$$

  Write $f \times g$ for the function

$$f \times g \ :\ \big([\![\Delta \vdash A]\!] \times [\![\Delta \vdash A']\!]\big) \longrightarrow \big([\![\Delta \vdash B \,!\, \theta]\!] \times [\![\Delta \vdash B' \,!\, \theta']\!]\big)\ .$$

  If the product $f \times g$ preserves R (maps pairs satisfying R to pairs satisfying R), then also the pair of Kleisli extensions $f^* \times g^*$ preserves R.

- Let additionally Δ' be an arity assignment. For its domain, write $K = \mathrm{dom}(\Delta')$; and suppose that the underlying sets are disjoint, $\mathrm{dom}(\Delta) \cap K = \varnothing$. Let f, g be functions

$$f \ :\ F_{[\![K;\Delta']\!]}\big([\![\Delta \vdash A\ !\,\theta\,]\!]\big) \longrightarrow [\![\Delta \vdash A\ !\,\theta\,]\!]$$
$$g \ :\ F_{[\![K;\Delta']\!]}\big([\![\Delta \vdash A' \,!\, \theta']\!]\big) \longrightarrow [\![\Delta \vdash A' \,!\, \theta']\!]$$

so that we have the extended initial algebra morphisms from Definition 62:

$$f^{\text{ext}} : [\![\Delta,\Delta' \vdash A \,!\, (\theta \cup K)\,]\!] \to [\![\Delta \vdash A \,!\, \theta\,]\!]$$
$$g^{\text{ext}} : [\![\Delta,\Delta' \vdash A' \,!\, (\theta \cup K)']\!] \to [\![\Delta \vdash A' \,!\, \theta']\!]\ .$$

Suppose that $f \times g$ preserve R on all operations, that is, on each operation α they map elementwise R-related tuples to R-related trees:

$$\forall(\alpha \in K)\ \forall\left(\vec{m} \in [\![\Delta \vdash A \,!\, \theta]\!]^{\Delta'(\alpha)}\right)\ \forall\left(\vec{n} \in [\![\Delta \vdash A' \,!\, \theta']\!]^{\Delta'(\alpha)}\right) :$$
$$\Big(\vec{m} \text{ and } \vec{n} \text{ pairwise } R\text{-related}\Big) \implies f_\alpha(\vec{m})\, R\, g_\alpha(\vec{n})$$

Then $f^{\text{ext}} \times g^{\text{ext}}$ preserves R.

The proof is again trivial.

Compatibility with syntactic constructs

Recall from Proposition 68 that the semantics of types only depends on the part of Δ that is mentioned in the types.

Proposition 76. Let $\Delta; \Gamma \vdash_{\text{v}} V : A$ with some derivation D, and let D' be a weakening (Prop. 69) formed by adding operations and variables to all levels of the derivation. Then $[\![D']\!]$ is merely $[\![D]\!]$ precomposed with the function that forgets the new variables.

Analogously for computations.

Corollary 77 (weakening in Γ preserves R). Let us have two derivations

$$D :: \Delta; \Gamma \vdash_{\text{v}} V : A$$
$$D' :: \Delta; \Gamma' \vdash_{\text{v}} V : A'$$

with weakenings

$$D_{\text{w}} :: \Delta, \Delta'; \Gamma\,, \Gamma'' \vdash_{\text{v}} V : A$$
$$D'_{\text{w}} :: \Delta, \Delta'; \Gamma', \Gamma''' \vdash_{\text{v}} V : A'$$

and let $[\![D]\!]\, R\big[\Delta; \Gamma \vdash A; \Gamma' \vdash A\big]\, [\![D']\!]$. Then also $[\![D_{\text{w}}]\!]\, R\, [\![D'_{\text{w}}]\!]$.

Analogously for computations.

Like in Chapter 4, given a derivation D, let us write $(A \leq B)\, D$ for the derivation D extended with a coercion from A to B at the bottom. Analogously for computations.

Lemma 78 (coercion preserves R). Let D, D' be two related derivations of the same raw value.

- If $[\![D]\!]\, R\, [\![D']\!]$, then $[\![D]\!]\, R\, [\![(A \leq B)\, D']\!]$.
- If $[\![D]\!]\, R\, [\![D']\!]$, then $[\![(A \leq B)\, D]\!]\, R\, [\![D']\!]$.

Analogously for computations.

Proof. By Lemma 74. □

Lemma 79 (all derivations of a raw term are related). Let Δ be an arity assignment. Let Γ, Γ' be two value contexts over $\mathrm{dom}(\Delta)$ on the same set of variables. Let A and A' be two types over $\mathrm{dom}(\Delta)$.

1. Let D, D' be two derivations of the same raw value,

$$D :: \Delta; \Gamma \vdash_{\mathrm{v}} \mathtt{V} : A$$
$$D' :: \Delta; \Gamma' \vdash_{\mathrm{v}} \mathtt{V} : A' \ .$$

   Then $[\![D]\!]\, R\, [\![D']\!]$.

2. Let θ and θ' be two subsets of $\mathrm{dom}(\Delta)$, and let D, D' be two derivations of the same raw computation,

$$D :: \Delta; \Gamma \vdash_{\mathrm{c}} \mathtt{M} : A \,!\, \theta$$
$$D' :: \Delta; \Gamma' \vdash_{\mathrm{c}} \mathtt{M} : A' \,!\, \theta' \ .$$

   Then $[\![D]\!]\, R\, [\![D']\!]$.

Proof. By induction on the sum of the sizes of the derivations. If either derivation ends in coercion, then apply Lemma 78. Otherwise both end in a syntactic rule. Using Lemma 75, each case follows readily, analogously to the proofs in Section 4.3.

For instance, sequencing goes as follows. Suppose we have two derivations of $\mathtt{M}\ \mathrm{to}\ x.\, \mathtt{N}$. By induction we know that the pairwise subtrees have related semantics, that is:

- for $\mathtt{M}$, we have derivations $D_{\mathtt{M}}, D'_{\mathtt{M}}$ with related semantics;
- for $\mathtt{N}$, we have derivations $D_{\mathtt{N}}, D'_{\mathtt{N}}$ with related semantics.

Write $D_{\mathtt{M}}$ to $x.\, D_{\mathtt{N}}$ (resp. $D'_{\mathtt{M}}$ to $x.\, D'_{\mathtt{N}}$) for the combined derivations. For related environments $\rho\, R\, \rho'$, we must show that

$$[\![D_{\mathtt{M}} \text{ to } x.\, D_{\mathtt{N}}]\!]_\rho \;\; R \;\; [\![D'_{\mathtt{M}} \text{ to } x.\, D'_{\mathtt{N}}]\!]_{\rho'} \;\; .$$

By assumption, we know that $[\![D_{\mathtt{M}}]\!]_\rho \, R \, [\![D'_{\mathtt{M}}]\!]_{\rho'}$. We also know that

$$[\![D_{\mathtt{N}}]\!]_{\rho,-} \times [\![D'_{\mathtt{N}}]\!]_{\rho',-} \quad = \quad (x \mapsto [\![D_{\mathtt{N}}]\!]_{\rho,x}) \times (x \mapsto [\![D'_{\mathtt{N}}]\!]_{\rho',x})$$

preserves R. Using Lemma 75 we find that $(x \mapsto [\![D_{\mathtt{N}}]\!]_{\rho,x})^* \times (x \mapsto [\![D'_{\mathtt{N}}]\!]_{\rho',x})^*$ preserves R. So trivially,

$$(x \mapsto [\![D_{\mathtt{N}}]\!]_{\rho,x})^* \, [\![D_{\mathtt{M}}]\!]_\rho \qquad R \qquad (x \mapsto [\![D'_{\mathtt{N}}]\!]_{\rho',x})^* \, [\![D'_{\mathtt{M}}]\!]_{\rho'}$$

which is what was required. □

All derivations of a judgement have the same semantics:

Corollary 80 (coherence). Let Δ be an arity assumption, let Γ be a value context over $\mathrm{dom}(\Delta)$, and let A be a type over $\mathrm{dom}(\Delta)$.

1. Let D, D' be two derivations of the same judgement,

$$D \;\; :: \Delta; \Gamma \vdash_{\mathrm{v}} \mathtt{V} : A$$

$$D' :: \Delta; \Gamma \vdash_{\mathrm{v}} \mathtt{V} : A \;\; .$$

   Then $[\![D]\!] = [\![D']\!]$.

2. Let θ be a subset of $\mathrm{dom}(\Delta)$, and let D, D' be two derivations of the same judgement,

$$D \;\; :: \Delta; \Gamma \vdash_{\mathrm{c}} \mathtt{M} : A \,!\, \theta$$

$$D' :: \Delta; \Gamma \vdash_{\mathrm{c}} \mathtt{M} : A \,!\, \theta \;\; .$$

   Then $[\![D]\!] = [\![D']\!]$.

This concludes the proof of coherence of our denotational semantics.

Lemma 81 (substitution lemma). Let $\Delta, \Gamma, \Gamma', \sigma$ as in Definition 71.

- For value terms $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$, we have $[\![V\sigma]\!] = [\![V]\!] \circ [\![\sigma]\!]$.
- For computation terms $\Delta; \Gamma \vdash_{\mathrm{c}} M : A$, we have $[\![M\sigma]\!] = [\![M]\!] \circ [\![\sigma]\!]$.

6.6 Base logic

We now give a "base logic" for reasoning about programs, which can be used to prove basic equalities on programs. We have a *value* and a *computation* equality judgement. The value equality judgement looks as follows: if we have two values $\Delta; \Gamma \vdash_{\mathrm{v}} V, W : A$ of the same type, then we may have

$$\Delta; \Gamma \vdash_{\mathrm{v}} V \equiv W : A \quad .$$

The computation equality judgement is analogous: if we have two computations $\Delta; \Gamma \vdash_{\mathrm{v}} M, N : A \,!\, \theta$ of the same type, then we may have

$$\Delta; \Gamma \vdash_{\mathrm{c}} M \equiv N : A \,!\, \theta \quad .$$

The logic is generated by the rules in Figure 6.4 on pages 133–135. Reflexivity follows from the compatibility rules. There are axioms for symmetry and transitivity, so $\equiv$ is an equivalence relation.

Most axioms are just the obvious rules for fine-grain call-by-value. In addition to them, we have

- beta axioms for computing with operation definition,
- the obvious compatibility rule for operation definition
- an axiom that says that algebraic operations are algebraic, that is, they commute with sequencing on the left

- an axiom that says that operation definition commutes with sequencing on the right.

The following rule can be derived, and may be surprising for people who are used to handlers.

Lemma 82. The logic shows $M \equiv M \text{ wherealg } \mathcal{A}$. That is, let Δ, Δ' be two arity assignments with disjoint underlying sets, let Γ a context over $\mathrm{dom}(\Delta)$, A be a type over $\mathrm{dom}(\Delta)$, and $\theta \subseteq \mathrm{dom}(\Delta)$, and let

$$\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$$
$$\Delta; \Gamma \vdash_{\mathrm{alg}} \mathcal{A} : \Delta' \Rightarrow A \,!\, \theta \ .$$

Then the logic shows that $M \equiv M \text{ wherealg } \mathcal{A}$.

Proof.

$$\begin{aligned} M \text{ wherealg } \mathcal{A} &\equiv (M \text{ to } x. \text{return } x) \text{ wherealg } \mathcal{A} \\ &\equiv M \text{ to } x. (\text{return } x \text{ wherealg } \mathcal{A}) \\ &\equiv M \text{ to } x. \text{return } x \\ &\equiv M \end{aligned}$$

□

Lemma 83 (weakening). Let Δ, Δ' be two arity assignments with disjoint underlying sets. Let $\Delta; \Gamma \vdash_{\mathrm{v}} V \equiv W : A$. Let Γ' be a context over $\mathrm{dom}(\Delta) \cup \mathrm{dom}(\Delta')$ with $|\Gamma| \cap |\Gamma'| = \varnothing$. Then also $\Delta, \Delta'; \Gamma, \Gamma' \vdash_{\mathrm{v}} V \equiv W : A$. Analogously for computations.

Lemma 84. Let $\Delta; \Gamma$ and $\Delta; \Gamma'$ be two typing contexts, and let σ, σ' be two substitutions from Γ to Γ' (Def. 70) which are pointwise $\equiv$. And:

1. Let furthermore $\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$. Then $\Delta; \Gamma' \vdash_{\mathrm{c}} M\sigma \equiv M\sigma' : A \,!\, \theta$.

2. Let furthermore $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$. Then $\Delta; \Gamma' \vdash_{\mathrm{v}} V\sigma \equiv V\sigma' : A$.

Proof. By induction on the term. □

$\boxed{\Delta;\Gamma \vdash_{\mathrm{v}} V \equiv W : A}$ $\boxed{\Delta;\Gamma \vdash_{\mathrm{c}} M \equiv N : A\,!\,\theta}$

Beta rules:

- return V to $x.\, M \equiv M[V/x]$
- let V be $x.\, M \equiv M[V/x]$
- $(\lambda x.\, M)\, V \equiv M[V/x]$
- return V wherealg $\mathcal{A} \;\equiv\; \text{return } V$
- If $\gamma \in K = \mathrm{dom}(\Delta')$, then

  $$\beta(M_i)_i \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha\in K} \;\equiv\; N_\beta \left[\big(\lambda\langle\rangle.\, M_i \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha\in K}\big) \,/\, k_i \right]_i$$

- $\beta(M_i)_i$ wherealg $(\alpha\, \vec{k} = N_\alpha)_{\alpha\in K} \;\equiv\; \beta\big(M_i \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha\in K}\big)_i$ — note that this only typechecks when $\beta \notin K$
- case $\langle\rangle$ of $\{\langle\rangle.\, M\} \;\equiv\; M$
- case inl V of $\big\{\text{inl } x.\, M;\ \text{inr } y.\, N\big\} \;\equiv\; M[V/x]$
- case inr V of $\big\{\text{inl } x.\, M;\ \text{inr } y.\, N\big\} \;\equiv\; N[V/y]$
- case $\langle V, W\rangle$ of $\big\{\langle x, y\rangle.\, M\big\} \;\equiv\; M[V/x, W/y]$

Define unused operations:

- M wherealg $\mathcal{A} \;\equiv\; M$ wherealg $(\mathcal{A}, \mathcal{B})$ when $\mathcal{A}$ and $\mathcal{B}$ define a disjoint set of operations

Figure 6.4a: The rules to determine when two terms are $\equiv$, first part. The full judgements are $\Delta;\Gamma \vdash_{\mathrm{v}} V{\equiv}W : A$ and $\Delta;\Gamma \vdash_{\mathrm{c}} M{\equiv}N : A\,!\,\theta$; we leave implicit the premises that both left and right hand sides must individually type check for the same context and type.

Eta rules:

- $M \equiv M \text{ to } x.\ \text{return } x$
- $V \equiv \lambda x.\ (V\ x)$
- $M[V/x] \ \equiv\ \text{case } V \text{ of } \{\}$ when V is of type 0
- $M[V/x] \ \equiv\ \text{case } V \text{ of } \{\langle\rangle.\ M[\langle\rangle/x]\}$ when V is of type 1
- $M[V/x] \ \equiv\ \text{case } V \text{ of } \{\text{inl } y.\ M[\text{inl } y/x];\ \text{inr } y'.\ M[\text{inr } y'/x]\}$ when V is of type $A+B$
- $M[V/x] \ \equiv\ \text{case } V \text{ of } \{\langle y, y'\rangle.\ M[\langle y, y'\rangle/x]\}$ when V is of type $A\times B$

Computations with $\theta = \varnothing$ are thunkable [29]: they have no side-effects and don't diverge:

- $M \text{ to } x.\, \text{return } (\lambda\langle\rangle.\, \text{return } x) \ \equiv\ \text{return } (\lambda\langle\rangle.\, M)$ when M is of type $A\,!\,\varnothing$

Sequencing is associative:

- $(M \text{ to } x.\, N) \text{ to } y.\, P \equiv M \text{ to } x.\, (N \text{ to } y.\, P)$ — note that this only typechecks when x is not mentioned in P

Algebraic operations are algebraic, that is, *operations commute with sequencing on the left:*

- $\alpha(M_i)_i \text{ to } x.\, N \ \equiv\ \alpha(M_i \text{ to } x.\, N)_i$

Operation definition commutes with sequencing on the right:

- $(M \text{ to } x.\, N) \text{ wherealg } \mathcal{A} \quad \equiv \quad M \text{ to } x.\, (N \text{ wherealg } \mathcal{A})$ — note that this only typechecks when M does not use any operations defined by $\mathcal{A}$

Figure 6.4b: The rules to determine when two terms are $\equiv$, continued. As mentioned in Figure 6.4a, both sides must type check the same. Rules such as associativity of sequencing require weakening (recall Prop. 69) which we leave implicit. Additionally, coercions may be inserted in these rules: this is necessary for the rule that commutes operation definition with sequencing. Note that for that rule to type-check, it is required that M does not mention any operation defined in $\mathcal{A}$.

Symmetry:

- If $V \equiv V'$ then $V' \equiv V$
- If $M \equiv M'$ then $M' \equiv M$

Transitivity:

- If $V \equiv V'$ and $V' \equiv V''$ then $V \equiv V''$
- If $M \equiv M'$ and $M' \equiv M''$ then $M \equiv M''$

Compatibility:

- $x \equiv x$
- $\lambda x.\, M \equiv \lambda x.\, M' \quad$ if $\; M \equiv M'$
- $\mathrm{inl}\ V \equiv \mathrm{inl}\ V'$ and $\mathrm{inr}\ V \equiv \mathrm{inr}\ V' \quad$ if $\; V \equiv V'$
- $\langle\rangle \equiv \langle\rangle$ always, and $\quad \langle V, W\rangle \equiv \langle V', W'\rangle \quad$ if $\; V \equiv V'$ and $W \equiv W'$
- $\mathrm{return}\ V \equiv \mathrm{return}\ V' \quad$ if $\; V \equiv V'$
- $\alpha\, \vec{M} \equiv \alpha\, \vec{N} \quad$ if $\; \forall i \in \{1, .., \Delta(\alpha)\} :\; M_i \equiv N_i$
- M wherealg $\left(\alpha\, \vec{k} = N_\alpha\right)_{\alpha \in \mathrm{dom}(\Delta')} \;\equiv\; M'$ wherealg $\left(\alpha\, \vec{k} = N'_\alpha\right)_{\alpha \in \mathrm{dom}(\Delta')}$
  if $\; M \equiv M'$ and $\forall \alpha \in \mathrm{dom}(\Delta') :\; N_\alpha \equiv N'_\alpha$
- $\mathrm{let}\ V\ \mathrm{be}\ x.\, M \equiv \mathrm{let}\ W\ \mathrm{be}\ x.\, N \quad$ if $\; V \equiv W$ and $M \equiv N$
- $M\ \mathrm{to}\ x.\, N \equiv M'\ \mathrm{to}\ x.\, N' \quad$ if $\; M \equiv M'$ and $N \equiv N'$
- $V\, W \equiv V'\, W' \quad$ if $\; V \equiv V'$ and $W \equiv W'$
- $\mathrm{case}\ V\ \mathrm{of}\ \{\} \equiv \mathrm{case}\ V'\ \mathrm{of}\ \{\}$
- $\mathrm{case}\ V\ \mathrm{of}\ \{\langle\rangle.\, M\} \equiv \mathrm{case}\ V'\ \mathrm{of}\ \{\langle\rangle.\, M'\} \quad$ if $\; M \equiv M'$
- $\mathrm{case}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\} \quad$ if $\; V \equiv V'$ and $M \equiv M'$ and $N \equiv N'$
- $\mathrm{case}\ V\ \mathrm{of}\ \{\langle x, y\rangle.\, M\} \equiv \mathrm{case}\ V'\ \mathrm{of}\ \{\langle x, y\rangle.\, M'\} \quad$ if $\; V \equiv V'$ and $M \equiv M'$

Figure 6.4c: The rules to determine when two terms are $\equiv$, last part.

Theorem 85 (soundness)**.**

- If the logic shows $\Delta; \Gamma \vdash_{\mathrm{v}} V \equiv W \ : \ A$, then $[\![V]\!] = [\![W]\!]$.
- If the logic shows $\Delta; \Gamma \vdash_{\mathrm{c}} M \equiv N \ : \ A \,!\, \theta$, then $[\![M]\!] = [\![N]\!]$.

Proof. Straightforward, by induction on the proof of $\equiv$. We highlight an interesting case: for the rule

$$M \text{ to } x.\, \mathrm{return}\ (\lambda\langle\rangle.\, \mathrm{return}\ x) \ \equiv \ \mathrm{return}\ (\lambda\langle\rangle.\, M) \quad \text{when } M \text{ is of type } A \,!\, \varnothing \ ,$$

we note that $[\![\Delta \vdash A \,!\, \varnothing]\!] = \varnothing\text{-tree}([\![\Delta \vdash A]\!]) \cong [\![\Delta \vdash A]\!]$, thus we must have $[\![M]\!]_\rho = \mathit{return}\ v$ for some semantic value v. □

6.7 Operational semantics

Definition and properties

We give the operational semantics in Figure 6.5 on the next page. Its structure is the following: for every arity assignment Δ, and for every computation type $A \,!\, \theta$ over $\mathrm{dom}(\Delta)$, we have a binary relation $\Downarrow$ between computations of type $\Delta; \cdot \vdash_{\mathrm{c}} A \,!\, \theta$.

By induction, we can tell that the right side of $\Downarrow$ is always of the shape $\mathrm{return}\ V$ or $\alpha(\vec{M})$; we call such terms "terminals" and range over them with T. For a terminal T we always have $T \Downarrow T$.

The following is clear from Figure 6.5 alone.

Proposition 86 (determinism). If $M \Downarrow T$ and $M \Downarrow T'$ then $T = T'$, and furthermore the two derivations of $M \Downarrow T$ are the same.

Overview of this section

We will first prove termination in Proposition 92, and then soundness in Proposition 93.

$\boxed{M \Downarrow T}$ with $\Delta; \cdot \vdash_{\mathrm{c}} M, T : A \,!\, \theta$

terminals $T ::= \mathrm{return}\ V \mid \alpha(\vec{M})$

$$\frac{}{\mathrm{return}\ V \Downarrow \mathrm{return}\ V} \qquad \frac{}{\alpha(\vec{M}) \Downarrow \alpha(\vec{M})} \qquad \frac{M[V/x] \Downarrow T}{\mathrm{let}\ V\ \mathrm{be}\ x.\, M \Downarrow T}$$

$$\frac{M \Downarrow \mathrm{return}\ V \qquad N[V/x] \Downarrow T}{M\ \mathrm{to}\ x.\, N \Downarrow T} \qquad \frac{M \Downarrow \alpha(\vec{N})}{M\ \mathrm{to}\ x.\, P \Downarrow \alpha\,(N_i\ \mathrm{to}\ x.\, P)_{i \in \{1,..,\Delta(\alpha)\}}}$$

$$\frac{M[V/x] \Downarrow T}{(\lambda x.\, M)\ V \Downarrow T} \qquad \frac{M \Downarrow \mathrm{return}\ V}{M\ \mathrm{wherealg}\ (\alpha\,\vec{k} = P_\alpha)_{\alpha \in K} \Downarrow \mathrm{return}\ V}$$

$$\frac{M \Downarrow \beta(\vec{N}) \quad \beta \notin K}{M\ \mathrm{wherealg}\ (\alpha\,\vec{k} = P_\alpha)_{\alpha \in K} \Downarrow \beta\left(N_i\ \mathrm{wherealg}\ (\alpha\,\vec{k} = P_\alpha)_{\alpha \in K}\right)_{i \in \{1,..,\Delta(\beta)\}}}$$

$$\frac{M \Downarrow \beta(\vec{N}) \qquad \beta \in K \qquad P_\beta[(\lambda\langle\rangle.N_i\ \mathrm{wherealg}\ (\alpha\,\vec{k} = P_\alpha)_{\alpha \in K})/k_i]_{i \in \{1,..,\Delta(\alpha)\}} \Downarrow T}{M\ \mathrm{wherealg}\ (\alpha\,\vec{k} = P_\alpha)_{\alpha \in K} \Downarrow T}$$

$$\frac{M \Downarrow T}{\mathrm{case}\ \langle\rangle\ \mathrm{of}\ \{\langle\rangle.\, M\} \Downarrow T} \qquad \frac{M[V/x, W/y] \Downarrow T}{\mathrm{case}\ \langle V, W\rangle\ \mathrm{of}\ \{\langle x, y\rangle.\, M\} \Downarrow T}$$

$$\frac{M[V/x] \Downarrow T}{\mathrm{case}\ \mathrm{inl}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\} \Downarrow T} \qquad \frac{N[W/y] \Downarrow T}{\mathrm{case}\ \mathrm{inr}\ V\ \mathrm{of}\ \{\mathrm{inl}\ x.\, M;\ \mathrm{inr}\ y.\, N\} \Downarrow T}$$

Figure 6.5: Operational semantics for fine-grain call-by-value with defined operations.

6.7.1 Termination

Like in Section 3.6, the proof is in Tait-style. In Def. 87 we introduce a family of predicates $P[\Delta; \cdot \vdash_{\mathrm{v}} A], P[\Delta; \cdot \vdash_{\mathrm{c}} A\,!\,\theta]$ on closed terms, stratified by their arity assignment and type. In Def. 88 we extend P to a predicate P^* on all terms, not just closed terms. All terms will turn out to satisfy P^*, thus all closed terms satisfy P, which implies termination (Prop. 92).

Definition 87. We define a predicate P on closed terms (that is, $\Gamma = \varnothing$) as the least fixpoint of the following system of equations. We let Δ over arity assignments, θ over subsets of $\mathrm{dom}(\Delta)$, and A, B over types over $\mathrm{dom}(\Delta)$. The argument of each $P[-]$ ranges over terms with the corresponding judgement.

$$
\begin{aligned}
&P[\Delta; \cdot \vdash_{\mathrm{v}} 0](V) && = \mathrm{true} \quad \text{(vacuously)} \\
&P[\Delta; \cdot \vdash_{\mathrm{v}} 1](\langle\rangle) && = \mathrm{true} \\
&P[\Delta; \cdot \vdash_{\mathrm{v}} A + B](\mathrm{inl}\ V) && = P[\Delta; \cdot \vdash_{\mathrm{v}} A](V) \\
&P[\Delta; \cdot \vdash_{\mathrm{v}} A + B](\mathrm{inr}\ V) && = P[\Delta; \cdot \vdash_{\mathrm{v}} B](V) \\
&P[\Delta; \cdot \vdash_{\mathrm{v}} A \times B](\langle V, W\rangle) && = P[\Delta; \cdot \vdash_{\mathrm{v}} A](V) \wedge P[\Delta; \cdot \vdash_{\mathrm{v}} B](W) \\
&P[\Delta; \cdot \vdash_{\mathrm{v}} A{\to}B!\theta](\lambda x.\ M) && = \forall(\Delta; \cdot \vdash_{\mathrm{v}} V : A) :\ P(V) \Rightarrow P(M[V/x]) \\
&P[\Delta; \cdot \vdash_{\mathrm{c}} B!\theta](M) && \\
&= \exists(\Delta; \cdot \vdash_{\mathrm{v}} V : B) :\ \big(P(V) \ \wedge\ M \Downarrow \mathrm{return}\ V\big) && \\
&\quad \vee\, \exists\, \alpha \in \mathrm{dom}(\Delta),\ \exists\ \Big(\vec{N} : \Delta(\alpha)\text{-ary family of terms } (\Delta; \cdot \vdash_{\mathrm{c}} N_i : B!\theta)_{i \in \{1,..,\Delta(\alpha)\}}\Big) && \\
&\qquad \text{such that } \big(\text{each } N_i \text{ satisfies } P \ \wedge\ M \Downarrow \alpha(\vec{N})\big) &&
\end{aligned}
$$

The equation for $P[\Delta; \cdot \vdash_{\mathrm{c}} B!\theta](M)$ refers to $P[\Delta; \cdot \vdash_{\mathrm{c}} B!\theta](N_i)$, but this system of equations is P is inductive on the type everywhere else, and thus the least fixpoint exists.

Definition 88. We lift P to a predicate P^* on all terms, not just the closed ones.

- For $\Delta; \Gamma \vdash_{\mathrm{v}} V : A$, we say that $P^*(V)$ when for all substitutions $\sigma : \Gamma \to \varnothing$, if each $\sigma(x)$ satisfies P, then $P(V\sigma)$.

- For $\Delta; \Gamma \vdash_{\mathrm{c}} M : A \,!\, \theta$, we say that $P^*(M)$ when for all substitutions $\sigma : \Gamma \to \varnothing$, if each $\sigma(x)$ satisfies P, then $P(M\sigma)$.

Lemma 89. Let

$$\begin{aligned} \Delta; \cdot \quad &\vdash_{\mathrm{c}} M : A \,!\, \theta \quad \text{satisfy } P \text{ and} \\ \Delta; x{:}A &\vdash_{\mathrm{c}} N : B \,!\, \theta \quad \text{satisfy } P^* \end{aligned}$$

Then also $P(M \text{ to } x.\, N)$.

Proof. By induction on a derivation of $P(M)$: recall from Definition 87 that P was defined inductively. (This derivation is unique, but that is not important for the proof.)

- Suppose $P(M)$ because $M \Downarrow \text{return } V$ with $P(V)$. Then $P(N[V/x])$, which implies that $N[V/x] \Downarrow T'$ for some T'. It is clear from the operational semantics that then also $M \text{ to } x.\, N \Downarrow T'$ and furthermore that $P(M \text{ to } x.\, N)$.

- Suppose $P(M)$ because $M \Downarrow \alpha(M'_i)_i$ with each M'_i satisfying P. The individual derivations that show P must be smaller. So by induction, we also know that

  $$P(M'_i \text{ to } x.\, N)$$

  for all i. By definition, $M \text{ to } x.\, N \Downarrow \alpha(M'_i \text{ to } x.\, N)_i$ which completes the proof. □

The following lemma is mostly analogous.

Lemma 90. Let $\Delta; \cdot \vdash M \text{ whereallg } (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} : A$. Suppose M satisfies P and suppose all N_α satisfy P^*. Then $P\big(M \text{ whereallg } (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}\big)$.

Proof. By induction on a derivation of $P(M)$.

- Suppose $P(M)$ because $M \Downarrow \text{return } V$ with $P(V)$. Then $M \text{ whereallg } (\cdots) \Downarrow \text{return } V$.

- Suppose $P(M)$ because $M \Downarrow \beta(M'_i)_i$ with each M'_i satisfying P, and that $\beta \notin K$. Analogous to in the proof of Lemma 89.

- Suppose $P(M)$ because $M \Downarrow \beta(M'_i)_i$ with each M'_i satisfying P, and that $\beta \in K$. The individual derivations that show P must be smaller. So by induction, we also know that

$$P(M'_i \text{ wherealg } (\cdots))$$

for all i. It is also obvious that for all i,

$$P(\lambda\langle\rangle.\, M'_i \text{ wherealg } (\cdots)) \qquad .$$

By assumption, $P^*(N_\beta)$. So therefore

$$P\Big(N_\beta\Big[\big(\lambda\langle\rangle.\, M'_i \text{ wherealg } (\cdots)\big)/k_i\Big]_i\Big)$$

and thus, by definition,

$$P(M \text{ wherealg } (\cdots)) \qquad . \qquad \square$$

Lemma 91. Let $\Delta;\Gamma$ be a typing context.

- Let $\Delta;\Gamma \vdash_{\mathrm{v}} V : A$ be a value. Then $P^*(V)$.
- Let $\Delta;\Gamma \vdash_{\mathrm{c}} M : A!\theta$ be a computation. Then $P^*(M)$.

Proof. By induction on the structure of V or M. Take some $\sigma : \Gamma \to \varnothing$ whose components all satisfy P; we have to prove $P(V\sigma)$ or $P(M\sigma)$, respectively.

- Suppose $V = x$. Then $P(x\sigma) = P(\sigma(x))$ by assumption.
- Suppose $V = \lambda x.M$. By induction, merging two P-satisfying substitutions.
- Suppose $V = \langle\rangle, \text{inl } W, \text{inr } W', \langle W, W'\rangle$. Trivial by induction.
- Suppose $M = \text{return } V$. Trivial by induction.
- Suppose $M = \alpha\,\vec{N}$. Trivial by induction.
- Suppose $M = V\,W$. Observe that $V\sigma$ must be of the shape $V\sigma = \lambda x.N$. By induction, we know $P(V\sigma)$ and thus $P^*(N)$, and by induction $P(W\sigma)$. So $N\sigma[W\sigma/x]$ satisfies P. By definition of $\Downarrow$, it is now obvious that $(VW)\sigma = V\sigma\, W\sigma$ also satisfies P.

- Suppose $M = \text{let } V \text{ be } x.\, N$. By induction we know $P(V\sigma)$ and $P^*(N)$, so $P(N\sigma[V\sigma/x])$. By definition of $\Downarrow$, it is now obvious that $(\text{let } V \text{ be } x.\, N)\sigma = \text{let } V\,\sigma \text{ be } x.\, N\sigma$ also satisfies P.
- Suppose M is a pattern matching. The proof is analogous.
- Suppose $M = N \text{ to } x.\, Q$. Use Lemma 89 on $N\sigma$ and $Q\sigma$.
- Suppose $M = N \text{ wherealg } (\alpha\, k = Q_\alpha)_{\alpha \in K}$. Use Lemma 90 on $N\sigma$ and $(Q_\alpha\sigma)_{\alpha\in K}$.

□

Corollary. All closed terms satisfy P.

Proposition 92 (termination). Let any closed computation $\Delta; \cdot \vdash_{\mathrm{c}} M : A\,!\,\theta$ be given. There exists T such that $M \Downarrow T$.

6.7.2 Soundness

Proposition 93 (soundness). If $M \Downarrow T$, then $[\![M]\!] = [\![T]\!]$.

Proof. By induction on the derivation of $M \Downarrow T$.

- $\text{return } V$ and $\alpha(\vec{M})$ are trivial.
- let, function call, and pattern matching follow from the substitution lemma, Lemma 81 on page 131.
- Sequencing and wherealg are analogous. □

Chapter 7

DEFINED ALGEBRAIC OPERATIONS THAT SATISFY THEORIES

7.1 Introduction

In this chapter, we take the language of Chapter 6 and prove more things about it. Specifically, we show that when you use some syntactic algebra $\mathcal{A}$ to define the operations in a program, then you can use properties about $\mathcal{A}$ when reasoning about that program.

These properties are captured as an equational theory. Let us give an example. Suppose that $\mathcal{A}$ defines a binary operation $\underline{\text{either}}$ in a "commutative" way. We can express that commutativity as a theory:

$$\Theta \quad = \quad \left\{ \; x, y \vdash \underline{\text{either}}(x, y) = \underline{\text{either}}(y, x) \; \right\} \; .$$

Then we have a judgement $\equiv_\Theta$ for programs over this axiom:

$$\begin{aligned} \underline{\text{either}}(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)) &\equiv_\Theta \underline{\text{either}}(\underline{\text{either}}(\text{return } 2, \text{return } 3), \text{return } 1) \\ &\equiv_\Theta \underline{\text{either}}(\underline{\text{either}}(\text{return } 3, \text{return } 2), \text{return } 1) \end{aligned}$$

and a logic $\equiv_\varnothing$ for pure programs with the same semantics, which admits the following rule about this particular $\mathcal{A}$:

$$\frac{M \;\equiv_\Theta\; N}{M \text{ wherealg } \mathcal{A} \;\equiv_\varnothing\; N \text{ wherealg } \mathcal{A}} \tag{7.1}$$

Both logics are congruences. We say that "under $(-\ \text{wherealg}\ \mathcal{A})$, either is commutative": we can apply commutativity of either within $(-\ \text{wherealg}\ \mathcal{A})$ without changing the meaning of the larger program.

Approach

First, in §7.2 we make precise the notion of equational theories on a signature. We recap the category of sets with partial equivalence relations (pers) (§7.3) and algebras and free congruences (§7.4). Then, in §7.5, for each equational theory we augment the set-based semantics from Chapter 6 with a per. It turns out that the semantics of every term — which is a function from environment to a semantic value or tree — preserves that per. In the last section, §7.6, we define what it means for an algebra to validate a theory: what is the precondition for deriving equivalences like in Equation (7.1) above.

We give examples of the resulting logic in §7.6, and further examples in §8.

Partial equivalence relations are a standard tool for analyzing the semantics of languages with equivalences, such as effectful languages. For an application of pers to *read* and *write* effects, see Benton et al. [10].

Size of signatures

The language in Chapter 6 works for arbitrary signatures (as we indicated in its introduction) as does the development in this chapter, except for the definition of closure on page 149 which is only included to build intuitions.

7.2 Signatures and theories

> *In this section, we give the notion of equational theory on a signature $\mathcal{S}$, as a set of axioms over some variables. We call the left and right hand sides of each axiom an* $\mathcal{S}$-term *over that set of variables. The notions of* $\mathcal{S}$-term over a set *and* $\mathcal{S}$-tree over a set *are exactly the same. Nevertheless, we choose to use the word* $\mathcal{S}$-tree

when we talk about semantics, and the word $\mathcal{S}$-term *when we talk about the sides of an equation of the theory.*

7.2.1 Terms

Recall the definition of signature on page 113.

Definition 57. A **signature** $\mathcal{S}$ is a set $|\mathcal{S}|$, whose elements are called "operations", together with an "arity" set $\mathrm{arity}_{\mathcal{S}}(s)$ for each operation $s \in |\mathcal{S}|$. We also call $|\mathcal{S}|$ the **underlying set**.

Fix a signature $\mathcal{S}$.

Definition 94. A syntactic $\boldsymbol{\mathcal{S}}$**-term over** a set of variables $\boldsymbol{I}$ is a well-founded term using operations from $\mathcal{S}$ and over variables $(\mathtt{x}_i)_{i\in I}$. We use the following syntax for $\mathcal{S}$-terms:

- we write $\mathtt{x}_i$ for a variable ($i \in I$);
- we write $s\,(t_j)_{j\in\mathrm{arity}_{\mathcal{S}}(s)}$ for operation $s \in |\mathcal{S}|$ on an $\mathrm{arity}_{\mathcal{S}}(s)$-ary family of terms $\vec{t}$.

The following two definitions make explicit the functorial and monadic structure of the set of terms over I.

Definition 95. Let t be an $\mathcal{S}$-term over I, and let function $\sigma{:}I{\to}J$ be given. Then the **renaming** σt is the $\mathcal{S}$-term over J formed by replacing each $\mathtt{x}_i$ by $\mathtt{x}_{\sigma i}$.

Definition 96. Let t be an $\mathcal{S}$-term over J, and let $\vec{u}$ be a J-ary family of $\mathcal{S}$-terms over I. Then the **substitution** $t[\vec{u}/J]$ is the I-term formed by replacing each $\mathtt{x}_j$ by u_j.

Definition 97. An $\boldsymbol{\mathcal{S}}$**-equation** is a triple (I, t, u), denoted $I \vdash t = u$, where I is a set and t,u are $\mathcal{S}$-terms over I.

7.2.2 Theories and implication

Definition 98. An $\boldsymbol{\mathcal{S}}$**-equational theory** $\boldsymbol{\Theta}$ is a set of $\mathcal{S}$-equations.

Theories do not have to be deductively complete with respect to anything. Instead, we define an "implication" relation $I \vdash_{\mathcal{S},\Theta} t = u$ that is the least congruence on $\mathcal{S}$-terms that is closed under substituted axioms from Θ.

Definition 99. Let $\mathcal{S}$ be a signature and let Θ be an $\mathcal{S}$-equational theory. Let I be a set and let t and u range over $\mathcal{S}$-terms over I. We define a relation $\boldsymbol{I \vdash_{\mathcal{S},\Theta} t = u}$ inductively from the following derivation rules:

1. (substituted axiom) If Θ contains $J \vdash t = u$, and if $\vec{v}$ is a J-ary family of terms over I, then $I \vdash_{\mathcal{S},\Theta} t[\vec{v}/J] = u[\vec{v}/J]$.
2. (reflexivity on variables) For all variables $\mathtt{x}_i$ ($i \in I$), we have $I \vdash_{\mathcal{S},\Theta} \mathtt{x}_i = \mathtt{x}_i$.
3. (symmetry) If $I \vdash_{\mathcal{S},\Theta} t = u$, then $I \vdash_{\mathcal{S},\Theta} u = t$.
4. (transitivity) If $I \vdash_{\mathcal{S},\Theta} t = u$ and $I \vdash_{\mathcal{S},\Theta} u = v$ then $I \vdash_{\mathcal{S},\Theta} t = v$.
5. (compatibility) Let s be an operation $\in |\mathcal{S}|$. Let $\vec{t}$, $\vec{u}$ be two $\mathrm{arity}_{\mathcal{S}}(s)$-ary families of terms over I, and on each component j let us have $I \vdash_{\mathcal{S},\Theta} t_j = u_j$. Then we have $I \vdash_{\mathcal{S},\Theta} s(\vec{t}) = s(\vec{u})$.

Proposition 100. The following properties are admissible:

6. (full reflexivity) For all terms t over I, we have $I \vdash_{\mathcal{S},\Theta} t = t$.
7. (axiom) If Θ contains $I \vdash t = u$, then $I \vdash_{\mathcal{S},\Theta} t = u$.
8. (subst. eqs. in term) Let us have a term t over J, and let $\vec{u}_1$, $\vec{u}_2$ be two J-ary families of terms over I such that at each index j we have $I \vdash_{\mathcal{S},\Theta} u_{1,j} = u_{2,j}$. Then we have $I \vdash_{\mathcal{S},\Theta} t[\vec{u}_1/J] = t[\vec{u}_2/J]$.
9. (subst. terms in eq.) Let us have a pair of terms $J \vdash_{\mathcal{S},\Theta} t_1 = t_2$, and let $\vec{u}$ be a J-ary family of terms over I. Then we have $I \vdash_{\mathcal{S},\Theta} t_1[\vec{u}/J] = t_2[\vec{u}/J]$.
10. (subst. eqs. in eq.) Let us have a pair of terms $J \vdash_{\mathcal{S},\Theta} t_1 = t_2$, and let $\vec{u}_1$, $\vec{u}_2$ be two J-ary families of terms over I such that at each index j we have $I \vdash_{\mathcal{S},\Theta} u_{1,j} = u_{2,j}$. Then $I \vdash_{\mathcal{S},\Theta} t_1[\vec{u}_1/J] = t_2[\vec{u}_2/J]$.

Proof.

- Of (2,5) $\Rightarrow$ (6): do induction on t.
- Of (1) $\Rightarrow$ (7): is immediate by using axiom 1 and choosing $v_i = \mathtt{x}_i$.
- Of (1,5) $\Rightarrow$ (8): do induction on t. For the leaves, use (1); for the nodes, use (5).
- Of (1,4,5,6) $\Rightarrow$ (9): By induction on the proof of $J \vdash_{\mathcal{S},\Theta} t_1 = t_2$.
  1. Suppose that the proof is a substituted axiom

     $$J \;\vdash_{\mathcal{S},\Theta}\; v_1[\vec{w}/K] \qquad = \; v_2[\vec{w}/K]$$

     for some axiom $(K \vdash v_1 = v_2)$ in Θ. Then we must prove that

     $$I \;\vdash_{\mathcal{S},\Theta}\; v_1[\vec{w}/K][\vec{u}/J] \qquad = \; v_2[\vec{w}/K][\vec{u}/J] \quad .$$

     But that's exactly

     $$I \;\vdash_{\mathcal{S},\Theta}\; v_1[w_k[\vec{u}/J] \;/\; \mathtt{x}_k]_{k\in K} \;=\; v_2[w_k[\vec{u}/J] \;/\; \mathtt{x}_k]_{k\in K}$$

     which follows from the substituted axiom rule.
  2. If the proof was reflexivity, then $I \vdash_{\mathcal{S},\Theta} t_1[\vec{u}/J] = t_2[\vec{u}/J]$ also follows by reflexivity.
  3. If the proof was symmetry, then use induction and apply symmetry.
  4. If the proof was transitivity, then use induction and apply transitivity.
  5. If the proof was compatibility, then use induction and apply compatibility.
- Of (4,8,9) $\Rightarrow$ (10): First we obtain $I \vdash_{\mathcal{S},\Theta} t_1[\vec{u}_1/J] = t_1[\vec{u}_2/J]$ using (8). Then we obtain $I \vdash_{\mathcal{S},\Theta} t_1[\vec{u}_2/J] = t_2[\vec{u}_2/J]$ using (9). Apply transitivity. □

The following shows that (3,4,5,6,7,9) is also sufficient, even though (3,4,5,6,7,9) would not make for an obviously universe-invariant definition.

Proposition 101. Properties (7,9) imply (1). And property (6) implies (2).

The proof is trivial.

Recall the definition of subsignature on page 115.

Definition 63. A signature $\mathcal{S}$ is a **subsignature** of another signature $\mathcal{S}'$ (notation: $\boldsymbol{\mathcal{S} \subseteq \mathcal{S}'}$) when $|\mathcal{S}| \subseteq |\mathcal{S}'|$ and the arities of the common operations are the same.

Proposition 102 (theories and subsignatures).

- If $\mathcal{S} \subseteq \mathcal{S}'$, and t is an $\mathcal{S}$-term over some set I, then t is also an $\mathcal{S}'$-term over I.
- If $\mathcal{S} \subseteq \mathcal{S}'$ and $I \vdash t = u$ is an $\mathcal{S}$-equation, then it is also an $\mathcal{S}'$-equation.
- If $\mathcal{S} \subseteq \mathcal{S}'$ and Θ is an $\mathcal{S}$-equational theory, then it is also an $\mathcal{S}'$-equational theory.

7.2.3 Implication of terms on a subsignature

The purpose of this subsection is to show that $I \vdash_{\mathcal{S},\Theta} t = u$ is independent of the value of $\mathcal{S}$, as long as Θ is an $\mathcal{S}$-equational theory. We show this in Proposition 105.

Suppose that t_0 is an $\mathcal{S}$-term over some set I. Then we can convert terms over a supersignature to $\mathcal{S}$-terms. Recall that we write $\mathcal{S} \subseteq \mathcal{S}'$ to indicate that $\mathcal{S}$ is a subsignature of $\mathcal{S}'$.

Definition 103. Suppose that $\mathcal{S} \subseteq \mathcal{S}'$, let I be a set, and let t_0 be an $\mathcal{S}$-term over I. Let u be an $\mathcal{S}'$-term over I. Then $\boldsymbol{u|_{\mathcal{S}}^{t_0}}$ is the $\mathcal{S}$-term over I formed by substituting t_0 for any nodes $s(-)$ for operations $s \notin |\mathcal{S}|$ — that is:

$$\begin{aligned} \mathtt{x}_i|_{\mathcal{S}}^{t_0} &= \mathtt{x}_i \\ s(\vec{v})|_{\mathcal{S}}^{t_0} &= s\left(\overrightarrow{v|_{\mathcal{S}}^{t_0}}\right) && \text{if } s \in |\mathcal{S}| \\ s(\vec{v})|_{\mathcal{S}}^{t_0} &= t_0 && \text{if } s \in |\mathcal{S}'| \setminus |\mathcal{S}| \end{aligned}$$

Lemma 104 (preserved by $\vdash_{-,\Theta}$). Suppose that $\mathcal{S}$, let Θ be an $\mathcal{S}$-equational theory, let I be a set, and let t_0 be an $\mathcal{S}$-term over I. Let $\mathcal{S}' \supseteq \mathcal{S}$ be a supersignature, and let t and u be two $\mathcal{S}'$-terms over I. Then:

$$I \vdash_{\mathcal{S}',\Theta} t = u \implies I \vdash_{\mathcal{S},\Theta} t|_{\mathcal{S}}^{t_0} = u|_{\mathcal{S}}^{t_0}$$

Proof. By induction on the proof of $I \vdash_{\mathcal{S},\Theta} t = u$. Symmetry, transitivity, and reflexivity on variables are trivial. So is compatibility, in both cases. For substituted axiom $(J \vdash w = z) \in \Theta$, note that w and z must be $\mathcal{S}$-terms, so we can apply $-|_{\mathcal{S}}^{t_0}$ pointwise on the substitution. □

Proposition 105 ($\vdash_{\mathcal{S},\Theta}$ independent of $\mathcal{S}$). Let $\mathcal{S}$ be a signature, let Θ be an $\mathcal{S}$-equational theory, let I be a set, and let t, u be $\mathcal{S}$-terms over I. Let $\mathcal{S}' \supseteq \mathcal{S}$ be a supersignature. Then:

$$I \vdash_{\mathcal{S}',\Theta} t = u \quad \Longleftrightarrow \quad I \vdash_{\mathcal{S},\Theta} t = u$$

Proof. The $\Leftarrow$ direction is trivial. The $\Rightarrow$ direction goes as follows. $-|_{\mathcal{S}}^{t}$ is the identity on $\mathcal{S}$-terms over I, in particular, $t|_{\mathcal{S}}^{t} = t$ and $u|_{\mathcal{S}}^{t} = u$. By Lemma 104. □

We shall henceforth leave the signature subscript implicit, and write merely $\vdash_{\Theta}$.

7.2.4 Closure and restriction of theories

The development in this subsection is not used in what follows, but may be insightful nonetheless.

Proposition 106 (weakening, antiweakening). Let $\mathcal{S}$ be a signature, let Θ be an $\mathcal{S}$-equational theory, let $I \subseteq J$ be two sets, and let t and u be two $\mathcal{S}$-terms over I. Then:

$$I \vdash_{\Theta} t = u \quad \Longleftrightarrow \quad J \vdash_{\Theta} t = u \quad .$$

Proof. The $\Rightarrow$ direction follows readily by property (subst. terms in eqs.) from Proposition 100, substituting $\mathtt{x}_i$ for $\mathtt{x}_i$. The $\Leftarrow$ direction also follows by (subst. terms in eqs.), by substituting t for variables $(\mathtt{x}_j)_{j \in J \setminus I}$. □

Definition 107. Let $\mathcal{S}$ be a signature with only countable operations, and let Θ be an $\mathcal{S}$-equational theory. The **$\mathcal{S}$-substitution closure of Θ**, denoted $\Theta^{\dagger}$, is the $\mathcal{S}$-equational theory formed as the set of all "implied" triples

$$I \vdash t = u$$

where $I \subseteq \mathbb{N}$, t and u are $\mathcal{S}$-terms over I, and $I \vdash_{\Theta} t = u$.

We will leave out the $\mathcal{S}$ subscript where it can be deduced from context.

Proposition 108. Let $\mathcal{S}$ be a signature with only operations of finite (resp. countable) arity. Let t be an $\mathcal{S}$-term over some set I. Then only finitely (resp. countably) many elements of I occur in t as variables $\mathtt{x}_i$.

Proposition 109. Let $\mathcal{S}$ be a signature with only countable operations, and let Θ be an $\mathcal{S}$-equational theory. Let I be a set and let t, u be two $\mathcal{S}$-terms over I. Then

$$I \vdash_\Theta t = u \quad \iff \quad I \vdash_{\Theta^\dagger} t = u \ .$$

Proof. For the $\Leftarrow$ direction, by induction on the proof. For a base case $(J \vdash v = w) \in \Theta^\dagger$ apply (subst. terms in eq.) from Proposition 100. Symmetry, transitivity, reflexivity on variables, and compatibility are trivial.

For $\Rightarrow$, let I' be the subset of I consisting of only the variables mentioned in t or u. It must be countable. By weakening, it suffices to prove that $I' \vdash_{\Theta^\dagger} t = u$.

For any numbering of I' — an injective partial function ι from $\mathbb{N}$ to I' — there must correspondingly be an axiom $(\mathrm{dom}(\iota) \vdash t[\mathtt{x}_{\iota^{-1}(i)}/\mathtt{x}_i]_{i\in I'} = u[\mathtt{x}_{\iota^{-1}(i)}/\mathtt{x}_i]_{i\in I'})$ in $\Theta^\dagger$. Take one such renumbering and the corresponding axiom $\in \Theta^\dagger$. We construct a proof of $I \vdash_{\Theta^\dagger} t = u$ from this axiom and the substitution $[\mathtt{x}_{\iota(n)}/\mathtt{x}_n]_{n\in\mathrm{dom}(\iota)}$. □

7.2.5 Restriction of theories

Definition 110. Let $\mathcal{S} \subseteq \mathcal{S}'$, and let Θ be a $\mathcal{S}'$-equational theory. Then **Θ restricted to $\mathcal{S}$**, notation $\Theta|_{\mathcal{S}}$, is a subset of Θ, namely the theory consisting of only the axioms where both sides are $\mathcal{S}$-terms. It is an $\mathcal{S}$-equational theory.

Proposition 111. Let $\mathcal{S}'$ be a signature with only countable operations, let Θ be an $\mathcal{S}'$-equational theory, and let $\mathcal{S} \subseteq \mathcal{S}'$ be a subsignature. Let I be a countable set, and let t and u be two $\mathcal{S}$-terms over I. Then

$$I \vdash_\Theta t = u \quad \iff \quad I \vdash_{\Theta^\dagger|_{\mathcal{S}}} t = u \ .$$

Proof. See the proof of Proposition 109. In the $\Rightarrow$ direction, it constructs a proof that is also a proof for $I \vdash_{\Theta^\dagger|_S} t = u$. The $\Leftarrow$ direction follows trivially from the proof of Proposition 109. □

7.3 Preliminary: The category Per of sets with a partial equivalence relation

We recall partial equivalence relations, which form the basis of the denotational semantics in this chapter.

Definition 112. A **partial equivalence relation** or **per** on a set X is a symmetric transitive endorelation on X.

Given a function $f : X \to Y$, a partial equivalence relation $\sim$ on X and a partial equivalence relation $\approx$ on Y, we say that f **preserves the partial equivalence relation** when for each $X \sim Y$ we have that $fx \approx fy$.

Proposition 113. Let $(X, \sim)$ be a set with a per, and let $x, y \in X$. If $x \sim y$, then by symmetry and transitivity, $x \sim x$.

Definition 114. Every set with a per $(X, \sim)$ has a domain $\mathrm{dom}(X, \sim)$ of elements that are related to themselves. On those elements, $\sim$ is an equivalence relation.

Proposition 115. Let X be a set. Every pair $(Y, \equiv)$ of a subset $Y \subseteq X$ with an equivalence relation $\equiv$ on Y can be reinterpreted as coming from Definition 114: $\equiv$ is a per on X and its domain is Y.

Definition 116. We write Per for the category of sets with pers. The category is concrete, and it is per-enriched in the following way. Let $f, g : (X, \sim) \to (Y, \approx)$. Then f is related to g if $\forall x \in \mathrm{dom}(X, \sim),\ fx \approx gx$.

Proposition 117. Category Per has all products and coproducts, and it is bicartesian closed. The underlying set of all these constructs agrees with the category Set. The binary product of

sets with pers $\mathbb{A} = (X, \sim), \mathbb{B} = (Y, \approx)$ is

$$\mathbb{A} \times \mathbb{B} = (X \times Y, \text{both components simultaneously related}) \ .$$

The coproduct relates only $\mathrm{dom}(X)$ to $\mathrm{dom}(X)$ and $\mathrm{dom}(Y)$ to $\mathrm{dom}(Y)$. Arbitrary products and coproducts generalise this. In particular, 1 uses the per that relates the element to itself. The exponential $\mathbb{B}^{\mathbb{A}}$ is:

$$\mathbb{B}^{\mathbb{A}} = \left(Y^X, \ \{(f,g) \mid \forall (x \sim y) : fx \approx gx\}\right) .$$

Proof for exponentials. Let $\mathbb{C} = (Z, W, \lessapprox)$. We use the obvious bijection $-^\dagger$ from functions $f : Z \times X \to Y$ to functions $f^\dagger : Z \to Y^X$, and the evaluation map is defined as on Set. We trivially have $f = ev \circ (f^\dagger \times \mathrm{id}_A)$. It remains to show that $-^\dagger$ is a bijection between *morphisms*. Indeed, f resp. $f^\dagger$ are both morphisms that map all related Z-pairs and all related X-pairs to related Y-pairs. □

Lemma 118 (characterisation of morphisms out of products). Let $(X, \sim)$, $(Y, \sim')$, $(Z, \sim'')$ be sets with partial equivalence relations. And let f be a function $f \ : \ X \times Y \longrightarrow Z$. Then f is a Per-morphism

$$f \ : \ (X, \sim) \ \times \ (Y, \sim') \ \longrightarrow \ (Z, \sim'')$$

if and only if three conditions hold:

1. f restricts to a function $f : \mathrm{dom}(\sim) \times \mathrm{dom}(\sim') \longrightarrow \mathrm{dom}(\sim'')$, and
2. f is a Per-morphism $f \ : \ (X, \sim) \times (\mathrm{dom}(\sim'), =_{\mathrm{dom}(\sim')}) \longrightarrow (\mathrm{dom}(\sim''), =_{\mathrm{dom}(\sim'')})$, and
3. f is a Per-morphism $f : (\mathrm{dom}(\sim), =_{\mathrm{dom}(\sim)}) \times (Y, \sim') \longrightarrow (\mathrm{dom}(\sim''), =_{\mathrm{dom}(\sim'')})$.

Lemma 118 is used in Section 7.5, where we prove that $[\![(\alpha \, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\mathrm{ext}}$ preserves the partial equivalence relations in both arguments. This proof is introduced on page 167 and concluded on page 176. There is a natural generalisation of Lemma 118 to finite products, but we will only need the version for binary products.

7.4 Algebras and free congruences

We recall some standard results about $\mathcal{S}$-algebras, and we look at certain congruences over $\mathcal{S}$-algebras that satisfy an $\mathcal{S}$-equational theory.

7.4.1 Algebras on a set

Definition 119. A **$\mathcal{S}$-algebra** (A, a) is a set A together with a function $a : F_{\mathcal{S}}(A) \to A$. More explicitly, it is A together with for every operation $s \in |\mathcal{S}|$ a function

$$a_s \;:\; A^{\mathrm{arity}_{\mathcal{S}}(s)} \longrightarrow A \;\;.$$

Definition 120. Let t be a term over some set I. The **interpretation of term t in (A, a)** is the $\mathcal{S}$-morphism $[\![t]\!]_{(A,a)}$,

$$[\![t]\!]_{(A,a)} \;:\; A^I \longrightarrow A \quad ,$$

$$\begin{aligned} [\![\mathtt{x}_i]\!]_{(A,a)}(\vec{\rho}) &= \rho_i \\ [\![s(\vec{t})]\!]_{(A,a)}(\vec{\rho}) &= a_s\Big(\langle \cdots,\ [\![t_i]\!]_{(A,a)}(\vec{\rho}),\ \cdots\rangle\Big) \qquad . \end{aligned}$$

Lemma 121 (substitution). Let t be a term over I, and let $\vec{u}$ be an I-ary family of terms over J. Recall that $t[\vec{u}/I]$ is then a term over J. The interpretations are related as follows. Let $\vec{\rho}$ be a J-ary family of elements from A, then:

$$[\![t[\vec{u}/I]]\!]_{(A,a)}(\vec{\rho}) = [\![t]\!]_{(A,a)}\Big([\![u_i]\!]_{(A,a)}(\vec{\rho})\Big)_{i\in I}$$

Proof. Recall (page 144) that $\mathcal{S}$-terms over variables I (such as t) are merely elements of $\mathcal{S}\text{-tree}(I)$, and $\mathcal{S}\text{-tree}(-)$ is the free monad over $F_{\mathcal{S}}$. We can represent $\vec{u}$ as a function $\vec{u}$: $I \to \mathcal{S}\text{-tree}(J)$, and $\vec{\rho}$ as a function $\vec{\rho}\colon J \to \mathcal{S}\text{-tree}(A)$. The left and right hand sides are merely Kleisli compositions

$$1 \xrightarrow{t} \mathcal{S}\text{-tree}(I) \xrightarrow{\vec{u}^*} \mathcal{S}\text{-tree}(J) \xrightarrow{\vec{\rho}^*} \mathcal{S}\text{-tree}(A)$$

with different bracketings: $\vec{\rho}^* \circ (\vec{u}^* \circ t) = (\vec{\rho}^* \circ \vec{u})^* \circ t$. It is a standard fact of Kleisli extensions that $(\vec{\rho}^* \circ \vec{u})^* = \vec{\rho}^* \circ \vec{u}^*$, so clearly the left and right hand sides are equal. □

The proof can also be done simply by induction on t.

When $\mathcal{S} \subseteq \mathcal{S}'$, then $\mathcal{S}'\text{-tree}(X)$ together with the obvious function is an $\mathcal{S}$-algebra. We write just $\mathcal{S}'\text{-tree}(X)$ for this algebra.

Lemma 122. Let

- $\mathcal{S}, \mathcal{S}', \mathcal{S}''$ be signatures such that $\mathcal{S} \subseteq \mathcal{S}'$ and $\mathcal{S} \subseteq \mathcal{S}''$,
- X, Y be two sets
- $f : \mathcal{S}'\text{-tree}(X) \longrightarrow \mathcal{S}''\text{-tree}(Y)$ preserve the $\mathcal{S}$-algebra (recall Def. 64), and
- t be an $\mathcal{S}$-term over I.

Then f commutes with the interpretation of t in the sense that for all $\vec{\rho} : I \to \mathcal{S}'\text{-tree}(X)$,

$$f\left(\llbracket t \rrbracket_{\mathcal{S}'\text{-tree}(X)}(\vec{\rho})\right) \quad = \quad \llbracket t \rrbracket_{\mathcal{S}''\text{-tree}(Y)}\Big(f(\rho_i)\Big)_{i \in I} \quad .$$

The proof is by induction on t.

We recall that the Kleisli extension for $\mathcal{S}\text{-tree}(-)$ is as follows.

Proposition 123. Let (A, a) be an $\mathcal{S}$-algebra, let X be a set, and let f be a function $f \ : \ X \longrightarrow A$. Then the **Kleisli extension of f for (A, a)** (notation: f^*) is the function

$$\begin{aligned} & f^* \ : \ \mathcal{S}\text{-tree}(X) \longrightarrow A \\ & f^*(\text{leaf}(x)) \qquad\qquad = f(x) \\ & f^*\Big(s\,(x_i)_{i \in \text{arity}_{\mathcal{S}}(s)}\Big) = a_s\,\Big(f^*(x_i)\Big)_{i \in \text{arity}_{\mathcal{S}}(s)} \quad . \end{aligned}$$

Proposition 124. f^* preserves the $\mathcal{S}$-algebra.

The following is a trivial result (or axiom) about Kleisli extensions.

Proposition 125. Let t be an $\mathcal{S}$-term over some set I, and let $\vec{\rho}$ be an I-collection of elements of X. Then

$$f^*\Big(\llbracket t \rrbracket_{\mathcal{S}\text{-tree}(X)}\big(\text{leaf}(\rho_i)\big)_{i \in I}\Big) \quad = \quad \llbracket t \rrbracket_{(A,a)}\big(f(\rho_i)\big)_{i \in I} \quad .$$

Proof. We represent t as an element of $\mathcal{S}\text{-tree}(I)$, and $\vec{\rho}$ as a function $I \to X$. Then the following compositions are equal:

$$1 \xrightarrow{t} \mathcal{S}\text{-tree}(I) \xrightarrow{(\text{leaf} \circ \vec{\rho})^*} \mathcal{S}\text{-tree}(X) \xrightarrow{f^*} A \quad = \quad 1 \xrightarrow{t} \mathcal{S}\text{-tree}(I) \xrightarrow{(f \circ \vec{\rho})^*} A \qquad \square$$

7.4.2 Congruences

In order to interpret equations, we first need a congruence on the algebra:

Definition 126. An **$\mathcal{S}$-congruence** $\simeq$ on an $\mathcal{S}$-algebra (A, a) is a per on A that is preserved by semantic operations in the following way: If $s \in |\mathcal{S}|$ and $\vec{x}, \vec{y}$ are two $\text{arity}_{\mathcal{S}}(s)$-ary family of elements of A, pairwise related, then $a_s(\vec{x}) \simeq a_s(\vec{y})$.

Term evaluation on congruences is monotonous in its environment argument:

Lemma 127. Let t be an $\mathcal{S}$-term over some set I, and let $\vec{\rho}$, $\vec{\sigma}$ be two I-ary families of elements of A, pairwise $\simeq$. Then

$$[\![t]\!]_{(A,a)}(\vec{\rho}) \;\simeq\; [\![t]\!]_{(A,a)}(\vec{\sigma}) \;\; .$$

The proof is by induction on t.

Definition 128. We say that an $\mathcal{S}$-congruence $\simeq$ on (A, a) **satisfies** equation $I \vdash t = u$ when for I-ary families $\vec{\rho}$ of elements from A, $[\![t]\!]_{(A,a)}(\vec{\rho}) \simeq [\![u]\!]_{(A,a)}(\vec{\rho})$. We say that $\simeq$ **satisfies** $\mathcal{S}$-equational theory Θ when it satisfies all equations in Θ.

Term evaluation on a congruence that satisfies a theory preserves the per on terms implied by that theory:

Lemma 129 (satisfaction of implied equations). Let $\simeq$ be a $\mathcal{S}$-congruence on an $\mathcal{S}$-algebra that satisfies some $\mathcal{S}$-equational theory Θ, which in turn implies some equation $I \vdash_\Theta t_1 = t_2$. And suppose that $\vec{\rho}$ is an I-ary family of elements $\in A$. Then $[\![t_1]\!]_{(A,a)}(\vec{\rho}) \;\simeq\; [\![t_2]\!]_{(A,a)}(\vec{\rho})$.

Proof. By induction on the proof of $I \vdash_\Theta t_1 = t_2$.

- Suppose it's a substituted axiom $I \vdash_\Theta u_1[\vec{v}/J] = u_2[\vec{v}/J]$ for some axiom $J \vdash_\Theta u_1 = u_2$ and some J-ary family $\vec{v}$ of terms over I. Then

$$\begin{aligned}
[\![u_1[\vec{v}/J]]\!]_{(A,a)}(\vec{\rho}) &= [\![u_1]\!]_{(A,a)}\Big([\![v_j]\!]_{(A,a)}(\vec{\rho})\Big)_{j\in J} && \text{by Lemma 121} \\
&\simeq [\![u_2]\!]_{(A,a)}\Big([\![v_j]\!]_{(A,a)}(\vec{\rho})\Big)_{j\in J} && \text{because } \simeq \text{ satisfies } \Theta \\
&= [\![u_2[\vec{v}/J]]\!]_{(A,a)}(\vec{\rho}) && \text{by Lemma 121}
\end{aligned}$$

- Suppose it's reflexivity, then trivial.
- Suppose it's symmetry, then apply induction and use symmetry of $\simeq$.
- Suppose it's transitivity, then apply induction and use transitivity of $\simeq$.
- Otherwise it's compatibility, then apply induction and use the fact that $\simeq$ is a congruence.

□

7.4.3 Free Θ-congruences

In this section, we will concern ourselves with least congruences that satisfy an equational theory.

We define three notions of free Θ-congruence:

1. A notion of "free Θ-congruence" on trees with leaves valued in a set,
2. a notion of "free Θ-congruence" on trees with leaves valued in a setoid (= set with an equivalence relation),
3. a notion of "free Θ-congruence" on trees with leaves valued in a per.

On trees with leaves valued in a set

Definition 130. Let us have a signature $\mathcal{S}$, a set X, and an equational $\mathcal{S}$-theory Θ. The **free Θ-congruence $\sim_\Theta$** is the least endorelation on $\mathcal{S}\text{-tree}(X)$ that

1. is reflexive,

2. is symmetric,

3. is transitive,

4. is preserved by semantic operations in the sense that

$$\forall s{\in}|\mathcal{S}|\ \forall \vec{x},\vec{y} \in \mathcal{S}\text{-tree}(X)^{\text{arity}_{\mathcal{S}}(s)} :\quad (\forall i : x_i \sim_\Theta y_i) \Rightarrow \boldsymbol{s}(\vec{x}) \sim_\Theta \boldsymbol{s}(\vec{y}) \quad \text{, and}$$

5. is closed under substitutions of axioms in Θ in the sense that for each axiom $(I \vdash t = u)$ in Θ, for all I-ary families $\vec{\rho}$ of elements of $\mathcal{S}\text{-tree}(X)$, we have $[\![t]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho}) \sim_\Theta [\![u]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho})$.

We can construct $\sim_\Theta$ as a least fixed point. By definition, it is an $\mathcal{S}$-congruence (1,2,3) which satisfies Θ (4).

Recall the Kleisli extension from Proposition 123.

Lemma 131. Let us have a signature $\mathcal{S}$, a set X, an $\mathcal{S}$-algebra (A, a), and an $\mathcal{S}$-congruence $\simeq$ on (A, a) that satisfies some $\mathcal{S}$-equational theory Θ. Let us have a function $f : X \to A$. Then $f^* : \mathcal{S}\text{-tree}(X) \to A$ preserves the free congruence $\sim_\Theta$ into $\simeq$. That is:

$$\text{if } x \sim_\Theta y \quad \text{then } f^*(x) \simeq f^*(y) \quad .$$

Proof. We induct on the proof of $x \sim_\Theta y$; recall its definition from Def. 130.

1. (*Reflexivity*.) Trivially $f^*(x) \simeq f^*(x)$.

2. (*Symmetry*.) If $x \sim_\Theta y$ because $y \sim_\Theta x$, then by induction $f^*(y) \simeq f^*(x)$, and thus $f^*(x) \simeq f^*(y)$ because $\simeq$ is an equivalence relation.

3. (*Transitivity*.) If $x \sim_\Theta y$ because $x \sim_\Theta z$ and $z \sim_\Theta y$, then by induction $f^*(x) \simeq f^*(z)$ and $f^*(z) \simeq f^*(y)$, therefore $f^*(x) \simeq f^*(y)$.

4. (*Compatibility*.) Let $s \in |\mathcal{S}|$ and let $\vec{x},\vec{y}$ be two $\text{arity}_{\mathcal{S}}(s)$-ary families of elements from X, such that they are componentwise $\sim_\Theta$. By induction, assume that on each component i we have also $f^*(x_i) \simeq f^*(y_i)$. Because $\simeq$ is a congruence, we know that $a_s(f^*(x_i)_{i\in\text{arity}_{\mathcal{S}}(s)}) \simeq a_s(f^*(y_i)_{i\in\text{arity}_{\mathcal{S}}(s)})$ which was to be proven.

5. (*Substituted axiom.*) Suppose that the premise is $[\![t]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho}) \sim_\Theta [\![u]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho})$ for some axiom $(I \vdash t{=}u) \in \Theta$. Because $\simeq$ satisfies Θ, we know that for any A-valued family $\vec{\sigma}$, $[\![t]\!]_{(A,a)}(\vec{\sigma}) \simeq [\![u]\!]_{(A,a)}(\vec{\sigma})$. So then

$$f^*([\![t]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho})) \;=\; [\![t]\!]_{(A,a)}\Big(f(\rho_i)\Big)_{i\in I} \simeq [\![u]\!]_{(A,a)}\Big(f(\rho_i)\Big)_{i\in I} \;=\; f^*([\![u]\!]_{\mathcal{S}\text{-tree}(X)}(\vec{\rho})) \quad .$$

□

Relevance to our semantics. In the denotational semantics for our language, computations are represented by trees. The axioms in Θ are used to equate computations that should be equated according to Θ. Two semantic computations might be equated if one is "essentially the same" according to some interpretation of the operations.

On trees with leaves valued in a setoid (= set with an equivalence relation)

If there is already an equivalence relation on X, say $\approx$, then we define a slightly stronger "free" Θ-congruence. Fix a signature $\mathcal{S}$ and a $\mathcal{S}$-equational theory Θ.

Definition 132. The **free Θ-congruence $\sim_{\Theta,\approx}$** is the least endorelation on $\mathcal{S}\text{-tree}(X)$ that

1. contains $\approx$ on the leaves, that is, whenever two elements x, y of X are ordered $x \approx y$ then $\text{leaf}(x) \sim_{\Theta,\approx} \text{leaf}(y)$,
2. is reflexive,
3. is symmetric,
4. is transitive,
5. is preserved by semantic operations (as in Def. 130), and
6. is closed under substitutions of axioms in Θ (as in Def. 130).

The Kleisli extension is again well-behaved with respect to $\sim_{\Theta,\approx}$:

Lemma 133. Let $(X, \approx)$ be a set with an equivalence relation, let $\simeq$ be an $\mathcal{S}$-congruence on (A, a) that satisfies Θ, and suppose that f preserves $\approx$ into $\simeq$, that is,

$$\forall x, y \in X : \quad x \approx y \implies f(x) \simeq f(y) \quad ,$$

then f^* preserves the free congruence $\sim_{\Theta,\approx}$ into $\simeq$. That is:

$$\forall t, u \in \mathcal{S}\text{-tree}(X) : \quad t \sim_{\Theta,\approx} u \implies f^*(t) \simeq f^*(u) \quad .$$

Proof. Like the proof of Lemma 131; there is just one extra case to prove which we give now. (*Relation on the leaves*.) Suppose that $x \approx y$ for two elements x,$y \in X$. Then we must prove that $f^*(\text{leaf}(x)) \simeq f^*(\text{leaf}(y))$. The left and right hand sides of this equation are equal to $f(x)$ and $f(y)$, respectively, so this is just the assumption. □

Relevance to our semantics. Tree leaves correspond to results of a computation. If the result type of the computation is a ground type, then the per on that type will be discrete. If the results are functions, then two functions may be related if for all inputs the functions produce related results.

On trees with leaves valued in a subset

Definition 134. Let (A, a) be an $\mathcal{S}$-algebra. An ($\mathcal{S}$-)**subalgebra** is a subset $V \subseteq A$ that is preserved by all operations:

$$\forall s \in |\mathcal{S}| \; \forall \vec{x} \in V^{\text{arity}_{\mathcal{S}}(s)} : \; a_s(\vec{x}) \in V \; .$$

Any subset $U \subseteq X$ induces a subalgebra on trees:

Proposition 135. If $U \subseteq X$ then $\mathcal{S}\text{-tree}(U)$ is an $\mathcal{S}$-subalgebra of $\mathcal{S}\text{-tree}(X)$.

Recall the definition of subsignature from page 115:

Definition 63. A signature $\mathcal{S}$ is a **subsignature** of another signature $\mathcal{S}'$ (notation: $\boldsymbol{\mathcal{S} \subseteq \mathcal{S}'}$) when $|\mathcal{S}| \subseteq |\mathcal{S}'|$ and the arities of the common operations are the same.

Subsignatures also give rise to subalgebras:

Proposition 136. If $\mathcal{S} \subseteq \mathcal{S}'$, then $\mathcal{S}\text{-tree}(X)$ is an $\mathcal{S}$-subalgebra of $\mathcal{S}'\text{-tree}(X)$.

Again, Kleisli extension preserves subsets:

Lemma 137. Let (A, a) be an $\mathcal{S}$-algebra and let $V \subseteq A$ be a subalgebra of it. If function $f : X \to A$ happens to map $U \subseteq X$ to V, then the Kleisli extension

$$f^* \ : \ \mathcal{S}\text{-tree}(X) \ \to \ A$$

maps $\mathcal{S}\text{-tree}(U)$ to V.

The proof is trivial.

Relevance to our semantics. We took the semantics of function types to be functions between two sets, but we will give pers on the domain and codomain. Definable functions map related arguments to related trees (as we will prove in Theorem 148). So we will have a subset of "good" functions which map all related pairs of arguments to related trees.

Definable computations of function type can only return definable functions. So in that case, we define a "good" tree to be a tree whose leaves are all good functions.

On trees with leaves valued in a per

We would like to use the free congruence on trees with leaves valued in a per. But this is the wrong concept, as congruences are reflexive and we want "bad" leaves to be excluded from our free congruence.

The trick is to realise that every per $\approx$ on a set X is just an equivalence relation on its domain $\mathrm{dom}(X, \approx)$ (recall Proposition 115 on page 151). We use the following construction:

Definition 138. Let $\mathbb{A} = (A, \approx)$ be a set with a per, so that $\approx$ is an equivalence relation on $\mathrm{dom}(A, \approx)$. On trees with leaves in the subset $\mathcal{S}\text{-tree}(\mathrm{dom}(A, \approx)) \subseteq \mathcal{S}\text{-tree}(A)$ we have the free Θ-congruence $\sim_{\Theta,\approx}$, which is an equivalence relation. And set $\mathcal{S}\text{-tree}(\mathrm{dom}(A, \approx))$ is a subset of $\mathcal{S}\text{-tree}(A)$, therefore $\sim_{\Theta,\approx}$ is a per on $\mathcal{S}\text{-tree}(A)$. We call this set with a per: $\boldsymbol{\mathcal{S}/_{\Theta}\text{-tree}(\mathbb{A})} \stackrel{\text{def}}{=} (\mathcal{S}\text{-tree}(A), \sim_{\Theta,\approx})$.

Again, Kleisli extension behaves well with respect to this construction:

Lemma 139. Let additionally (B, b) be a set with an $\mathcal{S}$-algebra, with a subalgebra $V \subseteq B$. And let $\simeq$ be a congruence on (V, b), so that $\simeq$ is a per on B. And let f be a Per-morphism $f : (A, \approx) \to (B, \simeq)$. Then f^* is a Per-morphism

$$f^* \quad : \quad \mathcal{S}\!/_{\Theta}\text{-tree}(A, \approx) \longrightarrow (B, \simeq) \quad .$$

Proof. A function is a Per-morphism if it preserves the subset, and if it preserves the per on the subset (Lemma 118). We have proven the former in Lemma 137 and the latter in Lemma 133. □

Proposition 140. $\mathcal{S}\!/_{\Theta}\text{-tree}(-)$ is a strong monad on Per.

Proof. Recall that $\mathcal{S}\text{-tree}(-)$ is a monad on Set; we show that it lifts to a monad $\mathcal{S}\!/_{\Theta}\text{-tree}(-)$ on Per. Recall that Per is a concrete category, so we can define its morphisms merely by giving the function on the underlying set.

The unit, functor action, multiplication, and strength for $\mathcal{S}\!/_{\Theta}\text{-tree}(-)$ are the same as the ones for $\mathcal{S}\text{-tree}(-)$. It is obvious that $\eta = \mathrm{leaf}(-)$ is a Per-morphism, and that functorial action on Per-morphisms forms a Per-morphism. Multiplication is just Kleisli extension of the unit, which must therefore also be a Per-morphism. It is easy to check that strength $str : \mathbb{A} \times \mathcal{S}\!/_{\Theta}\text{-tree}(\mathbb{B}) \to \mathcal{S}\!/_{\Theta}\text{-tree}(\mathbb{A} \times \mathbb{B})$ is a Per-morphism. The corresponding equations, including naturality, follow from the fact that $\mathcal{S}\text{-tree}(-)$ was already a strong monad on Set. □

7.5 An enriched semantics

In this section, we enrich the denotational semantics that we gave in Chapter 6.[1] There, we defined sets

$$\llbracket \Delta \vdash A \rrbracket \in \mathrm{Set} \qquad \text{for every value type } A$$

$$\llbracket \Delta \vdash \underline{A} \rrbracket \in \mathrm{Set} \qquad \text{for every computation type } \underline{A}\,.$$

Then the semantics of a value was a function

$$\llbracket V \rrbracket : \llbracket \Delta \vdash \Gamma \rrbracket \to \llbracket \Delta \vdash A \rrbracket$$

and the semantics of a computation was a function

$$\llbracket M \rrbracket : \llbracket \Delta \vdash \Gamma \rrbracket \to \llbracket \Delta \vdash \underline{A} \rrbracket\,.$$

In this section, we enrich the semantics by defining pers on all these sets. And in Theorem 148 on page 173, we prove that all the $\llbracket V \rrbracket, \llbracket M \rrbracket$ preserve the respective pers.

Recap

But first, let us quickly recap the structure of our language and the structure of the set-based semantics from Chapter 6. We assume a finite set L of operations. The set L may contain some ambient effects, as well as operations that are defined by a $(-\ \mathrm{wherealg}\ \mathcal{A})$ construct. Given L, we have both value and computation types:

value types	$A, B \ ::= \ A \to \underline{B} \mid 0 \mid 1 \mid A + B \mid A \times B$
computation types	$\underline{A}, \underline{B} \ ::= \ B \,!\, \theta \qquad (\theta \subseteq L)$

Recall that values are "inert" things that can be passed around, and values of sum or product type can be inspected. Computations are "run" by a computer, and may choose to perform operations from a subset $\theta \subseteq L$ of *allowed* operations, before ultimately returning a value from some type A. Such computation would have type $A \,!\, \theta$. We write an underline below type

[1] The word *enrich* here has nothing to do with enriched category theory. We apologise for the concidence.

metavariables that represent a computation type, and no underline below type metavariables that represent a value type. We use L for the set of all operation names in the context, and $\Delta : L \to \mathbb{N}$ assigns to every operation name an arity.

We reproduce from page 116 the semantics of types. Recall that $[\![\theta; \Delta]\!]$ is notation for the tree signature consisting of only operations θ, with arities according to Δ.

Definition 66. Let L be a finite set. The **semantics of a type** over L is a set that depends on the arity assignment. Let Δ be an arity assignment on L, then

$$\begin{aligned}
[\![\Delta \vdash 0]\!] &= \varnothing \\
[\![\Delta \vdash 1]\!] &= \{*\} \\
[\![\Delta \vdash A + B]\!] &= [\![\Delta \vdash A]\!] + [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \times B]\!] &= [\![\Delta \vdash A]\!] \times [\![\Delta \vdash B]\!] \\
[\![\Delta \vdash A \to \underline{B}]\!] &= [\![\Delta \vdash A]\!] \to [\![\Delta \vdash \underline{B}]\!] \\
[\![\Delta \vdash B\,!\,\theta]\!] &= [\![\theta; \Delta]\!]\text{-tree}([\![\Delta \vdash B]\!])
\end{aligned}$$

Given Δ, the semantics of a value context $\Gamma = (x : \Gamma(x))_{x \in |\Gamma|}$ is the product of the semantics of types:

$$[\![\Delta \vdash \Gamma]\!] = \prod_{x \in |\Gamma|} [\![\Delta \vdash \Gamma(x)]\!] \ .$$

The pers

Assume a $[\![\mathrm{dom}(\Delta); \Delta]\!]$-equational theory Θ. For a fixed Θ and for every type $A, \underline{A}$ we are going to define a per $\sim_{[\![\Delta \vdash -]\!]/\Theta}$:

$$\begin{aligned}
\sim_{[\![\boldsymbol{\Delta} \vdash \boldsymbol{A}]\!]/\boldsymbol{\Theta}} &\subseteq ([\![\Delta \vdash A]\!] \times [\![\Delta \vdash A]\!]) \\
\sim_{[\![\boldsymbol{\Delta} \vdash \boldsymbol{\underline{A}}]\!]/\boldsymbol{\Theta}} &\subseteq ([\![\Delta \vdash \underline{A}]\!] \times [\![\Delta \vdash \underline{A}]\!])
\end{aligned}$$

so that we have the following objects of Per:

$$\begin{aligned}
[\![\boldsymbol{\Delta} \vdash \boldsymbol{A}]\!]^{/\boldsymbol{\Theta}} &\overset{\text{def}}{=} \left([\![\Delta \vdash A]\!],\ \sim_{[\![\Delta \vdash A]\!]/\Theta}\right) \in \mathrm{Per} \\
[\![\boldsymbol{\Delta} \vdash \boldsymbol{\underline{A}}]\!]^{/\boldsymbol{\Theta}} &\overset{\text{def}}{=} \left([\![\Delta \vdash \underline{A}]\!],\ \sim_{[\![\Delta \vdash \underline{A}]\!]/\Theta}\right) \in \mathrm{Per}
\end{aligned}$$

Remark. The difference in notation between the set $[\![\Delta \vdash -]\!]$ and the set-with-per $[\![\Delta \vdash -]\!]^{/\Theta}$ is merely the $/\Theta$ superscript.

Let Θ be an $[\![\mathrm{dom}(\Delta);\Delta]\!]$-equational theory.

We define the set-with-per $\boxed{[\![\Delta \vdash A]\!]^{/\Theta}}$ (*resp.* $\boxed{[\![\Delta \vdash \underline{A}]\!]^{/\Theta}}$)

as the per $\boxed{\sim_{[\![\Delta\vdash A]\!]^{/\Theta}}}$ (*resp.* $\boxed{\sim_{[\![\Delta\vdash \underline{A}]\!]^{/\Theta}}}$) *on set* $[\![\Delta \vdash A]\!]^{/\Theta}$ (*resp.* $[\![\Delta\vdash \underline{A}]\!]^{/\Theta}$) .

$$
\begin{array}{lcl}
[\![\Delta \vdash 0]\!]^{/\Theta} & = & \varnothing \text{ with the trivial per} \\
[\![\Delta \vdash 1]\!]^{/\Theta} & = & \{\star\} \text{ with the full per} \\
[\![\Delta \vdash A + B]\!]^{/\Theta} & = & \text{the } \mathrm{Per}\text{-coproduct of } [\![\Delta \vdash A]\!]^{/\Theta} \text{ and } [\![\Delta \vdash B]\!]^{/\Theta} \\
[\![\Delta \vdash A \times B]\!]^{/\Theta} & = & \text{the } \mathrm{Per}\text{-product of } [\![\Delta \vdash A]\!]^{/\Theta} \text{ and } [\![\Delta \vdash B]\!]^{/\Theta} \\
[\![\Delta \vdash A \to \underline{B}]\!]^{/\Theta} & = & \text{the } \mathrm{Per}\text{-exponential from } [\![\Delta \vdash A]\!]^{/\Theta} \text{ into } [\![\Delta \vdash \underline{B}]\!]^{/\Theta} \\
[\![\Delta \vdash B\,!\,\theta]\!]^{/\Theta} & = & [\![\theta;\Delta]\!]/_{\Theta|_{[\![\theta;\Delta]\!]}}\text{-}\mathrm{tree}\big([\![\Delta \vdash B]\!]^{/\Theta}\big) \\
 & = & [\![\theta;\Delta]\!]\text{-}\mathrm{tree}\big([\![\Delta \vdash B]\!]\big) \text{ with the free } \Theta|_{[\![\theta;\Delta]\!]}\text{-congruence } \sim_{\Theta|_{[\![\theta;\Delta]\!]},\sim_{[\![\Delta\vdash B]\!]^{/\Theta}}} \\
[\![\Delta \vdash \Gamma]\!]^{/\Theta} & = & \text{the } |\Gamma|\text{-wide } \mathrm{Per}\text{-product of the sets with pers } \Big([\![\Delta \vdash \Gamma(x)]\!]^{/\Theta}\Big)_{x\in|\Gamma|}
\end{array}
$$

Figure 7.1: Detail of the definition of the set-with-per $[\![\Delta \vdash -]\!]^{/\Theta}$, for a given $[\![\mathrm{dom}(\Delta);\Delta]\!]$-equational theory Θ. Recall pers from §7.3. The underlying set of each set-with-per is always equal to the corresponding set $[\![\Delta \vdash -]\!]$ from Chapter 6. Recall the definition of restricted theory $\Theta|_{-}$ from Def. 110, and $\mathcal{S}/_{\Theta}\text{-}\mathrm{tree}(-)$ and $\sim_{-,-}$ from Def. 132. The definition of signature $[\![\theta;\Delta]\!]$ was on page 115.

We define the pers on $[\![\Delta \vdash -]\!]$ as follows. On $01{+}{\times}{\to}$-types, we follow the bicartesian closed structure of Per. For computation types $A!\theta$, recall that the semantics $[\![\Delta \vdash A!\theta]\!]$ are the $[\![\theta;\Delta]\!]$-trees with leaves in $[\![\Delta \vdash A]\!]$. We consider the trees with only good leaves. On these trees, we consider the free $\Theta|_{[\![\theta;\Delta]\!]}$-congruence $[\![\theta;\Delta]\!]/_{\Theta|_{[\![\theta;\Delta]\!]}}\text{-}\mathrm{tree}\big([\![\Delta \vdash A]\!]^{/\Theta}\big)$.[2] Additionally, we define the enriched semantics $[\![\boldsymbol{\Delta \vdash \Gamma}]\!]^{/\boldsymbol{\Theta}}$ of a context to be the $|\Gamma|$-wide Per-product of each $[\![\Delta \vdash \Gamma(x)]\!]^{/\Theta}$.

We detail the pers in Figure 7.1.

Before we proceed to an overview of the rest of this section, let us consider the following

[2] Recall Definitions 110, 138.

proposition. In Chapter 6, Prop. 68, we showed that the $[\![\Delta \vdash -]\!]$-semantics of a type does not depend on L as long as L mentions all the operations in the type, and it only depends on the values of Δ for those operations. The analogous statement also holds for the enriched semantics:

Proposition 141. Let Δ and Δ' be two arity assignments with disjoint underlying sets, and let Θ be an $[\![\mathrm{dom}(\Delta); \Delta]\!]$-equational theory. Recall from Proposition 102 that Θ is also a $[\![\mathrm{dom}(\Delta), \mathrm{dom}(\Delta'); \Delta,\Delta']\!]$-equational theory.

1. Let A (resp. $\underline{A}$) be a value (resp. computation) type over $\mathrm{dom}(\Delta)$. Then

$$\begin{aligned} [\![\Delta, \Delta' \vdash A]\!]^{/\Theta} &= [\![\Delta \vdash A]\!]^{/\Theta} \\ \text{and} \quad [\![\Delta, \Delta' \vdash \underline{A}]\!]^{/\Theta} &= [\![\Delta \vdash \underline{A}]\!]^{/\Theta} \ . \end{aligned}$$

2. Let Γ be a context over $\mathrm{dom}(\Delta)$. Then

$$[\![\Delta, \Delta' \vdash \Gamma]\!]^{/\Theta} = [\![\Delta \vdash \Gamma]\!]^{/\Theta} \ .$$

Recall that equality of pers means that the underlying sets are the same, the subsets are the same, and the equivalences are the same.

Like for Proposition 68, the proof is trivial, with the first part by induction on the type.

Recall that in Figure 6.3 on page 123 we defined the semantics of subsumption as a function

$$[\![\Delta \vdash A \leq A']\!] \ : \ [\![\Delta \vdash A]\!] \to [\![\Delta \vdash A']\!] \ .$$

Lemma 142 (semantic subsumption preserves pers). Let Δ be an arity assignment, and let Θ be a $[\![\theta; \Delta]\!]$-equational theory. Let A and A' be two types over $\mathrm{dom}(\Delta)$.

1. If $A \leq A'$, then $[\![\Delta \vdash A \leq A']\!]$ preserves the per:

$$[\![\Delta \vdash A \leq A']\!] \ : \ [\![\Delta \vdash A]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash A']\!]^{/\Theta} \in \mathrm{Per} \ .$$

2. Let $\theta \subseteq \theta' \subseteq \mathrm{dom}(\Delta)$. If $A \leq A'$, then applying $[\![\Delta \vdash A \leq A']\!]$ on the leaves preserves the per:

$$[\![\theta; \Delta]\!]\text{-tree}\Big([\![\Delta \vdash A \leq A']\!]\Big) \ : \ [\![\Delta \vdash A \,!\, \theta]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash A' \,!\, \theta']\!]^{/\Theta} \in \mathrm{Per} \ .$$

Proof. We show the first part; the second part is completely analogous.

Apply induction on the proof of $A \leq A'$. The only interesting case is:

$$\frac{A' \leq A \qquad B \leq B' \qquad \theta \subseteq \theta'}{A \to B \,!\, \theta \;\leq\; A' \to B' \,!\, \theta'}$$

Recall that $[\![\Delta \vdash A \to B \,!\, \theta]\!]^{/\Theta}$ and $[\![\Delta \vdash A' \to B' \,!\, \theta']\!]^{/\Theta}$ are formed by exponentials into trees. The semantic subsumption $[\![\Delta \vdash A{\to}B!\theta \leq A'{\to}B'!\theta]\!]$ precomposes with $[\![\Delta \vdash A' \leq A]\!]$ and postcomposes with applying $[\![\Delta \vdash B \leq B']\!]$ on the leaves; they both preserve the pers by induction. This gives us a Per-morphism

$$\begin{aligned}
&[\![\Delta \vdash A \to B \,!\, \theta \leq A' \to B' \,!\, \theta]\!] \\
&\quad : \Big([\![\Delta \vdash A]\!]^{/\Theta} \;\to\; [\![\theta; \Delta]\!]_{/\Theta|_{[\![\theta;\Delta]\!]}}\text{-tree}\big([\![\Delta \vdash B]\!]^{/\Theta}\big)\Big) \\
&\qquad \longrightarrow \Big([\![\Delta \vdash A']\!]^{/\Theta} \to [\![\theta; \Delta]\!]_{/\Theta|_{[\![\theta;\Delta]\!]}}\text{-tree}\big([\![\Delta \vdash B']\!]^{/\Theta}\big)\Big)
\end{aligned}$$

It remains to check that the injection from $[\![\theta; \Delta]\!]_{/\Theta|_{[\![\theta;\Delta]\!]}}\text{-tree}(\cdots)$ to $[\![\theta'; \Delta]\!]_{/\Theta|_{[\![\theta';\Delta]\!]}}\text{-tree}(\cdots)$ preserves the per, but indeed the subset becomes larger and the equivalence relation is weaker on the right, so it does preserve the per. □

Overview of the rest of this section

The main theorem of this section (Theorem 148 on page 173) says that the semantics of terms preserves the pers:

$$\begin{aligned}
[\![V]\!] \;&:\; [\![\Delta \vdash \Gamma]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash A]\!]^{/\Theta} \;\in \mathrm{Per} \\
[\![M]\!] \;&:\; [\![\Delta \vdash \Gamma]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash \underline{A}]\!]^{/\Theta} \;\in \mathrm{Per}
\end{aligned}$$

Let us now sketch the rest of this section. The proof of Theorem 148 is by induction on the term, and mostly routine, as our semantics is built using a strong monad on a cartesian closed category. The interesting case is $[\![M \text{ wherealg } (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]$, which is defined as the composition of $[\![M]\!]$ and $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$.

Especially the latter is interesting; let us recall its definition. By induction, we have a semantic definition of each new operation $\alpha \in K$, where K is the underlying set of Δ':

$$[\![N_\alpha]\!] \;:\; \Big([\![\Delta \vdash \Gamma]\!] \times [\![\Delta \vdash A \,!\, \theta]\!]^{\Delta'(\alpha)}\Big) \longrightarrow [\![\Delta \vdash A \,!\, \theta]\!]$$

We gather all these $[\![N_\alpha]\!]$ together in the semantics of the syntactic algebra:

$$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!] = \left([\![N_\alpha]\!]\right)_{\alpha\in K} \;:\; \left([\![\Delta \vdash \Gamma]\!] \times \sum_{\alpha\in K} [\![\Delta \vdash A\,!\,\theta]\!]^{\Delta'(\alpha)}\right) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!] \;.$$

We map trees on a larger signature $[\![(\theta \cup K); \Delta, \Delta']\!]$ to trees on a smaller signature $[\![\theta; \Delta]\!]$ using the *extended initial algebra morphism* (page 114) on each of the fibers $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]_\rho$, to obtain the function

$$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} \;:\; \left([\![\Delta \vdash \Gamma]\!] \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]\right) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!] \;.$$

We must prove that this function preserves the pers:

$$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} \;:\; \left([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

Per Lemma 118, that means that we must prove three things:

1. *Lemma 144*: $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ restricts to a function

   $$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$$
   $$: \left(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta}\right) \times \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)\right) \longrightarrow \mathrm{dom}\left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\right)$$

2. *Lemma 145*: $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ preserves the equivalence relation in the left argument when the right argument is in the subset — that is, $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ is a Per-morphism:

   $$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} \;:\; \left([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)\right) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

   For convenience, we write $\mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)$ for the full subset and discrete equivalence relation on the set $\mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)$.

3. *Corollary 147*: $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ preserves the equivalence relation in the right argument when the left argument is in the subset:

   $$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} : \left(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta}\right) \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right) \to [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

The actual application of Lemma 118 is in the proof of Theorem 148, on page 176.

Analysis

We will now proceed to give some lemmas.

Remark 143. Let $\Delta : L \to \mathbb{N}$ be an arity assignment, and let $\theta \subseteq L$.

1. Recall from page 115: $[\![\theta; \Delta]\!]$ is a subsignature of $[\![L; \Delta]\!]$.

2. Suppose that we additionally have some arity assignment $\Delta' : K \to \mathbb{N}$ with disjoint underlying set, $L \cap K = \varnothing$. Then of course $\theta \subseteq L \cup K$. Signature $[\![\theta; \Delta]\!]$ is equal to signature $[\![\theta; \Delta, \Delta']\!]$.

And Proposition 136 shows that:

3. For $\theta \subseteq \psi \subseteq L$ and a value type A over L, we have that $[\![\Delta \vdash A\,!\,\theta]\!] \subseteq [\![\Delta \vdash A\,!\,\psi]\!]$.

Lemma 144. Let Δ be an arity assignment on L, let Θ be an $[\![L; \Delta]\!]$-equational theory, let $\Delta; \Gamma \vdash_{\text{alg}} (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} : \Delta' \Rightarrow A\,!\,\theta$, and suppose that each $[\![N_\alpha]\!]$ preserves the pers (is a Per-morphism)

$$[\![N_\alpha]\!] \quad : \quad \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \big([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \big)^{\Delta'(\alpha)} \Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \,. \tag{7.2}$$

Then $[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\text{ext}}$ maps the subsets to subsets:

$$[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\text{ext}} : \Big(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta} \right) \times \mathrm{dom}\left([\![\Delta, \Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta} \right) \Big) \to \mathrm{dom}\left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \right) \tag{7.3}$$

The proof is trivial by induction on the right argument.

Lemma 145. Let Δ be an arity assignment on L, let Θ be an $[\![L; \Delta]\!]$-equational theory, let $\Delta; \Gamma \vdash_{\text{alg}} (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} : \Delta' \Rightarrow A\,!\,\theta$, and suppose that each $[\![N_\alpha]\!]$ preserves the pers according to (7.2). If the right argument to $[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\text{ext}}$ is in the subset, then $[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\text{ext}}$ preserves the per in the left argument:

$$[\![(\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}]\!]^{\text{ext}} : \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \mathrm{dom}\left([\![\Delta, \Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta} \right) \Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \;\in\; \mathrm{Per}$$

Proof. By induction on the right argument:

- *In case of a leaf.* For every $v \in \mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A \rrbracket^{/\Theta}\right) = \mathrm{dom}\left(\llbracket \Delta \vdash A \rrbracket^{/\Theta}\right)$, we have that the function

$$\rho \;\mapsto\; \llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle \rho, \mathrm{leaf}(v) \rangle$$

  is constantly $\mathrm{leaf}(v)$, which is in the subset.

- *In case of a tree $\boldsymbol{\beta}(\vec{t})$ for operation $\beta \in \theta$.* The induction hypothesis tells us that each $\left(\rho \mapsto \llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle \rho, t_i \rangle\right)$ preserves the per, and thus

$$\rho \;\mapsto\; \boldsymbol{\beta}\left(\overrightarrow{\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle \rho, t \rangle}\right) \tag{7.4}$$

  preserves the per, by definition of that per. We need to show that

$$\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle -, \boldsymbol{\beta}(\vec{t}) \rangle \tag{7.5}$$

  preserves the per. But functions (7.4) and (7.5) are equal, so (7.5) preserves the per.

- *In case of a tree $\boldsymbol{\beta}(\vec{t})$ for operation $\beta \in K$.* The induction hypothesis tells us that each $\left(\rho \mapsto \llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle \rho, t_i \rangle\right)$ preserves the per. We also know that $\llbracket N_\beta \rrbracket$ preserves the per, and thus

$$\rho \;\mapsto\; \llbracket N_\beta \rrbracket \left(\overrightarrow{\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle \rho, t \rangle}\right) \tag{7.6}$$

  preserves the per by some reshuffling in Per.

  We need to show that

$$\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}\, \langle -, \boldsymbol{\beta}(\vec{t}) \rangle \tag{7.7}$$

  preserves the per. But functions (7.6) and (7.7) are equal, so (7.7) preserves the per. □

The following lemma helps us prove that $\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}$ preserves the equivalence relation in the right argument.

Lemma 146. Let Δ and Δ' be arity assignment on disjoint underlying sets L resp. K. Let Θ be an $[\![L;\Delta]\!]$-equational theory. Let A be a type over L, and let $\theta \subseteq L$. For each $\alpha \in K$, let f_α be a Per-morphism

$$\boxed{f_\alpha : F_{[\![K;\Delta']\!]}\big([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}$$

so that trivially (Def. 62):

$$\boxed{\vec{f}^{\mathrm{ext}} : [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!] \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]} \ .$$

We claim that it is also a Per-morphism

$$\boxed{\vec{f}^{\mathrm{ext}} : [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}} \ . \tag{7.8}$$

Proof. We can see that $\vec{f}^{\mathrm{ext}}$ preserves the subset

$$\begin{aligned}\vec{f}^{\mathrm{ext}} \ : \ & \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right) \ \longrightarrow \ \mathrm{dom}\left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\right) \\ = \ & \Big([\![\theta \cup K;\ \Delta,\Delta']\!]\text{-tree}(\mathrm{dom}\left([\![\Delta \vdash A]\!]^{/\Theta}\right)) \to [\![\theta;\Delta]\!]\text{-tree}(\mathrm{dom}\left([\![\Delta \vdash A]\!]^{/\Theta}\right))\Big)\end{aligned}$$

by a simple induction argument. The complicated part is proving that it preserves the equivalence relation on $\mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)$. We prove this by induction on the proof that two inputs are related. So let x, x' range over

$$\forall x, x' \ \in \ \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right) \quad \text{such that} \quad x \ \sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}} \ x'$$

and we must prove:

$$\vec{f}^{\mathrm{ext}}(x) \ \sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}} \ \vec{f}^{\mathrm{ext}}(x')$$

Recall from Figure 7.1 on page 164 that $\sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}}$ was defined as the free congruence $\sim_{\Theta|_{[\![\theta;\Delta]\!]},\sim_{[\![\Delta \vdash A]\!]^{/\Theta}}}$, which was defined in Definition 132 as the least endorelation closed under a certain set of 6 rules. So there must be a proof of $x \sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}} x'$ built up from those 6 rules. We do induction on that proof. We will split up the last rule in two separate cases; here are the 7 cases.

1. *Leaf.* Let $z, z' \in \mathrm{dom}\left(\llbracket \Delta \vdash A \rrbracket^{/\Theta}\right)$ and $z \sim_{\llbracket \Delta \vdash A \rrbracket^{/\Theta}} z'$. We apply $\vec{f}^{\mathrm{ext}}$ to $\mathrm{leaf}(z)$ and $\mathrm{leaf}(z')$. But $\vec{f}^{\mathrm{ext}}$ is the identity on leaves in the right argument, and per the leaf rule for the codomain,

$$\vec{f}^{\mathrm{ext}}\big(\mathrm{leaf}(z)\big) = \mathrm{leaf}(z) \;\; \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \;\; \mathrm{leaf}(z') = \vec{f}^{\mathrm{ext}}\big(\mathrm{leaf}(z')\big) \;\; .$$

2. *Reflexivity*. Let $x \in \mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A!(\theta \cup K) \rrbracket^{/\Theta}\right)$ so that $\vec{f}^{\mathrm{ext}}(x) \in \mathrm{dom}\left(\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}\right)$. Per the reflexivity rule for $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}}$, we have $\vec{f}^{\mathrm{ext}}(x) \sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x)$.

3. *Symmetry*. Let $x, x' \in \mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A!(\theta \cup K) \rrbracket^{/\Theta}\right)$ and $x \sim_{\llbracket \Delta,\Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta}} x'$. By the induction hypothesis, $\vec{f}^{\mathrm{ext}}(x) \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x')$. Then by the symmetry rule for $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}}$ we have $\vec{f}^{\mathrm{ext}}(x') \sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x)$.

4. *Transitivity*. Let $x, x', x'' \in \mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A!(\theta \cup K) \rrbracket^{/\Theta}\right)$ and $x \sim_{\llbracket \Delta,\Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta}} x'$ and $x' \sim_{\llbracket \Delta,\Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta}} x''$. By the induction hypothesis, $\vec{f}^{\mathrm{ext}}(x) \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x')$ and $\vec{f}^{\mathrm{ext}}(x') \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x'')$. Then by the transitivity rule for $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}}$ we have $\vec{f}^{\mathrm{ext}}(x) \sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}(x'')$.

5. *Closed under substitutions of axioms in* $\Theta|_{\llbracket \theta;\Delta \rrbracket}$.

   Let $(I \vdash t = u) \in \Theta|_{\llbracket \theta \cup K;\Delta,\Delta' \rrbracket}$. As Θ is a set of $\llbracket L;\Delta \rrbracket$-equations, and $L \cap (\theta \cup K) = \theta$, we know that $\Theta|_{\llbracket \theta \cup K;\Delta,\Delta' \rrbracket}$ are all $\llbracket \theta;\Delta \rrbracket$-equations. So $\Theta|_{\llbracket \theta \cup K;\Delta,\Delta' \rrbracket} = \Theta|_{\llbracket \theta;\Delta \rrbracket}$ and t and u are $\llbracket \theta;\Delta \rrbracket$-terms. And let $\vec{\sigma}$ be an I-ary family of elements of $\mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta}\right)$, so that our x-inputs in this case are

$$\llbracket t \rrbracket_{\llbracket \theta,K;\Delta,\Delta' \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A \rrbracket^{/\Theta}\right)\right)}(\vec{\sigma}) \;\; \sim_{\llbracket \Delta,\Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta}} \;\; \llbracket u \rrbracket_{\llbracket \theta,K;\Delta,\Delta' \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A \rrbracket^{/\Theta}\right)\right)}(\vec{\sigma})$$

   and we have to show that

$$\vec{f}^{\mathrm{ext}}\left(\llbracket t \rrbracket_{\llbracket \theta,K;\Delta,\Delta' \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A \rrbracket^{/\Theta}\right)\right)}(\vec{\sigma})\right) \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \vec{f}^{\mathrm{ext}}\left(\llbracket u \rrbracket_{\llbracket \theta,K;\Delta,\Delta' \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta,\Delta' \vdash A \rrbracket^{/\Theta}\right)\right)}(\vec{\sigma})\right) \;\; . \tag{7.9}$$

   Recall from Prop. 65 that $\vec{f}^{\mathrm{ext}}$ preserves the $\llbracket \theta;\Delta \rrbracket$-algebra, and so by Lemma 122 $\vec{f}^{\mathrm{ext}}$ commutes with the evaluation: we merely have to show that

$$\llbracket t \rrbracket_{\llbracket \theta;\Delta \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta \vdash A \rrbracket^{/\Theta}\right)\right)}\left(\vec{f}^{\mathrm{ext}}(\sigma_i)\right)_{i \in I} \;\; \sim_{\llbracket \Delta \vdash A\,!\,\theta \rrbracket^{/\Theta}} \;\; \llbracket u \rrbracket_{\llbracket \theta;\Delta \rrbracket\text{-tree}\left(\mathrm{dom}\left(\llbracket \Delta \vdash A \rrbracket^{/\Theta}\right)\right)}\left(\vec{f}^{\mathrm{ext}}(\sigma_i)\right)_{i \in I} \;\; .$$

But $\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}$ is the free $\Theta|_{[\![\theta;\Delta]\!]}$-congruence on an equivalence relation, and so it also satisfies $\Theta|_{[\![\theta;\Delta]\!]}$. Recall Definition 128: this directly implies the required equation.

6. *Preserved by semantic operations from $\theta \subseteq L$.*

   Let $\beta \in \theta$ be the operation; it has arity $\Delta(\beta)$. And let $\vec{x}$, $\vec{y}$ be two $\Delta(\beta)$-ary families of elements of $\mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)$, pairwise $\sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta\cup K)]\!]^{/\Theta}}$. By the induction hypothesis,

$$\forall i \in \{1,..,\Delta(\beta)\}: \quad \vec{f}^{\mathrm{ext}}(x_i) \;\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}\; \vec{f}^{\mathrm{ext}}(y_i) \;\;.$$

   Recall that $\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}$ is a $[\![\theta;\Delta]\!]$-congruence, and so

$$\boldsymbol{\beta}\big(\overrightarrow{\vec{f}^{\mathrm{ext}}(x)}\big) \;\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}\; \boldsymbol{\beta}\big(\overrightarrow{\vec{f}^{\mathrm{ext}}(y)}\big) \;\;.$$

   And recall from Prop. 65 that $\vec{f}^{\mathrm{ext}}$ preserves the $[\![\theta;\Delta]\!]$-algebra, so we can exchange $\boldsymbol{\beta}$ and $\vec{f}^{\mathrm{ext}}$, which gives us the required equation:

$$\vec{f}^{\mathrm{ext}}(\boldsymbol{\beta}(\vec{x})) \;\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}\; \vec{f}^{\mathrm{ext}}(\boldsymbol{\beta}(\vec{y})) \;\;.$$

7. *Preserved by semantic operations from K.*

   Let $\beta \in K$ be the operation; it has arity $\Delta'(\beta)$. And let $\vec{x}$, $\vec{y}$ be two $\Delta'(\beta)$-ary families of elements of $\mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)$, pairwise $\sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta\cup K)]\!]^{/\Theta}}$. By the induction hypothesis,

$$\forall i \in \{1,..,\Delta'(\beta)\}: \quad \vec{f}^{\mathrm{ext}}(x_i) \;\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}\; \vec{f}^{\mathrm{ext}}(y_i) \;\;.$$

   We must prove that

$$\vec{f}^{\mathrm{ext}}(\boldsymbol{\beta}(\vec{x})) \;\sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}}\; \vec{f}^{\mathrm{ext}}(\boldsymbol{\beta}(\vec{y})) \;\;.$$

   By definition, the two sides are equal to

$$f_\beta\Big(\overrightarrow{\vec{f}^{\mathrm{ext}}(x)}\Big) \quad \text{and} \quad f_\beta\Big(\overrightarrow{\vec{f}^{\mathrm{ext}}(y)}\Big)$$

   which follows from the assumption that f_β preserves the equivalence relation, together with the induction hypothesis.

This concludes the induction on the proof of $x \sim_{[\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}} x'$. □

Corollary 147. Let Δ be an arity assignment on L, let Θ be an $[\![L;\Delta]\!]$-equational theory, let $\Delta;\Gamma \vdash_{\mathrm{alg}} (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K} : \Delta' \Rightarrow A\,!\,\theta$, and suppose that each $[\![N_\alpha]\!]$ preserves the pers according to (7.2). If the left argument to $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ is in the subset, then $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ preserves the per in the right argument:

$$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} : \Big(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta}\right) \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\Big) \to [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \in \mathrm{Per}$$

The main result

We state and prove the main theorem of this section.

Theorem 148 (the semantics preserves the pers)**.** Let Δ be an arity assignment on L, and let Θ be an $[\![L;\Delta]\!]$-equational theory.

1. For each judgement $\Delta;\Gamma \vdash_{\mathrm{v}} V : A$, we have that $[\![V]\!]$ is a Per-morphism $[\![V]\!] : [\![\Delta \vdash \Gamma]\!]^{/\Theta} \to [\![\Delta \vdash A]\!]^{/\Theta}$. That is, it preserves the per in its environment argument.

2. For each judgement $\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\theta$, we have that $[\![M]\!]$ is a Per-morphism $[\![M]\!] : [\![\Delta \vdash \Gamma]\!]^{/\Theta} \to [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$. That is, it preserves the per in its environment argument.

3. For each judgement $\Delta;\Gamma \vdash_{\mathrm{alg}} (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K} : \Delta' \Rightarrow A\,!\,\theta$, we have that $[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}}$ is a Per-morphism

$$[\![(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}]\!]^{\mathrm{ext}} : \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\Big) \longrightarrow [\![\Delta \vdash A!\theta]\!]^{/\Theta} \ .$$

In the last statement, the notation $[\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}$ may raise some eyebrows, as surely Θ is an $[\![L;\Delta]\!]$-equational theory not an $[\![L,K;\Delta,\Delta']\!]$-equational theory. But $[\![L;\Delta]\!]$ is a subsignature of $[\![L,K;\Delta,\Delta']\!]$, and thus (recall Prop. 102), Θ is also an $[\![L,K;\Delta,\Delta']\!]$-equational theory.

Proof. By induction on the proof tree of the judgement of V resp. M. We have an additional case for algebras $(\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}$.

The judgement fixes L and Δ. In each case, we also quantify over Θ, which is an $[\![L;\Delta]\!]$-equational theory.

Look at the semantics of values and computations in Figure 6.3 on page 123; we have to prove that these functions are morphisms between the indicated objects. In most cases, we do this by constructing a morphism from morphisms that we know to preserve pers, and showing that the constructed morphism is the same function.

We consider each of the cases in turn.

- *Variable* x. Formed by projection out of the product object (Prop. 117).
- $\mathrm{return}\ V$. Formed by postcomposing $[\![V]\!]$ with $\eta{=}\mathrm{leaf}$, which is a Per-morphism (Prop. 140).
- *Operation* $\alpha\left(\vec{M}\right)$. Formed by postcomposing with $\boldsymbol{\alpha}$, which preserves the subset (by Prop. 135) and the equivalence relation (by definition of the equivalence relation, Def. 132).
- $\mathrm{let} - \mathrm{be}\ x.-$, *and abstraction* $(\lambda x.-)$. Formed by shuffling around using the cartesian structure.
- *Function call* $(--)$. Formed by shuffling around using the cartesian structure and composition.
- *Value construction:* $\langle\rangle$, inl, inr, $\langle -,-\rangle$. The semantics of these values are obtained as morphisms using the bicartesian structure of Per.
- *Pattern matching:* $\mathrm{case} - \mathrm{of}\ \{\cdots\}$. The semantics of these computations can also be obtained as morphisms using the bicartesian structure of Per.
- *Sequencing*: $M\ \mathrm{to}\ x.\ N$. By induction, we know that $[\![N]\!]$ is a Per-morphism

  $$[\![N]\!] \quad : \quad [\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta \vdash A]\!]^{/\Theta} \longrightarrow [\![\Delta \vdash B\,!\,\theta]\!]^{/\Theta}$$

  so its Kleisli extension is a Per-morphism

  $$[\![N]\!]^* \quad : \quad \mathcal{S}\!/_{\Theta|_{[\![\theta;\Delta]\!]}}\text{-tree}\big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta \vdash A]\!]^{/\Theta}\big) \longrightarrow [\![\Delta \vdash B\,!\,\theta]\!]^{/\Theta}\ ,$$

which we can precompose with monadic strength str to get

$$[\![N]\!]^* \circ str \ : \ [\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \mathcal{S}\!/_{\Theta|_{[\![\theta;\Delta]\!]}}\text{-tree}([\![\Delta \vdash A]\!]^{/\Theta}) \ \longrightarrow \ [\![\Delta \vdash B\,!\,\theta]\!]^{/\Theta} \ .$$

Precompose again with $\langle id, [\![M]\!]\rangle$ to get a morphism $[\![\Delta \vdash \Gamma]\!]^{/\Theta} \to [\![\Delta \vdash B\,!\,\theta]\!]^{/\Theta}$. As a function on the underlying sets, it coincides with $[\![M \text{ to } x.\ N]\!]$.

- *Operation definition*: $(M \text{ wherealg } \mathcal{A})$. By induction using the second and third parts of this theorem, we know that we have Per-morphisms

$$[\![M]\!] \quad : \quad [\![\Delta \vdash \Gamma]\!]^{/\Theta} \quad \longrightarrow \quad [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}$$

and

$$[\![\mathcal{A}]\!] \ : \ \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \ \times \ [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\Big) \ \longrightarrow \ [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \ .$$

The semantics of $[\![M \text{ wherealg } \mathcal{A}]\!]$ is simply the composition of these, modulo cartesian shuffling.

- *Syntactic algebra*: $(\alpha\,\vec{k} = N_\alpha)_{\alpha \in K}$.

By induction, we know that for all $\alpha \in K$, $[\![N_\alpha]\!]$ is a Per-morphism

$$\forall \alpha \in K: \ \ [\![N_\alpha]\!] : \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \big([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\big)^{\Delta'(\alpha)}\Big) \to [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \ .$$

Let us abbreviate $\mathcal{A} = (\alpha\,\vec{k} = N_\alpha)_{\alpha \in K}$. On its face, $[\![\mathcal{A}]\!]^{\mathrm{ext}}$ is a function

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} \ : \ \Big([\![\Delta \vdash \Gamma]\!] \ \times \ [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]\Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!] \ .$$

By Lemma 145 and the induction hypothesis, we know that function $[\![\mathcal{A}]\!]^{\mathrm{ext}}$ preserves the subsets

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} \ : \ \Big(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta}\right) \times \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)\Big) \ \longrightarrow \ \mathrm{dom}\left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\right) \tag{7.10}$$

and is furthermore a Per-morphism in the left argument:

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} \ : \ \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \ \times \ \mathrm{dom}\left([\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\right)\Big) \ \longrightarrow \ [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

By (7.10) we are allowed to invoke Corollary 147, which tells us that $[\![\mathcal{A}]\!]^{\mathrm{ext}}$ is also a Per-morphism in the right argument:

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} \;:\; \Big(\mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta}\right) \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

So $[\![\mathcal{A}]\!]^{\mathrm{ext}}$ preserves pers both in its left and right argument. By Lemma 118, this means that it preserves pers from the product:

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} \;:\; \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta}\Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}$$

- *Coercion of values or computations.* The semantics of these terms are formed by post-composition with $[\![\Delta \vdash A \leq B]\!]$ (resp. $[\![\Delta \vdash A\,!\,\theta \leq B\,!\,\psi]\!]$). These preserve the per by Lemma 142.

This concludes the induction, and the proof of Theorem 148. □

7.6 Theory-dependent logic

We define a theory-dependent logic for reasoning about programs with defined operations, as well as programs on ambient effects that are not I/O, and which may satisfy an equational theory as per Definition 98. Contrary to the base logic in §6.6, this judgement is parametrised by an $[\![L;\Delta]\!]$-equational theory Θ. The judgements are:

$$\Delta;\Gamma \;\vdash_{\mathrm{v}}\; V \equiv_\Theta W \;:\; A$$

$$\Delta;\Gamma \;\vdash_{\mathrm{c}}\; M \equiv_\Theta N \;:\; A\,!\,\theta$$

And the logic includes the base logic:

$$V \equiv_\Theta W \quad \text{if } V \equiv W$$

$$M \equiv_\Theta N \quad \text{if } M \equiv N$$

Furthermore, $\equiv_\Theta$ is an equivalence relation, and still compatible with all syntactic constructs.[3] The theory-dependent logic has two additional rules:

[3] For compatibility with respect to $\mathrm{wherealg}$, care must be taken that Θ only mentions operations that are in the resulting context. See Figure 7.2 for a slightly expanded version of this axiom.

- a rule that relates operation-trees that were declared related in an axiom of the theory Θ,

- a rule about operation definitions $M \text{ wherealg } \mathcal{A}$: if $\mathcal{A}$ "validates" extra axioms, then those axioms may be assumed in M.

When $\Theta = \varnothing$, then statements from the theory-dependent logic for the empty theory have strong content: in §7.6.5 we show that $V \equiv_\varnothing W$ implies $[\![V]\!] = [\![W]\!]$, and $M \equiv_\varnothing N$ implies $[\![M]\!] = [\![N]\!]$.

We explain the two new rules now; the full list of rules can be found in Figure 7.2 on page 179.

7.6.1 New rule 1: Apply axioms from the theory

The main new rule concerns computations that are formed from operations. If Θ implies some equation $I \vdash t = u$, and $\vec{M}$ is an I-collection of computations, all of the same type, then

$$t[\vec{M}/\vec{x}] \;\equiv_\Theta\; u[\vec{M}/\vec{x}] \;\;.$$

Here's an example. Suppose that

$$\Delta \stackrel{\text{def}}{=} \{\; \underline{\text{either}} : 2 \;\} \;\;.$$

Then we can form the computation

$$\underline{\text{either}}(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)) \;\;. \tag{7.11}$$

Now setting

$$\Theta_{\text{rightmost}} \stackrel{\text{def}}{=} \{x, y \;\vdash\; \underline{\text{either}}(x, y) = y\}$$

$$\Theta_{\text{comm}} \stackrel{\text{def}}{=} \{x, y \;\vdash\; \underline{\text{either}}(x, y) = \underline{\text{either}}(y, x)\}$$

the axiom says that

$$\begin{aligned}
&\underline{\text{either}}\big(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)\big) \\
&\quad =_{\Theta_{\text{rightmost}}} \ \underline{\text{either}}(\text{return } 2, \text{return } 3) \\
&\quad =_{\Theta_{\text{rightmost}}} \ \text{return } 3
\end{aligned}$$

and

$$\begin{aligned}
&\underline{\text{either}}\big(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)\big) \\
&\quad =_{\Theta_{\text{comm}}} \ \underline{\text{either}}\big(\underline{\text{either}}(\text{return } 2, \text{return } 3), \text{return } 1\big) \\
&\quad =_{\Theta_{\text{comm}}} \ \underline{\text{either}}\big(\underline{\text{either}}(\text{return } 3, \text{return } 2), \text{return } 1\big) \ . \qquad (7.12)
\end{aligned}$$

The last step follows from the axiom in Θ_{comm} together with compatibility.

7.6.2 Example algebras

Before we explain the theory-dependent logic rule for operation definitions, let us first give some syntactic algebras for these theories. The first one combines results of type nat using a presumed function $\max : \text{nat} \times \text{nat} \to \text{nat}\,!\,\varnothing$. The $\max$ function is not definable in our language (and neither does the type nat of natural numbers exist) but it can trivially be added.

We give our first algebra for $\underline{\text{either}}$:

$$\mathcal{A}_{\text{comm}} \stackrel{\text{def}}{=} \Big(\underline{\text{either}}\, k_1\, k_2 = k_1\, \langle\rangle \text{ to } v_1.\, k_2\, \langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle\Big) \ : \ (\{\underline{\text{either}}\}; \Delta) \Rightarrow \text{nat}\,!\,\varnothing$$

Let us compute with the computation from (7.11). We only need the base logic (Figure 6.4 on pages 133–135).

$$\begin{aligned}
&\underline{\text{either}}(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)) \text{ wherealg } \mathcal{A}_{\text{comm}} \\
\equiv\ &\text{(beta on operation)} \\
&\big(\lambda\langle\rangle.\, \text{return } 1 \text{ wherealg } \mathcal{A}_{\text{comm}}\big)\, \langle\rangle \text{ to } v_1. \\
&\big(\lambda\langle\rangle.\, \underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ wherealg } \mathcal{A}_{\text{comm}}\big)\, \langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle
\end{aligned}$$

$\boxed{\Delta;\Gamma \ \vdash_{\mathrm{v}} \ V \equiv_\Theta W : A}$ $\qquad$ $\boxed{\Delta;\Gamma \ \vdash_{\mathrm{c}} \ M \equiv_\Theta N : A\,!\,\theta}$

where Θ is a $[\![\mathrm{dom}(\Delta);\Delta]\!]$-equational theory.

Apply axiom of the theory:

- $\boxed{\text{If } (I \vdash t = u) \in \Theta, \text{ then} \quad t[\vec{M}/\vec{x}] \ \equiv_\Theta \ u[\vec{M}/\vec{x}]}$

Where operations are defined, the theory can grow: Let Δ and Δ' be arity assignments with disjoint underlying sets L and K, respectively, and let Γ and A be a context and type over $\mathrm{dom}(\Delta)$. Let M, M', and $(N_\alpha)_{\alpha\in K}$ be terms of type

- $\Delta,\Delta';\ \Gamma \ \vdash_{\mathrm{c}} \ M, M' \ : \ A\,!\,(\theta\cup K)$
- for each $\alpha \in K$: $\quad \Delta;\Gamma,(k_i{:}1\to A!\theta)_{i\in\{1,..,\Delta'(\alpha)\}} \vdash_{\mathrm{c}} N_\alpha : A\,!\,\theta$

and abbreviate $\mathcal{A} \stackrel{\text{def}}{=} (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K}$. Let Θ' be an $[\![\theta\cup K;\Delta,\Delta']\!]$-equational theory. Suppose that $\Theta \vdash \mathcal{A}$ validates Θ', which implies by Remark 153 that $\Theta \vdash \mathcal{A}$ validates $(\Theta\cup\Theta')$. Then:

$$\frac{\Delta,\Delta';\ \Gamma \ \vdash_{\mathrm{c}} \quad M \ \equiv_{\Theta\cup\Theta'} \ M' \quad : \ A\,!\,(\theta\cup K)}{\Delta \ ;\ \Gamma \ \vdash_{\mathrm{c}} \ M \text{ wherealg } \mathcal{A} \ \equiv_\Theta \ M' \text{ wherealg } \mathcal{A} \ : \ A\,!\,\theta}$$

Equivalences from Figure 6.4 on page 133 are inherited:

- If $V \equiv W$, then $V \equiv_\Theta W$. If $M \equiv N$, then $M \equiv_\Theta N$.

Symmetry: If $V \equiv_\Theta V'$ then $V' \equiv_\Theta V$. If $M \equiv_\Theta M'$ then $M' \equiv_\Theta M$

Transitivity:

- If $V \equiv_\Theta V'$ and $V' \equiv_\Theta V''$ then $V \equiv_\Theta V''$
- If $M \equiv_\Theta M'$ and $M' \equiv_\Theta M''$ then $M \equiv_\Theta M''$

Figure 7.2a: The rules to determine when two terms are $\equiv_\Theta$, first part. The full judgement is $\Delta;\Gamma \vdash_{\mathrm{v}} V \equiv_\Theta W : A$ (resp. $\vdash_{\mathrm{c}}$ and $\underline{A}$), but we omit some of the fields when they can be deduced. Recall the definition of $I \vdash_\Theta t = u$ from Definition 99 on page 146, and the definition of *validation* on page 186.

Compatibility rules:

- $x \equiv_\Theta x$
- $\lambda x.\, M \equiv_\Theta \lambda x.\, M' \quad$ if $\; M \equiv_\Theta M'$
- $\langle\rangle \equiv_\Theta \langle\rangle$
- $\langle V, W\rangle \equiv_\Theta \langle V', W'\rangle \quad$ if $\; V \equiv_\Theta V'$ and $W \equiv_\Theta W'$
- $\text{inl}\ V \equiv_\Theta \text{inl}\ V' \quad$ if $\; V \equiv_\Theta V'$
- $\text{inr}\ V \equiv_\Theta \text{inr}\ V' \quad$ if $\; V \equiv_\Theta V'$
- $\text{return}\ V \equiv_\Theta \text{return}\ V' \quad$ if $\; V \equiv_\Theta V'$
- $\alpha\, \vec{M} \equiv_\Theta \alpha\, \vec{N} \quad$ if $\; \forall i \in \{1, .., \Delta(\alpha)\} :\; M_i \equiv_\Theta N_i$
- M $\text{wherealg}\ (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \;\equiv_\Theta\; M'$ $\text{wherealg}\ (\alpha\, \vec{k} = N'_\alpha)_{\alpha \in K}$
  if $\; M \equiv_\Theta M'$ and $\forall \alpha \in K : N_\alpha \equiv_\Theta N'_\alpha$ — note that this only typechecks when Θ does not mention any operations from K
- $\text{let}\ V\ \text{be}\ x.\, M \equiv_\Theta \text{let}\ W\ \text{be}\ x.\, N \quad$ if $\; V \equiv_\Theta W$ and $M \equiv_\Theta N$
- $M\ \text{to}\ x.\, N \equiv_\Theta M'\ \text{to}\ x.\, N' \quad$ if $\; M \equiv_\Theta M'$ and $N \equiv_\Theta N'$
- $V\, W \equiv_\Theta V'\, W' \quad$ if $\; V \equiv_\Theta V'$ and $W \equiv_\Theta W'$
- $\text{case}\ V\ \text{of}\ \{\} \equiv_\Theta \text{case}\ V'\ \text{of}\ \{\}$
- $\text{case}\ V\ \text{of}\ \{\langle\rangle.\, M\} \equiv_\Theta \text{case}\ V'\ \text{of}\ \{\langle\rangle.\, M'\} \quad$ if $\; M \equiv_\Theta M'$
- $\text{case}\ V\ \text{of}\ \{\text{inl}\ x.\, M;\ \text{inr}\ y.\, N\} \equiv_\Theta \text{case}\ V'\ \text{of}\ \{\text{inl}\ x.\, M';\ \text{inr}\ y.\, N'\}$
  if $\; V \equiv_\Theta V'$ and $M \equiv_\Theta M'$ and $N \equiv_\Theta N'$
- $\text{case}\ V\ \text{of}\ \{\langle x, y\rangle.\, M\} \equiv_\Theta \text{case}\ V'\ \text{of}\ \{\langle x, y\rangle.\, M'\} \quad$ if $\; V \equiv_\Theta V'$ and $M \equiv_\Theta M'$

Figure 7.2b: The rules to determine when two terms are $\equiv_\Theta$, second part.

$\equiv$ (administrative)

$\big(\text{return } 1 \text{ wherealg } \mathcal{A}_{\text{comm}}\big) \text{ to } v_1.$

$\big(\underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ wherealg } \mathcal{A}_{\text{comm}}\big) \text{ to } v_2.\ \max\langle v_1, v_2\rangle$

$\equiv$ (Lemma 82)

$\text{return } 1 \text{ to } v_1.$

$\big(\underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ wherealg } \mathcal{A}_{\text{comm}}\big) \text{ to } v_2.\ \max\langle v_1, v_2\rangle$

$\equiv$ (administrative)

$\big(\underline{\text{either}}(\text{return } 2, \text{return } 3) \text{ wherealg } \mathcal{A}_{\text{comm}}\big) \text{ to } v_2.\ \max\langle 1, v_2\rangle$

$\equiv$ (beta on operation)

$\Big((\lambda\langle\rangle.\, \text{return } 2 \text{ wherealg } \mathcal{A}_{\text{comm}})\ \langle\rangle \text{ to } v_3.$

$\qquad (\lambda\langle\rangle.\, \text{return } 3 \text{ wherealg } \mathcal{A}_{\text{comm}})\ \langle\rangle \text{ to } v_4.\ \max\langle v_3, v_4\rangle\Big)$

$\text{to } v_2.\ \max\langle 1, v_2\rangle$

$\equiv$ (Lemma 82; administrative)

$\text{return } 2 \text{ to } v_3.\, \text{return } 3 \text{ to } v_4.\ \max\langle v_3, v_4\rangle \text{ to } v_2.\ \max\langle 1, v_2\rangle$

$\equiv$ (administrative)

$\max\langle 2, 3\rangle \text{ to } v_2.\ \max\langle 1, v_2\rangle$

$\equiv$ (compute)

$\text{return } 3$

We may suspect that if we had swapped the operands to either and had instead evaluated

$$\underline{\text{either}}(\underline{\text{either}}(\text{return } 3, \text{return } 2), \text{return } 1) \text{ wherealg } \mathcal{A}_{\text{comm}}\ ,$$

the result would also have been $\text{return } 3$. This is no coincidence: above in Equation (7.12) we derived that

$$\underline{\text{either}}\big(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)\big)$$
$$\equiv_{\Theta_{\text{comm}}} \underline{\text{either}}\big(\underline{\text{either}}(\text{return } 3, \text{return } 2), \text{return } 1\big)\ ,$$

and from the theory-dependent logic we can derive the following rule:

$$\frac{\{\underline{\text{either}} : 2\} \vdash_{\text{c}} \quad M \;\equiv_{\Theta_{\text{comm}}}\; N \quad : \text{nat} \,!\, \{\underline{\text{either}}\}}{\varnothing \vdash_{\text{c}} \quad M \text{ wherealg } \mathcal{A}_{\text{comm}} \;\equiv_{\varnothing}\; N \text{ wherealg } \mathcal{A}_{\text{comm}} \quad : \quad \text{nat} \,!\, \varnothing} \tag{7.13}$$

We will explain below in §7.6.4 how we obtained (7.13). Verify for yourself that (7.13) implies:

$$\begin{aligned} &\underline{\text{either}}(\text{return } 1, \underline{\text{either}}(\text{return } 2, \text{return } 3)) \text{ wherealg } \mathcal{A}_{\text{comm}} \\ &\equiv_{\varnothing} \; \underline{\text{either}}(\underline{\text{either}}(\text{return } 3, \text{return } 2), \text{return } 1) \text{ wherealg } \mathcal{A}_{\text{comm}} \end{aligned}$$

The actual rule in the logic (page 179) contains more premises. Let us expand the relevant premise here to a readable form. In order to get rule (7.13) from the logic, we are required to show the following statement about the definition of $\mathcal{A}_{\text{comm}}$:

$$\begin{aligned} k_1 : (1 \to \text{nat} ! \varnothing), k_2 : (1 \to \text{nat} ! \varnothing) \vdash_{\text{c}} \quad & \Big(\lambda\langle\rangle.\, k_1\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_1.\, \Big(\lambda\langle\rangle.\, k_2\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle \\ \equiv_{\varnothing} \; & \Big(\lambda\langle\rangle.\, k_2\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_1.\, \Big(\lambda\langle\rangle.\, k_1\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle \quad : \text{nat} \,!\, \varnothing \end{aligned}$$

or equivalently, after simplifying using β-rules, that

$$\begin{aligned} k_1 : (1 \to \text{nat} ! \varnothing), k_2 : (1 \to \text{nat} ! \varnothing) \vdash_{\text{c}} \quad & k_1\,\langle\rangle \text{ to } v_1.\, k_2\,\langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle \\ \equiv_{\varnothing} \; & k_2\,\langle\rangle \text{ to } v_1.\, k_1\,\langle\rangle \text{ to } v_2.\, \max\langle v_1, v_2\rangle \quad : \text{nat} \,!\, \varnothing \end{aligned} \tag{7.14}$$

This follows from the commutativity of $\max$, together with the fact that two effectless computations commute. We will prove the latter fact in §8.2 using the base logic.

Before we proceed to explain the general rule of the theory-dependent logic precisely, let us give an intuition to why rule (7.13) may hold.

7.6.3 Intuition: Where operations are defined, the theory can grow

In our language, operation definitions ($-$ wherealg $\mathcal{A}$) introduce new operations with a meaning. According to the equational theory, we can exploit the meaning of these new operations to get additional (in)equalities.

In the previous section we saw an operation definition ($-$ wherealg $\mathcal{A}_{\text{comm}}$) which introduced a "commutative" operation $\underline{\text{either}}$. By *commutative*, we mean that we can "exchange"

$\underline{\text{either}}(M, N)$ for $\underline{\text{either}}(N, M)$ inside the left operand of wherealg, without changing the meaning — that is,

$$\left[\!\left[P\left[\underline{\text{either}}(M, N)\right] \text{ wherealg } \mathcal{A}_{\text{comm}} \right]\!\right] \;=\; \left[\!\left[P\left[\underline{\text{either}}(N, M)\right] \text{ wherealg } \mathcal{A}_{\text{comm}} \right]\!\right] .$$

Semantically, the domain of $[\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!]$ is the set of $\underline{\text{either}}$-trees with leaves in $\mathbb{N}$, quotiented by commutativity of the operations. We can express this *commutativity* property of $[\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!]$ semantically as follows, using m and n to quantify over two semantic computations:

$$\forall m, n : \quad [\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!] \left(\underline{\text{either}}(m, n)\right) \;=\; [\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!] \left(\underline{\text{either}}(n, m)\right)$$

Recall that $[\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!]$ is just a straightforward induction on its argument, applying $k_1 \langle\rangle \text{ to } v_1 . \, k_2 \langle\rangle \text{ to } v_2 . \, \max\langle v_1, v_2 \rangle$ on the operations — let us abbreviate that to $N_{\underline{\text{either}}}$ — so we can simplify the two sides:

$$\begin{aligned} \forall m, n : \quad & [\![N_{\underline{\text{either}}}]\!] \Big([\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!](m), [\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!](n) \Big) \\ & = \; [\![N_{\underline{\text{either}}}]\!] \Big([\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!](n), [\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!](m) \Big) \end{aligned}$$

Clearly, this would follow from $[\![N_{\underline{\text{either}}}]\!]$ being commutative:

$$\forall m, n : \quad [\![N_{\underline{\text{either}}}]\!] \, (m, n) \;=\; [\![N_{\underline{\text{either}}}]\!] \, (n, m)$$

Equivalently we may swap the free variables of $N_{\underline{\text{either}}}$ on a syntactic level:

$$\forall m, n : \quad \left[\!\left[N_{\underline{\text{either}}} \right]\!\right] (m, n) \;=\; \left[\!\left[N_{\underline{\text{either}}}[k_2/k_1, k_1/k_2] \right]\!\right] (m, n)$$

which is, in turn, implied by this syntactic statement:

$$N_{\underline{\text{either}}} \;\equiv_{\varnothing}\; N_{\underline{\text{either}}}[k_2/k_1, k_1/k_2]$$

which is Equation (7.14) above — even though they might not look equivalent at first sight — and which we have argued is true.

Here is again the chain of reasoning that we apply.

- We can use the logic to prove the "commutative" property about $\mathcal{A}_{\text{comm}}$, that is, we can prove $N_{\underline{\text{either}}} \equiv_{\varnothing} N_{\underline{\text{either}}}[k_2/k_1, k_1/k_2]$.
- Therefore, $[\![N_{\underline{\text{either}}}]\!]$ is commutative in its two arguments.
- Therefore, if t_1 and t_2 are two $\underline{\text{either}}$-trees with leaves in nat, differing only by the order of the branches, then $[\![- \text{ wherealg } \mathcal{A}_{\text{comm}}]\!]$ maps them to the same natural number.
- Therefore, if the semantics of $M_1, M_2 : \text{nat}!\{\underline{\text{either}}\}$ are two trees that only differ by the order of the branches, then $[\![M_1 \text{ wherealg } \mathcal{A}_{\text{comm}}]\!] = [\![M_2 \text{ wherealg } \mathcal{A}_{\text{comm}}]\!]$.
- Therefore, we say that $M_1 \text{ wherealg } \mathcal{A}_{\text{comm}} \equiv_{\varnothing} M_2 \text{ wherealg } \mathcal{A}_{\text{comm}}$.

7.6.4 New rule 2: Where operations are defined, the theory can grow

Figure 7.2 on page 179 lists the precise generalised rule of the theory-dependent logic that works for arbitrary operation definitions. Let us explain it here. Assume that we have

- An "ambient" syntactic signature $[\![L; \Delta]\!]$ and a "new" syntactic signature $[\![K; \Delta']\!]$ with no overlapping operation names,
- an "ambient" equational theory Θ on $[\![L; \Delta]\!]$,
- a "new" equational theory Θ' on $[\![L, K; \Delta, \Delta']\!]$,
- two terms M, M' that may use both the operations from $[\![L; \Delta]\!]$ and the operations from $[\![K; \Delta']\!]$,
- and a syntactic algebra $\mathcal{A}$ that defines the new operations $[\![K; \Delta']\!]$.

If $\mathcal{A}$ additionally satisfies a suitable premise, then the rule shows following:

$$\frac{\Delta,\Delta';\ \Gamma \ \vdash_{\text{c}} \quad M \quad \equiv_{\Theta\cup\Theta'} \quad M' \quad : \ A\,!\,(\theta\cup K)}{\Delta \ ;\ \Gamma \ \vdash_{\text{c}} \quad M \text{ wherealg } \mathcal{A} \quad \equiv_{\Theta} \quad M' \text{ wherealg } \mathcal{A} \quad : \ A\,!\,\theta}$$

The additional premise for this is $\Theta \vdash \mathcal{A}$ validates Θ'; we define the meaning of this in the rest of this section. This involves first "expanding" both sides of each axiom of Θ' using $\mathcal{A}$

(Definition 149), and then showing that the $\langle\!\langle - \rangle\!\rangle_{\mathcal{A}}$-expanded sides of each axiom are related in the ambient theory (Definition 152).

In Remark 150 below, we show how we expand Θ_{comm} to obtain the simplified premise (7.14).

Definition 149. Let Δ be an arity assignment on L and let Δ' be an arity assignment on K, with L and K disjoint. Let $\theta \subseteq L$ and let Γ, A be a context and type over L. For every operation $\alpha \in K$, let N_α be a term

$$\forall \alpha \in K: \quad \Delta; \Gamma, (k_i : 1 \to A!\theta) \;\vdash_{\mathrm{c}}\; N_\alpha \;:\; A\,!\,\theta \;\;,$$

so that terms of the form $(-\ \mathrm{wherealg}\ (\alpha\,\vec{k} = N_\alpha)_{\alpha\in K})$ can type check. Let t be an $[\![\theta, K; \Delta, \Delta']\!]$-term over some set of variables I. Then we define the **algebra expansion** $\langle\!\langle \boldsymbol{t} \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}$ to be the computation

$$\Delta;\ \ \Gamma, (\kappa_i : 1 \to A!\theta)_{i\in I} \;\vdash_{\mathrm{c}}\; \langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} \;:\; A\,!\,\theta$$

defined as

$$\begin{aligned}
\langle\!\langle \mathbf{x}_i \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} &= \kappa_i\,\langle\rangle \\
\langle\!\langle \alpha(\vec{t}) \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} &= \alpha\Big(\langle\!\langle t_j \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}\Big)_{j\in\{1,..,\Delta(\alpha)\}} && \text{if } \alpha \in \theta \\
\langle\!\langle \alpha(\vec{t}) \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} &= N_\alpha\Big[\lambda\langle\rangle.\,\langle\!\langle t_j \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} \,/\, k_j\Big]_{j\in\{1,..,\Delta(\alpha)\}} && \text{if } \alpha \in K \quad .
\end{aligned}$$

Remark 150. For our example, we need to expand both sides of $\Theta_{\mathrm{comm}} = \{x, y \vdash \underline{\mathrm{either}}(x, y) = \underline{\mathrm{either}}(y, x)\}$ using $\mathcal{A}_{\mathrm{comm}}$:

$$\kappa_x{:}(1{\to}\mathrm{nat}!\varnothing), \kappa_y{:}(1{\to}\mathrm{nat}!\varnothing) \vdash_{\mathrm{c}} \langle\!\langle \underline{\mathrm{either}}(x, y) \rangle\!\rangle_{\mathcal{A}_{\mathrm{comm}}}$$
$$\stackrel{\text{def}}{=} \Big(\lambda\langle\rangle.\,\kappa_x\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_1.\,\Big(\lambda\langle\rangle.\,\kappa_y\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_2.\,\max\langle v_1, v_2\rangle \;:\; \mathrm{nat}\,!\,\varnothing$$

$$\kappa_x{:}(1{\to}\mathrm{nat}!\varnothing), \kappa_y{:}(1{\to}\mathrm{nat}!\varnothing) \vdash_{\mathrm{c}} \langle\!\langle \underline{\mathrm{either}}(y, x) \rangle\!\rangle_{\mathcal{A}_{\mathrm{comm}}}$$
$$\stackrel{\text{def}}{=} \Big(\lambda\langle\rangle.\,\kappa_y\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_1.\,\Big(\lambda\langle\rangle.\,\kappa_x\,\langle\rangle\Big)\,\langle\rangle \text{ to } v_2.\,\max\langle v_1, v_2\rangle \;:\; \mathrm{nat}\,!\,\varnothing$$

Verify that these terms are equivalent to the left and right hand side of Equation (7.14), up to renaming of context variables.

The following lemma shows that $\langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}$ is a "syntactisation" of what $(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}$ does to substitution instances of t.

Lemma 151. Let $\rho \in [\![\Delta \vdash \Gamma]\!]$. Recall from Definition 120 the definition of interpreting an operation-term in an algebra, written $[\![-]\!]_{(A,a)}$. Let us extend $[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho$ to an $[\![\theta, K; \Delta, \Delta']\!]$-algebra on $[\![\Delta \vdash A!\theta]\!]$ by passing through the operations from $[\![\theta; \Delta]\!]$; write $\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho, \mathrm{id}\right]$ for the resulting algebra. Let us have some $[\![\theta,K; \Delta,\Delta']\!]$-term t over some set of variables I. Then

$$[\![\langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}]\!]_\rho = [\![t]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]} \quad : \quad \left([\![\Delta \vdash A\,!\,\theta]\!]\right)^I \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!] \ .$$

The former is a Per-morphism in both the environment and the tree, therefore the latter is also:

$$[\![\langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}]\!] = \left((\rho, \vec{\kappa}) \mapsto [\![t]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]}(\vec{\kappa})\right)$$
$$: \quad \left([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times \left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\right)^I\right) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \ .$$

Definition 152. Let $L, K, \Delta, \Delta', \Gamma, A, \theta, \vec{N}$ as above in Def. 149. And let Θ be an $[\![L; \Delta]\!]$-equational theory.

- Let furthermore t and u be two $[\![\theta,K; \Delta,\Delta']\!]$-terms over some set of variables I. We say that "$\mathcal{A} = (\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}$ *validates* $I \vdash t = u$ *under* Θ" if $\langle\!\langle t \rangle\!\rangle_{\mathcal{A}} \equiv_\Theta \langle\!\langle u \rangle\!\rangle_{\mathcal{A}}$. Notation: $\boldsymbol{\Theta \vdash \mathcal{A}}$ **validates** $\boldsymbol{I \vdash t = u}$.
- Let Θ' be an $[\![\theta,K; \Delta,\Delta']\!]$-equational theory. We say that "$\mathcal{A} = (\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}$ *validates* Θ' *under* Θ" if $\mathcal{A}$ validates all equations in Θ' under Θ.
  Notation: $\boldsymbol{\Theta \vdash \mathcal{A}}$ **validates** $\boldsymbol{\Theta'}$.

Remark 153. Let $L, K, \Delta, \Delta', \Gamma, A, \theta, \vec{N}$ as above in Def. 149. And let Θ be an $[\![\theta; \Delta]\!]$-equational theory.

- $\mathcal{A} = (\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}$ validates Θ under Θ.
- Let Θ', Θ'' be two $[\![\theta, K; \Delta, \Delta']\!]$-equational theories. Then $\mathcal{A}$ validates $\Theta' \cup \Theta''$ under Θ if it validates Θ' under Θ as well as Θ'' under Θ.

- Let Θ' be an $[\![\theta, K; \Delta, \Delta']\!]$-equational theory, and suppose that $\Theta \vdash \mathcal{A}$ validates Θ'. Then $\Theta \vdash \mathcal{A}$ validates any subset of Θ'.

Lemma 154. If $\Theta \vdash \mathcal{A}$ validates Θ', and if $I \vdash_{\Theta'} t = u$, then $\Theta \vdash \mathcal{A}$ validates $I \vdash t = u$.

Proof. By induction on the proof of $I \vdash_{\Theta'} t = u$. All cases are trivial, although depending on the size of I, this may involve typing judgements about non-finite Γ. As we stated in the introduction to Chapter 6, this is fine. □

Recall the definition of theory closure $\Theta^\dagger$ from Definition 107 on page 149. All signatures that we are considering contain (by definition) only finitary operations, so that we can form closures of theories.

Corollary 155. Let Θ and Θ' be two $\mathcal{S}$-equational theories, and suppose that $\Theta \vdash \mathcal{A}$ validates Θ'. Then $\Theta \vdash \mathcal{A}$ validates $\Theta'^\dagger$.

We never use this fact in the rest of this chapter. As we promised on page 144, everything else in the technical development works for arbitrary size signatures.

7.6.5 Soundness

Notation. Recall that for two sets-with-a-per $\mathbb{A}, \mathbb{B}$, we can form the Per-exponential $\mathbb{A} \to \mathbb{B}$. We will use this often in this section, in particular when referring to its subset $\mathrm{dom}(\mathbb{A} \to \mathbb{B})$ or partial equivalence relation $\sim_{\mathbb{A}\to\mathbb{B}} \subseteq \mathrm{dom}(\mathbb{A} \to \mathbb{B}) \times \mathrm{dom}(\mathbb{A} \to \mathbb{B})$.

Theorem 156 (soundness)**.** The following three statements hold.

1. If the theory-dependent logic shows $\Delta; \Gamma \vdash_{\mathrm{v}} V \equiv_\Theta W : A$,
   then $[\![V]\!] \sim_{[\![\Delta\vdash\Gamma]\!]^{/\Theta} \to [\![\Delta\vdash A]\!]^{/\Theta}} [\![W]\!]$.

2. If the theory-dependent logic shows $\Delta; \Gamma \vdash_{\mathrm{c}} M \equiv_\Theta N : A\,!\,\theta$,
   then $[\![M]\!] \sim_{[\![\Delta\vdash\Gamma]\!]^{/\Theta} \to [\![\Delta\vdash A!\theta]\!]^{/\Theta}} [\![N]\!]$.

3. If $\Theta \vdash \mathcal{A}$ validates $(\Theta \cup \Theta')$, then $[\![\mathcal{A}]\!]^{\mathrm{ext}}$ is a Per-morphism

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} : \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta, \Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/(\Theta\cup\Theta')} \Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \quad . \tag{7.15}$$

Proof. By mutual induction: for the first two statements on the proof of $\equiv_\Theta$; for the last statement on the size of $(\Theta \vdash \mathcal{A} \text{ validates } (\Theta \cup \Theta'))$.

The first two statements go as follows. By Theorem 148, we already know that the semantics of every definable term is in the subset $\mathrm{dom}\left(\llbracket \Delta \vdash \Gamma \rrbracket^{/\Theta} \to \llbracket \Delta \vdash A \rrbracket^{/\Theta}\right)$
(resp. $\mathrm{dom}\left(\llbracket \Delta \vdash \Gamma \rrbracket^{/\Theta} \to \llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}\right)$) corresponding to the Per-exponential. It remains to prove that $(\llbracket V \rrbracket, \llbracket W \rrbracket)$ resp. $(\llbracket M \rrbracket, \llbracket N \rrbracket)$ are in the equivalence relation. We have these cases:

- *Apply axiom of the theory*. We defined $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \subseteq \mathrm{dom}(\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta})^2$ as a free $\Theta|_{\llbracket \theta;\Delta \rrbracket}$-congruence; by definition it satisfies $\Theta|_{\llbracket \theta;\Delta \rrbracket}$. Follows straightforwardly from Lemma 129.

- *Where operations are defined, the theory can grow*. The semantics of $M \text{ wherealg } \mathcal{A}$ is the composition of $(\llbracket \mathcal{A} \rrbracket_\rho)^{\mathrm{ext}}$ with $\llbracket M \rrbracket_\rho$. In our interpretation of the definition of the logic, $(\Theta \vdash \mathcal{A} \text{ validates } (\Theta \cup \Theta'))$ is a premise, so by induction we can apply the last case of the current theorem.

- *Equivalences from the equational theory are inherited*. Soundness for the base logic (Theorem 85) already shows that $\llbracket V \rrbracket = \llbracket W \rrbracket$ (resp. $\llbracket M \rrbracket = \llbracket N \rrbracket$).

- *Symmetry*. $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \subseteq \mathrm{dom}(\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta})^2$ is an equivalence relation.

- *Transitivity*. $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}} \subseteq \mathrm{dom}(\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta})^2$ is an equivalence relation.

- *Compatibility*. Most compatibility rules hold because the semantics of the construct can be built from the bicartesian closed structure of Per, or because $\llbracket \theta; \Delta \rrbracket\!/_{\Theta|_{\llbracket \theta;\Delta \rrbracket}}\text{-tree}(-)$ is a strong monad on Per, or because $\sim_{\llbracket \Delta \vdash A!\theta \rrbracket^{/\Theta}}$ is a congruence on $\llbracket \theta; \Delta \rrbracket\!/_{\Theta|_{\llbracket \theta;\Delta \rrbracket}}\text{-tree}(-)$.

  Only the compatibility of $M \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K}$ cannot be derived this way. Recall that $\llbracket M \text{ wherealg } (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket$ is defined as the composition of $\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}}$ with $\llbracket M \rrbracket$, so we only need to show that

$$\llbracket (\alpha\, \vec{k} = N_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}} \sim_{\llbracket \Delta \vdash \Gamma \rrbracket^{/\Theta} \times \llbracket \Delta, \Delta' \vdash A\,!\,(\theta \cup K) \rrbracket^{/\Theta} \to \llbracket \Delta \vdash A\,!\,\theta \rrbracket} \llbracket (\alpha\, \vec{k} = N'_\alpha)_{\alpha \in K} \rrbracket^{\mathrm{ext}} \quad .$$

They are functions from an environment and a tree to a tree. Recalling the definition of exponentials in Per in Proposition 117 on page 151, we merely need to check that the pair of functions map every "good" environment and "good" tree to related trees. This follows by a straightforward induction on the tree.

The third statement goes as follows. Recall that in Theorem 148 we already showed that

$$[\![\mathcal{A}]\!]^{\mathrm{ext}} : \Big([\![\Delta \vdash \Gamma]\!]^{/\Theta} \times [\![\Delta,\Delta' \vdash A\,!\,(\theta \cup K)]\!]^{/\Theta} \Big) \longrightarrow [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta} \ .$$

The proof of the third statement goes very much analogously. The difference between the two statements is that here in Theorem 156, we are considering a stronger relation on the right argument. Compared to the proof of (7.15), this creates one deficiency, namely in Lemma 146, case 4 on page 171, Eq. (7.9). For $\rho \in \mathrm{dom}\left([\![\Delta \vdash \Gamma]\!]^{/\Theta} \right)$ and an equation

$$(I \vdash t = u) \quad \in \quad (\Theta \cup \Theta')|_{[\![\theta,K;\Delta,\Delta']\!]} \quad \subseteq \quad (\Theta \cup \Theta') \ ,$$

we need to show that

$$\begin{aligned} &(\,[\![\mathcal{A}]\!]_\rho)^{\mathrm{ext}} \Big([\![t]\!]_{[\![\theta,K;\Delta,\Delta']\!]\text{-tree}\left(\mathrm{dom}\left([\![\Delta,\Delta' \vdash A]\!]^{/\Theta}\right)\right)(\vec{\sigma})} \Big) \\ &\qquad \sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}} \ (\,[\![\mathcal{A}]\!]_\rho)^{\mathrm{ext}} \Big([\![u]\!]_{[\![\theta,K;\Delta,\Delta']\!]\text{-tree}\left(\mathrm{dom}\left([\![\Delta,\Delta' \vdash A]\!]^{/\Theta}\right)\right)(\vec{\sigma})} \Big) \ . \end{aligned}$$

Assuming that $\mathcal{A} = (\alpha\,\vec{k} = N_\alpha)_{\alpha \in K}$, we can expand this a bit to:

$$[\![t]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]} \Big(\overrightarrow{(\,[\![\mathcal{A}]\!]_\rho)^{\mathrm{ext}}(\sigma)} \Big) \ \sim_{[\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}} \ [\![u]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]} \Big(\overrightarrow{(\,[\![\mathcal{A}]\!]_\rho)^{\mathrm{ext}}(\sigma)} \Big) \ .$$

But recall that by the induction hypothesis, $\Theta \vdash \mathcal{A}$ validates $(\Theta \cup \Theta')$, and thus $\Theta \vdash \mathcal{A}$ validates $(I \vdash t = u)$, that is:

$$\langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} \equiv_\Theta \langle\!\langle u \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}} \ .$$

We are allowed to use induction on this, and find that

$$[\![\langle\!\langle t \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}]\!]_\rho \ \sim_{\left([\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}\right)^I \to [\![\Delta \vdash A\,!\,\theta]\!]^{/\Theta}} \ [\![\langle\!\langle u \rangle\!\rangle_{(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}}]\!]_\rho \ .$$

Then because of Lemma 151,

$$[\![t]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]}\ \sim_{\left([\![\Delta\vdash A\,!\,\theta]\!]^{/\Theta}\right)^I\to[\![\Delta\vdash A\,!\,\theta]\!]^{/\Theta}}\ [\![u]\!]_{\left[[\![(\alpha\,\vec{k}=N_\alpha)_{\alpha\in K}]\!]_\rho,\,\mathrm{id}\right]}$$

which suffices. □

Corollary 157. If $V \equiv_\varnothing W$, then $[\![V]\!] = [\![W]\!]$. And if $M \equiv_\varnothing N$, then $[\![M]\!] = [\![N]\!]$.

Proof. $[\![\Delta \vdash A\,!\,\theta]\!]^{/\varnothing}$ is the discrete equivalence relation on the entirety of $[\![\Delta \vdash A\,!\,\theta]\!]$. □

Chapter 8

FURTHER EXAMPLES OF PROGRAMS AND REASONING

8.1 Introduction

In this chapter, we expand on example DAO programs given earlier in this thesis, and we give new example programs. We also give examples of how to reason with such programs, using either the base logic $\equiv$ (§6.6) or the stronger theory-dependent logic $\equiv_\Theta$ (§7.6).

- In Section 1.5, we defined a binary operation $\underline{\text{either}}$ that chooses the maximum of both continuations. We alluded to the potential commutativity of it: that $\underline{\text{either}}(M, N)$ is interchangeable with $\underline{\text{either}}(N, M)$ in places that are subject to this operation definition. Throughout §7.6 we expanded on this example, and we showed step by step how to use our logic to prove that $\underline{\text{either}}$ is commutative. The proof relied on the fact that the sequencing of two pure computations is commutative, which follows from our base logic. We give the proof here in §8.2. In fact, we prove a more general statement, namely that pure computations are "central": they commute with any other computation. This follows from our axiom that pure computations are thunkable.

- In Chapter 1, we alluded to an operation-definition for binary state, that works in such a

way that obtaining the state is idempotent, in some sense. We show this, and the other state equations, in this chapter in §8.4.

- In §8.3 we show a third example of an operation definition, namely a funky calculator that uses operations to represent the syntax tree.
- For two further examples, we recall that in Chapter 1 we gave an example with an operation that aborts / causes nonlocal control (page 31), and an operation that evaluates its continuation twice (page 32).

Comparison with handling

Many (but not all) examples of handling transfer to DAO. Bauer and Pretnar [8] and Leijen [46] give a good list of examples for handling. We have already mentioned that we give examples of "nondeterminism" and "state": in §1 we saw how to define $\underline{\text{either}}$ to take the maximum of the results of both branches, and in §8.4 we demonstrate binary state. The aborting on page 31 is analogous to exceptions.

If we add lists and numbers to our language, then more examples translate to DAO: the classic handling examples of "nondeterminism into a list", and "probabilistic choice" will work equally well in DAO.

In Chapter 9, on page 208, we sketch a generalisation of DAO where operations have types as their arity and coarity, and then also

- the "iterators" and the "asynchronous programming" of Leijen [46] translate trivially;
- "general state" works as it does in the handling situation — as long as the type of the stored value mentions only operations that were already present in the context. In the approach sketched on page 208, we have to rule out circularities in the (co)arities of operations to get a well-defined semantics of types.

The multi-threading example in Bauer and Pretnar [8] will not work in this sketched generalisation of DAO, because it uses a storage cell for a value whose type would mention that storage cell, which is circular and therefore forbidden in the sketched approach.

The "transactions", "intercepting standard output", and "read from list" examples from [8] do not translate, because they temporarily give a meaning to ("intercept") an existing operation, which is possible in handling but not in DAO.

8.2 Lemma: effectless computations are central

Let $\Delta;\Gamma$ be a typing context, let A be a type over $\mathrm{dom}(\Delta)$, and let $\Delta;\Gamma \vdash_{\mathrm{c}} M : A\,!\,\varnothing$. Our base logic contains the following axiom, which says that M is "thunkable" [29]:

$$M \text{ to } x.\, \mathrm{return}\ (\lambda\langle\rangle.\, \mathrm{return}\ x) \ \equiv\ \mathrm{return}\ (\lambda\langle\rangle.\, M) \qquad \text{when } M \text{ is of type } A\,!\,\varnothing$$

From the thunkability of M, it follows that M is "central" [63]: it commutes with every other computation. This is not very particular to our logic, but we nevertheless write out the proof for our language for completeness. Here is one way to express centrality of M. Let also B, C be types over $\mathrm{dom}(\Delta)$, let $\theta \subseteq \mathrm{dom}(\Delta)$, and let N, P be computations

$$\begin{aligned} \Delta;&\Gamma && \vdash_{\mathrm{c}} N : B\,!\,\theta \\ \Delta;&\Gamma, x{:}A, y{:}B && \vdash_{\mathrm{c}} P : C\,!\,\theta \end{aligned}$$

Then we prove the following:

$$M \text{ to } x.\, N \text{ to } y.\, P \ \equiv\ N \text{ to } y.\, M \text{ to } x.\, P$$

The proof goes as follows.

$$\begin{aligned} &\Delta;\Gamma \ \vdash_{\mathrm{c}} \\ &\qquad M \text{ to } x.\, N \text{ to } y.\, P \\ &= \qquad \text{(renaming)} \\ &\qquad M \text{ to } m.\, N \text{ to } y.\, P[m/x] \\ &\equiv \qquad (\beta) \\ &\qquad M \text{ to } m.\, N \text{ to } y.\, \Big(\lambda\langle\rangle.\, \mathrm{return}\ m\Big)\langle\rangle \text{ to } x.\, P \\ &\equiv \qquad (\beta) \end{aligned}$$

$$
\begin{array}{ll}
 & M \text{ to } m.\ \text{return } \Big(\lambda\langle\rangle.\ \text{return } m\Big) \text{ to } f.\ N \text{ to } y.\ f\,\langle\rangle \text{ to } x.\ P \\
\equiv & \quad (M \text{ is of type } B\,!\,\varnothing, \text{ therefore thunkable}) \\
 & \text{return } \Big(\lambda\langle\rangle.\ M\Big) \text{ to } f.\ N \text{ to } y.\ f\,\langle\rangle \text{ to } x.\ P \\
\equiv & \quad (\beta) \\
 & N \text{ to } y.\ \Big(\lambda\langle\rangle.\ M\Big)\,\langle\rangle \text{ to } x.\ P \\
\equiv & \quad (\beta) \\
 & N \text{ to } y.\ M \text{ to } x.\ P
\end{array}
$$

8.3 Example: a calculator

In this section, we give a syntactic algebra on natural numbers, that provides calculation functionality. We give this syntactic algebra more as an interesting thought experiment, not because we expect that it will be used much in practice.

We use the DAO language from Chapter 6, augmented with a type nat of natural numbers, natural number constants, and value constructors $\mathrm{succ}(-)$, binary $+$, $\times$, and $\max(-,-)$, that behave according to their mathematical meanings. Let $\Delta;\Gamma$ be any typing context, and let $\theta \subseteq \mathrm{dom}(\Delta)$. Our operations have the following arities:

$$\Delta_{\mathrm{calc}} \quad = \quad \Big\{\ \underline{\mathrm{zero}} : 0,\ \underline{\mathrm{s}} : 1,\ \underline{\mathrm{double}} : 1,\ \underline{\mathrm{add}} : 2,\ \underline{\mathrm{max}} : 2\ \Big\}$$

The syntactic algebra is as follows:

$$
\begin{array}{rl}
\Delta;\Gamma\ \vdash_{\mathrm{alg}}\ \mathcal{A}_{\mathrm{calc}} = \Big\{ & \underline{\mathrm{zero}} = \text{return } 0, \\
 & \underline{\mathrm{s}}\,k = k\,\langle\rangle \text{ to } n.\ \text{return } \mathrm{succ}(n), \\
 & \underline{\mathrm{double}}\,k = k\,\langle\rangle \text{ to } n.\ \text{return } (2 \times n), \\
 & \underline{\mathrm{add}}\,k_1\,k_2 = k_1\,\langle\rangle \text{ to } m.\ k_2\,\langle\rangle \text{ to } n.\ \text{return } (m+n), \\
 & \underline{\mathrm{max}}\,k_1\,k_2 = k_1\,\langle\rangle \text{ to } m.\ k_2\,\langle\rangle \text{ to } n.\ \text{return } \max(m,n) \\
\Big\} & :\ \Delta_{\mathrm{calc}} \Rightarrow \mathrm{nat}\,!\,\theta
\end{array}
$$

Note that $\underline{\max}$ works the same as $\underline{\text{either}}$ in §1.5.

Here are some programs using $\mathcal{A}_{\text{calc}}$.

Example. The following program computes $1 + (1 + (1 + 0)) = 3$:

$$\underline{\text{s}}(\underline{\text{s}}(\underline{\text{s}}(\underline{\text{zero}}()))) \text{ wherealg } \mathcal{A}_{\text{calc}}$$

Example. The following program computes $1 + (1 + (1 + 2)) = 5$:

$$\underline{\text{s}}\Big(\underline{\text{s}}\Big(\underline{\text{add}}\big(\underline{\text{s}}(\underline{\text{zero}}()), \underline{\text{s}}(\underline{\text{s}}(\underline{\text{zero}}()))\big)\Big)\Big) \text{ wherealg } \mathcal{A}_{\text{calc}}$$

To do this calculation, we may exploit the fact that we are working with algebraic effects.

Example. The following program also computes $1 + (1 + (1 + 2)) = 5$:

$$\left\{ \begin{array}{l} \underline{\text{s}}(\text{return } \langle\rangle)\,; \\ \quad \underline{\text{s}}(\text{return } \langle\rangle)\,; \\ \quad \underline{\text{add}}\Big(\underline{\text{s}}\big(\underline{\text{zero}}()\big), \underline{\text{s}}\big(\underline{\text{s}}(\underline{\text{zero}}())\big)\Big) \end{array} \right\} \text{ wherealg } \mathcal{A}_{\text{calc}}$$

If we write $n^\bullet$ for the constant function returning n, then the program may be understood by analogy to the mathematical expression $\text{succ} \circ \text{succ} \circ 3^\bullet = 5^\bullet$.

Example. The following program computes $(1 + 2) + (1 + 2) = 6$:

$$\left\{ \begin{array}{l} \underline{\text{add}}(\text{return } \langle\rangle, \text{return } \langle\rangle)\,; \\ \quad \underline{\text{add}}\Big(\underline{\text{s}}\big(\underline{\text{zero}}()\big), \underline{\text{s}}\big(\underline{\text{s}}(\underline{\text{zero}}())\big)\Big) \end{array} \right\} \text{ wherealg } \mathcal{A}_{\text{calc}}$$

Let us use the categorical notation $\langle f, g\rangle$ for constructing a function of type $A \to B \times C$ from functions $f : A \to B$, $g : A \to C$. Then the program may be understood by analogy to $(+) \circ \langle \text{id}, \text{id}\rangle \circ (1 + 2)^\bullet = 6^\bullet$.

Example. Suppose that Δ contains a binary operation $askUser$, which asks the user for a choice, *left* or *right*. Then the following program asks the user for a choice, and depending on

the choice returns either 7 (=1 + (2×3)) or 5 (=1 + (1+3)).

$$\Big\{\ \underline{\mathrm{s}}(\mathrm{return}\ \langle\rangle)\,;\quad askUser\Big(\underline{\mathrm{double}}(\mathrm{return}\ \langle\rangle), \underline{\mathrm{s}}(\mathrm{return}\ \langle\rangle)\Big)\,;\quad \underline{\mathrm{add}}\Big(\underline{\mathrm{s}}\big(\underline{\mathrm{zero}}()\big), \underline{\mathrm{s}}\big(\underline{\mathrm{s}}(\underline{\mathrm{zero}}())\big)\Big)\ \Big\}\ \ \mathrm{wherealg}\ \mathcal{A}_{\mathrm{calc}}$$

Proposition 158. $\mathcal{A}_{\mathrm{calc}}$ validates $x \vdash \underline{\mathrm{s}}(\underline{\mathrm{s}}(\underline{\mathrm{double}}(x)))) = \underline{\mathrm{double}}(\underline{\mathrm{s}}(x))$ under any theory (for instance, the empty theory).

Proof. The proof requires us to prove that $\langle\!\langle \underline{\mathrm{s}}(\underline{\mathrm{s}}(\underline{\mathrm{double}}(x))))\rangle\!\rangle_{\mathcal{A}_{\mathrm{calc}}} \equiv_{\varnothing} \langle\!\langle \underline{\mathrm{double}}(\underline{\mathrm{s}}(x))\rangle\!\rangle_{\mathcal{A}_{\mathrm{calc}}}$. Abbreviate $\Gamma_x = \{k_x : 1 \to \mathrm{nat}\,!\,\theta\}$.

$$\begin{array}{ll}
\Delta; \Gamma, \Gamma_x \ \vdash_{\mathrm{c}} & \\
\quad \langle\!\langle \underline{\mathrm{s}}(\underline{\mathrm{s}}(\underline{\mathrm{double}}(x))))\rangle\!\rangle_{\mathcal{A}_{\mathrm{calc}}} & \\
\overset{\mathrm{def}}{=} \quad \Big(\lambda\langle\rangle.\,\Big(\lambda\langle\rangle.\,\big(\lambda\langle\rangle.\,k_x\,\langle\rangle\big)\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ (2\times n)\Big)\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ \mathrm{succ}(n)\Big)\,\langle\rangle\ \mathrm{to}\ n. & \\
\quad \mathrm{return}\ \mathrm{succ}(n) & \\
\equiv \quad \text{(administrative)} & \\
\quad k_x\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ \mathrm{succ}(\mathrm{succ}(2\times n)) & \\
\equiv \quad \text{(mathematics)} & \\
\quad k_x\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ (2\times \mathrm{succ}(n)) & \\
\equiv \quad \text{(administrative)} & \\
\quad \Big(\lambda\langle\rangle.\,\big(\lambda\langle\rangle.\,k_x\,\langle\rangle\big)\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ \mathrm{succ}(n)\Big)\,\langle\rangle\ \mathrm{to}\ n.\,\mathrm{return}\ (2\times n) & \\
\overset{\mathrm{def}}{=} \quad \langle\!\langle \underline{\mathrm{double}}(\underline{\mathrm{s}}(x))\rangle\!\rangle_{\mathcal{A}_{\mathrm{calc}}} \quad : \quad (\mathrm{bool}\to A\,!\,\theta)\,!\,\theta & \qquad \square
\end{array}$$

Example. Proposition 158 allows us to quickly conclude equations such as

$$\underline{\mathrm{s}}(\underline{\mathrm{s}}(\underline{\mathrm{double}}(\underline{\mathrm{s}}(\underline{\mathrm{zero}}()))))\ \ \mathrm{wherealg}\ \mathcal{A}_{\mathrm{calc}} \equiv_{\varnothing} \underline{\mathrm{double}}(\underline{\mathrm{s}}(\underline{\mathrm{s}}(\underline{\mathrm{zero}}())))\ \mathrm{wherealg}\ \mathcal{A}_{\mathrm{calc}}$$

$$(\ \equiv\ \ \mathrm{return}\ (2\times \mathrm{succ}(\mathrm{succ}(0)))$$

$$\equiv\ \ \mathrm{return}\ 4\ ) \qquad ,$$

as well as more complicated equations.

8.4 Example: binary state

8.4.1 Introduction, definition, usage

In §1.7, we alluded to an operation-definition for binary state, with

- two unary operations $\underline{\mathrm{setstate}_0}(-), \underline{\mathrm{setstate}_1}(-)$, and
- a binary operation $\underline{\mathrm{getstate}}(-,-)$.

The intuition is that $\underline{\mathrm{setstate}_0}, \underline{\mathrm{setstate}_1}$ "set the state" to 0 and 1, respectively, and $\underline{\mathrm{getstate}}(M, N)$ is equivalent to M when the state was last set to 0, and to N when the state was last set to 1. In this section, we explain and analyse binary state in DAO.

Abbreviate the signature as $\Delta_{\mathrm{state}} = \left\{\underline{\mathrm{getstate}} : 2, \underline{\mathrm{setstate}_0} : 1, \underline{\mathrm{setstate}_1} : 1\right\}$ (and let $L_{\mathrm{state}} = |\Delta_{\mathrm{state}}|$). Let Δ be an arity assignment on L, let Γ and A be a value context and type over L, and let $\theta \subseteq L$ be some ambient effects. **We fix $\boldsymbol{L}$, $\boldsymbol{\Delta}$, $\boldsymbol{\Gamma}$, $\boldsymbol{A}$, $\boldsymbol{\theta}$ throughout this section.** For convenience and clarity, we assume some type bool, which could be just $1{+}1$, with values $\mathrm{true}, \mathrm{false}$ and pattern matching, spelled $\mathrm{if}\, V\, \mathrm{then}\, M\, \mathrm{else}\, N$. We give our "binary state algebra in $\Delta; \Gamma$ for type A":

$$\Delta; \Gamma \vdash_{\mathrm{alg}} \mathcal{A}_{\mathrm{state}} \stackrel{\mathrm{def}}{=} \begin{pmatrix} \underline{\mathrm{getstate}}\, k_1\, k_2 = \Big(\mathrm{return}\ \lambda b.\, \mathrm{if}\, b\, \mathrm{then}\, (k_2\, \langle\rangle\ \mathrm{to}\ f.\, f\, \mathrm{true})\, \mathrm{else}\, (k_1\, \langle\rangle\ \mathrm{to}\ f.\, f\, \mathrm{false})\Big), \\ \underline{\mathrm{setstate}_0}\, k = \Big(\mathrm{return}\ \lambda b.\, k\, \langle\rangle\ \mathrm{to}\ f.\, f\, \mathrm{false}\Big), \\ \underline{\mathrm{setstate}_1}\, k = \Big(\mathrm{return}\ \lambda b.\, k\, \langle\rangle\ \mathrm{to}\ f.\, f\, \mathrm{true}\Big) \end{pmatrix} : \Delta_{\mathrm{state}} \Rightarrow (\mathrm{bool} \to A\,!\,\theta)\,!\,\theta$$

Remark. The type of $\mathcal{A}_{\mathrm{state}}$ looks a bit funny, with the double $!\,\theta$, and its definition is convoluted. This is an unfortunate consequence of needing an algebra on a function type in fine-grain call-by-value, where function types are not computation types. Handling in FGCBV has the

same inconvenience, see for instance Bauer and Pretnar [7] §7. This issue would not arise if we took call-by-push-value for our base language, see for instance the analogous state handler in CBPV in Plotkin and Pretnar [62].

State algebras are a tad convoluted to use in practice, because they require an initial state and because they are an algebra on a function type. The binary state algebra can be used as follows. Let M be a computation

$$\Delta, \Delta_{\mathrm{state}}; \Gamma \quad \vdash_{\mathrm{c}} M \quad : \quad A \,!\, (\theta \cup L_{\mathrm{state}}) \quad ,$$

then the following "runs" M with initial state b_{init}:

$$\Delta; \Gamma \vdash_{\mathrm{c}} \quad \Big(M \text{ to } \mathit{result}.\, \mathrm{return}\ (\lambda b.\, \mathrm{return}\ \mathit{result}) \quad \mathrm{wherealg}\ \mathcal{A}_{\mathrm{state}} \Big) \text{ to } f.\ f\, b_{init} \quad : \quad A \,!\, \theta \tag{8.1}$$

Here is an example usage. Assume that we have a type nat of natural numbers with addition as a value constructor. If we plug in the following computation M_1 into Equation (8.1) and run it with initial state false, then the result is 7:

$$\begin{aligned} \cdot; \cdot \vdash_{\mathrm{c}} \quad M_1 = \Big\{ & \underline{\mathrm{getstate}}(\mathrm{return}\ 3, \mathrm{return}\ 4) \text{ to } x. \\ & \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle); \\ & \underline{\mathrm{getstate}}(\mathrm{return}\ 3, \mathrm{return}\ 4) \text{ to } y. \\ & \mathrm{return}\ (x + y) \\ \Big\} \quad & : \mathrm{nat} \,!\, L_{\mathrm{state}} \end{aligned}$$

8.4.2 Properties

Again, we fix a typing context $\Delta; \Gamma$, a type A over $L = \mathrm{dom}(\Delta)$, and some set of ambient effects $\theta \subseteq L$. Additionally, fix a $[\![L; \Delta]\!]$-equational theory Θ, for instance $\Theta = \varnothing$.

We claim that $\mathcal{A}_{\mathrm{state}}$ validates the following eight equations.

- $x, y \ \vdash \ \underline{\mathrm{setstate}_0}(\underline{\mathrm{getstate}}(x, y)) \ = \ \underline{\mathrm{setstate}_0}(x)$
- $x, y \ \vdash \ \underline{\mathrm{setstate}_1}(\underline{\mathrm{getstate}}(x, y)) \ = \ \underline{\mathrm{setstate}_1}(y)$

- $x, y \vdash \underline{\mathrm{getstate}}(\underline{\mathrm{setstate}_0}(x), \underline{\mathrm{setstate}_1}(y)) = \underline{\mathrm{getstate}}(x, y)$
- $x, y, z, w \vdash \underline{\mathrm{getstate}}(\underline{\mathrm{getstate}}(x, y), \underline{\mathrm{getstate}}(z, w)) = \underline{\mathrm{getstate}}(x, w)$
- $x \vdash \underline{\mathrm{setstate}_0}(\underline{\mathrm{setstate}_1}(x)) = \underline{\mathrm{setstate}_1}(x)$, and analogously the other 3 combinations of $\underline{\mathrm{setstate}_0}$ and $\underline{\mathrm{setstate}_1}$

Only when $\theta = \varnothing$ does $\mathcal{A}_{\mathrm{state}}$ also validate the following equation:

$$x \vdash \underline{\mathrm{getstate}}(x, x) = x$$

This is because they are distinguishable when θ is nonempty: the computation of type $(\mathrm{bool} \to A!\theta)!\theta$ may either immediately return a function or it may first call an ambient operation.

These nine equations are a complete theory for boolean state: they are equivalent to axioms GS1–4 in Staton [68].

Theorem 159. $\Theta \vdash \mathcal{A}_{\mathrm{state}}$ validates the theory consisting of the top 8 equations. If $\theta = \varnothing$ then $\Theta \vdash \mathcal{A}_{\mathrm{state}}$ validates the theory consisting of all 9 equations.

The rest of this section consists of the proof. In each case, we expand using Definition 152 and we calculate.

Equation 1 & 2: set followed by get

We prove that $\mathcal{A}_{\mathrm{state}}$ validates equation 1 under Θ, namely that $x, y \vdash \underline{\mathrm{setstate}_0}(\underline{\mathrm{getstate}}(x, y)) = \underline{\mathrm{setstate}_0}(x)$. Equation 2 is analogous. Abbreviate $\Gamma_{xy} = \big(k_x : (1 \to (\mathrm{bool} \to A!\theta)!\theta), k_y : (1 \to (\mathrm{bool} \to A!\theta)!\theta)\big)$.

$$\begin{aligned}
&\Delta; \Gamma, \Gamma_{xy} \quad \vdash_{\mathrm{c}} \\
&\qquad \langle\!\langle \underline{\mathrm{setstate}_0}(\underline{\mathrm{getstate}}(x, y)) \rangle\!\rangle_{\mathcal{A}_{\mathrm{state}}} \\
&\overset{\mathrm{def}}{=} \\
&\qquad \mathrm{return}\ \lambda b.\ \left(\begin{array}{r} \lambda\langle\rangle.\mathrm{return}\ \lambda b.\ \mathrm{if}\ b\ \mathrm{then}\ \big((\lambda\langle\rangle.\, k_y\ \langle\rangle)\ \langle\rangle\ \mathrm{to}\ f.\, f\ \mathrm{true}\big) \\ \mathrm{else}\ \ \big((\lambda\langle\rangle.\, k_x\ \langle\rangle)\ \langle\rangle\ \mathrm{to}\ f.\, f\ \mathrm{false}\big) \end{array} \right)\ \langle\rangle\ \mathrm{to}\ f.\, f\ \mathrm{false} \\
&\equiv \qquad \text{(administrative)}
\end{aligned}$$

$$\text{return } \lambda b. \begin{pmatrix} \lambda b.\, \text{if } b \,\text{then } \big(k_y \,\langle\rangle \text{ to } f.\, f \,\text{true}\big) \\ \qquad\quad \text{else } \big(k_x \,\langle\rangle \text{ to } f.\, f \,\text{false}\big) \end{pmatrix} \text{false}$$

$\equiv$ (administrative)

$$\text{return } \lambda b. \big(k_x \,\langle\rangle \text{ to } f.\, f \,\text{false}\big)$$

$\equiv$ (administrative)

$$\text{return } \lambda b. \big((\lambda\langle\rangle.\, k_x \,\langle\rangle)\,\langle\rangle \text{ to } f.\, f \,\text{false}\big)$$

$$\stackrel{\text{def}}{=} \langle\!\langle \underline{\text{setstate}_0}(x) \rangle\!\rangle_{\mathcal{A}_{\text{state}}} \quad : \quad (\text{bool} \to A \,!\, \theta) \,!\, \theta$$

Equation 3: get followed by set

We prove that $x, y \;\vdash\; \underline{\text{getstate}}(\underline{\text{setstate}_0}(x), \underline{\text{setstate}_1}(y)) \;=\; \underline{\text{getstate}}(x, y)$ under Θ. Let Γ_{xy} be as above.

$\Delta; \Gamma, \Gamma_{xy} \;\vdash_{\text{c}}$

$$\langle\!\langle \underline{\text{getstate}}(\underline{\text{setstate}_0}(x), \underline{\text{setstate}_1}(y)) \rangle\!\rangle_{\mathcal{A}_{\text{state}}}$$

$\stackrel{\text{def}}{=}$

$$\text{return } \lambda b. \begin{pmatrix} \text{if } b \,\text{then } \Big(\lambda\langle\rangle.\, \text{return } \big(\lambda b.\, (\lambda\langle\rangle.\, k_y \,\langle\rangle)\,\langle\rangle \text{ to } f.\, f \,\text{true}\big)\Big)\,\langle\rangle \text{ to } f.\, f \,\text{true} \\ \quad \text{else } \Big(\lambda\langle\rangle.\, \text{return } \big(\lambda b.\, (\lambda\langle\rangle.\, k_x \,\langle\rangle)\,\langle\rangle \text{ to } f.\, f \,\text{false}\big)\Big)\,\langle\rangle \text{ to } f.\, f \,\text{false} \end{pmatrix}$$

$\equiv$ (administrative)

$$\text{return } \lambda b. \begin{pmatrix} \text{if } b \,\text{then } k_y \,\langle\rangle \text{ to } f.\, f \,\text{true} \\ \quad \text{else } \; k_x \,\langle\rangle \text{ to } f.\, f \,\text{false} \end{pmatrix}$$

$\equiv$ (administrative)

$$\text{return } \lambda b. \begin{pmatrix} \text{if } b \,\text{then } (\lambda\langle\rangle.\, k_y \,\langle\rangle)\,\langle\rangle \text{ to } f.\, f \,\text{true} \\ \quad \text{else } \; (\lambda\langle\rangle.\, k_x \,\langle\rangle)\,\langle\rangle \text{ to } f.\, f \,\text{false} \end{pmatrix}$$

$$\stackrel{\text{def}}{=} \langle\!\langle \underline{\text{getstate}}(x, y) \rangle\!\rangle_{\mathcal{A}_{\text{state}}} \quad : \quad (\text{bool} \to A \,!\, \theta) \,!\, \theta$$

Equation 4: get followed by get

We prove that $x, y, z, w \;\vdash\; \underline{\text{getstate}}(\underline{\text{getstate}}(x, y), \underline{\text{getstate}}(z, w)) \;=\; \underline{\text{getstate}}(x, w)$ under Θ. Define Γ_{xyzw} to consist of variables k_x, k_y, k_z, k_w, all of type $1 \to (\text{bool} \to A!\theta)!\theta$.

$$
\begin{array}{ll}
& \Delta; \Gamma, \Gamma_{xyzw} \;\vdash_{\text{c}} \\
& \quad \langle\!\langle \underline{\text{getstate}}(\underline{\text{getstate}}(x, y), \underline{\text{getstate}}(z, w)) \rangle\!\rangle_{\mathcal{A}_{\text{state}}} \\
\stackrel{\text{def}}{=} & \\
& \text{return } \lambda b.\, \text{if } b \,\text{then } \left(\begin{array}{r} \lambda\langle\rangle.\, \text{return } \lambda b.\, \text{if } b \,\text{then } \big[(\lambda\langle\rangle.\, k_w\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{true}\big] \\ \text{else } \big[(\lambda\langle\rangle.\, k_z\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{false}\big] \end{array} \right) \langle\rangle \text{ to } f.\, f \,\text{true} \\
& \qquad \text{else } \left(\begin{array}{r} \lambda\langle\rangle.\, \text{return } \lambda b.\, \text{if } b \,\text{then } \big[(\lambda\langle\rangle.\, k_y\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{true}\big] \\ \text{else } \big[(\lambda\langle\rangle.\, k_x\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{false}\big] \end{array} \right) \langle\rangle \text{ to } f.\, f \,\text{false} \\
\equiv & \quad \text{(administrative)} \\
& \quad \text{return } \lambda b.\, \text{if } b \,\text{then } \left(\begin{array}{r} \text{if true then } \big[(\lambda\langle\rangle.\, k_w\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{true}\big] \\ \text{else } \big[(\lambda\langle\rangle.\, k_z\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{false}\big] \end{array} \right) \\
& \qquad \text{else } \left(\begin{array}{r} \text{if false then } \big[(\lambda\langle\rangle.\, k_y\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{true}\big] \\ \text{else } \big[(\lambda\langle\rangle.\, k_x\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{false}\big] \end{array} \right) \\
\equiv & \quad \text{(administrative)} \\
& \quad \text{return } \lambda b.\, \text{if } b \,\text{then } [(\lambda\langle\rangle.\, k_w\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{true}] \\
& \qquad \text{else } [(\lambda\langle\rangle.\, k_x\, \langle\rangle)\, \langle\rangle \text{ to } f.\, f \,\text{false}] \\
\stackrel{\text{def}}{=} & \langle\!\langle \underline{\text{getstate}}(x, w) \rangle\!\rangle_{\mathcal{A}_{\text{state}}} \qquad : \quad (\text{bool} \to A\,!\,\theta)\,!\,\theta
\end{array}
$$

Remark. In Chapter 1, we saw how we can use this to reason about computations that are subject to $\mathcal{A}_{\text{state}}$. Let $M[\,]$ be a computation with a computation hole of type bool, and abbreviate

$$getstate = \underline{\text{getstate}}(\text{return false}, \text{return true}) \;\; .$$

Then the following holds by straightforward application of the logic in §7.6:

$$M[getstate; getstate] \text{ wherealg } \mathcal{A}_{\text{state}} \quad \equiv_{\varnothing} \quad M[getstate] \text{ wherealg } \mathcal{A}_{\text{state}}$$

Remark 160. In an analogous language with handling instead of DAO, one might wish to obtain an analogous proposition, something like:

$$\lightning \quad M[getstate; getstate] \text{ handled with } \mathcal{H}_{\text{state}} \quad = \quad M[getstate] \text{ handled with } \mathcal{H}_{\text{state}} \quad \lightning$$

But this is not possible, for two reasons.

Reason 1: operations may be re-handled. We can define a "verbose state" handler which handles $\underline{\text{getstate}}(-,-)$ to output a message to the user and proceed in the left continuation. Then

$$\begin{aligned} &\Big(getstate; getstate \text{ handled with } \mathcal{H}_{\text{verbstate}}\Big) \text{ handled with } \mathcal{H}_{\text{state}} \\ \neq\ &\Big(getstate \text{ handled with } \mathcal{H}_{\text{verbstate}}\Big) \text{ handled with } \mathcal{H}_{\text{state}} \end{aligned}$$

because the top computation prints twice and the bottom computation prints only once.

Reason 2: operations may not be handled in practice. If we immediately return a lambda containing $\underline{\text{getstate}}$, then that operation is not handled in practice:

$$\begin{aligned} \text{return } (\lambda\langle\rangle.\, getstate; getstate) \text{ handled with } \mathcal{H}_{\text{state}} \quad &= \quad \text{return } (\lambda\langle\rangle.\, getstate; getstate) \\ \text{return } (\lambda\langle\rangle.\, getstate) \text{ handled with } \mathcal{H}_{\text{state}} \quad &= \quad \text{return } (\lambda\langle\rangle.\, getstate) \end{aligned}$$

And the right sides are not equal.

Equations 5–8: set followed by set

To cover the 4 equations that we must prove, let p and q each stand for a bit $\in \{0, 1\}$. Then we will prove that $x \vdash \underline{\text{setstate}_p}(\underline{\text{setstate}_q}(x)) = \underline{\text{setstate}_q}(x)$ under Θ. We abbreviate

$\Gamma_x = \big(k_x : (1 \to (\mathrm{bool} \to A!\theta)!\theta)\big)$. Let p', q' be the boolean values corresponding to p, q.

$$
\begin{array}{lll}
\Delta; \Gamma, \Gamma_x & \vdash_{\mathrm{c}} & \\
& \langle\!\langle \underline{\mathrm{setstate}_p}(\underline{\mathrm{setstate}_q}(x)) \rangle\!\rangle_{\mathcal{A}_{\mathrm{state}}} & \\
\stackrel{\text{def}}{=} & \mathrm{return}\ \lambda b.\ \Big(\Big(\lambda\langle\rangle.\ \mathrm{return}\ \lambda b.\ \big((\lambda\langle\rangle.\ k_x\ \langle\rangle)\ \langle\rangle\ \mathrm{to}\ f.\ f\ q'\big)\Big)\ \langle\rangle\ \mathrm{to}\ f.\ f\ p'\Big) & \\
\equiv & \quad \text{(administrative)} & \\
& \mathrm{return}\ \lambda b.\ \Big(\big(\lambda b.\ (k_x\ \langle\rangle\ \mathrm{to}\ f.\ f\ q')\big)\ p'\Big) & \\
\equiv & \quad (\beta \text{ on } \to) & \\
& \mathrm{return}\ \lambda b.\ \big(k_x\ \langle\rangle\ \mathrm{to}\ f.\ f\ q'\big) & \\
\equiv & \quad \text{(administrative)} & \\
& \mathrm{return}\ \lambda b.\ \big((\lambda\langle\rangle.\ k_x\ \langle\rangle)\ \langle\rangle\ \mathrm{to}\ f.\ f\ q'\big) & \\
\stackrel{\text{def}}{=} & \langle\!\langle \underline{\mathrm{setstate}_q}(x) \rangle\!\rangle_{\mathcal{A}_{\mathrm{state}}} \qquad : \quad (\mathrm{bool} \to A\,!\,\theta)\,!\,\theta &
\end{array}
$$

Equation 9: ignored get

We prove that $x \vdash \underline{\mathrm{getstate}}(x, x) = x$ under Θ. Let Γ_x be as above. We must assume that $\theta = \varnothing$.

$$
\begin{array}{lll}
\Delta; \Gamma, \Gamma_x & \vdash_{\mathrm{c}} & \\
& \langle\!\langle \underline{\mathrm{getstate}}(x, x) \rangle\!\rangle_{\mathcal{A}_{\mathrm{state}}} & \\
\stackrel{\text{def}}{=} & \mathrm{return}\ \lambda b.\ \mathrm{if}\ b\ \mathrm{then}\ (k_x\ \langle\rangle\ \mathrm{to}\ f.\ f\ \mathrm{true})\ \mathrm{else}\ (k_x\ \langle\rangle\ \mathrm{to}\ f.\ f\ \mathrm{false}) & \\
\equiv & \quad (\eta \text{ on } \mathrm{bool}) & \\
& \mathrm{return}\ \lambda b.\ (k_x\ \langle\rangle\ \mathrm{to}\ f.\ f\ b) & \\
\equiv & \quad (\beta \text{ on } \to) & \\
& \mathrm{return}\ \lambda b.\ \Big(\big(\lambda\langle\rangle.\ k_x\ \langle\rangle\big)\ \langle\rangle\ \mathrm{to}\ f.\ f\ b)\Big) & \\
\equiv & \quad (\beta) & \\
& \mathrm{return}\ \big(\lambda\langle\rangle.\ k_x\ \langle\rangle\big)\ \mathrm{to}\ g.\ \mathrm{return}\ \lambda b.\ \Big(g\ \langle\rangle\ \mathrm{to}\ f.\ f\ b)\Big) & \\
\equiv & \quad (k_x\ \langle\rangle \text{ is of type } A!\varnothing, \text{ therefore thunkable}) &
\end{array}
$$

$$k_x\,\langle\rangle \text{ to } y.\,\text{return } \Big(\lambda\langle\rangle.\,\text{return } y\Big) \text{ to } g.\,\text{return } \lambda b.\,\Big(g\,\langle\rangle \text{ to } f.\,f\,b)\Big)$$

$\equiv$ (administrative)

$$k_x\,\langle\rangle \text{ to } y.\,\text{return } \lambda b.\,(y\,b)$$

$\equiv$ (η on $\to$)

$$k_x\,\langle\rangle \text{ to } y.\,\text{return } y$$

$\equiv$ (administrative)

$$k_x\,\langle\rangle$$

$\stackrel{\text{def}}{=}$ $\langle\!\langle x \rangle\!\rangle_{\mathcal{A}_{\text{state}}} \quad : \quad (\text{bool} \to A\,!\,\theta)\,!\,\theta$

Chapter 9

CONCLUSION AND FUTURE DIRECTIONS

This thesis presents *defined algebraic operations* (DAO), a new model of programming that falls broadly between imperative and purely functional programming. We argued in §1 that DAO promises to solve issues with both. It does so by combining algebraic effect handling with lexical binding.

Algebraic effect handling enables a "communication across stack frames", and has been well-studied in the literature, as we laid out in §2. Lexical binding in DAO adds three advantages compared to handling:

1. *In the DAO setting, the type system can catch an additional class of bugs.* We demonstrated this informally in §1.2.

2. *DAO automatically avoids name clashes when writing higher-order programs with non-local control.* We demonstrated this informally in §1.3. As we mention in particular, operation definition commutes with sequencing

$$\Big(M \text{ to } x.\, N\Big) \text{ wherealg } \mathcal{A} \quad \equiv \quad M \text{ to } x.\, \Big(N \text{ wherealg } \mathcal{A}\Big)$$

when both the left and right hand side are well-typed. And defined operations that are unused may as well not be defined at all (Lemma 82):

$$M \quad \equiv \quad M \text{ wherealg } \mathcal{A}$$

3. *DAO allows a strong "theory-dependent" logic, which gives us additional insight into programs with defined operations.* For instance, in the example of §8.4, we saw that under an operation definition for binary state, almost all of the state equations hold.

The heart of this thesis is in Chapters 6 and 7, where we formally develop a lambda-like calculus with DAO, and develop the base logic $\equiv$ and the theory-dependent logic $\equiv_\Theta$, and prove them sound with respect to our denotational semantics.

DAO vs. handling

Plotkin and Pretnar [62] had already studied *handlers* in combination with equational theories: They assumed a global equational theory, and defined a handling language where terms receive an interpretation if[1] the handlers in it preserve the axioms. In this thesis, we show that for *defined operations* we can reverse the story: first we define what programs and operation definitions are, and then we define a notion of validating a theory. All programs are valid, and programs have a semantics *at all times* that corresponds to the operational semantics. Contrary to the handling situation, we push the reasoning and modularity to the logic level: we may prove that our operation definition validates certain axioms.

This reversal is possible because our definitions create new names: if we are creating a program from an operation definition for α, β, γ and a program over those operations, then in the second part it is always okay to define more operations $\delta, \varepsilon, \zeta$, with arbitrary definitions, because they are *new* operations. If we consider axioms on α, β, γ, then this doesn't make the definition of $\delta, \varepsilon, \zeta$ invalid. To the contrary: the scope of definition of α, β, γ allows additional definitions (namely, the definitions may now mention α, β, γ) and the axioms for α, β, γ may be useful in showing that a definition for $\delta, \varepsilon, \zeta$ also validates a set of axioms.

DAO seems to admit stronger reasoning principles than handling. In §7.6 we saw how to use operation definitions to reason about programs that use operations, but even the simpler logic

[1] *If and when*: in [62], a handler receives an interpretation *in a given environment* when it forms a model in that environment. A handled computation receives an interpretation in a given environment when both the handler and the computation receive an interpretation in that environment.

in Figure 6.4 of §6.6 contains two rules that do not hold for handling:

$$M \text{ wherealg } \mathcal{A} \quad \equiv \quad M \text{ wherealg } (\mathcal{A}, \mathcal{B})$$
$$(M \text{ to } x.\, N) \text{ wherealg } \mathcal{A} \quad \equiv \quad M \text{ to } x.\, (N \text{ wherealg } \mathcal{A})$$

Because the latter equation does not hold in handling, it is common to use a ternary handling construct à la Benton and Kennedy [9]. For DAO, a binary construct suffices.

Reasoning for DAO is also *more incremental* than for handling. For a DAO program, we may play with the basic logic for a while, then we might find that we want some property about operations, proceed to prove that the definition for those operations satisfy something, and so on. With handling, theories, and program correctness à la Plotkin and Pretnar [62] this is not so easy: if our program is correct for a certain theory, then it may not be correct for a *weaker* theory! Here is a simple example. Consider two binary operations α, β, and a handler that changes the meaning of α to be β and vice versa. This handler is correct for the empty theory, and for the theory where both operations are commutative, but not for the theory where only one operation is commutative.

This situation can never happen in our DAO logic: all programs always have an interpretation, and $\equiv_\Theta$ is monotonic in Θ. If we reason a bit and gain knowledge about axioms validated by an operation definition, then this knowledge never gives us *obligations* because we will never handle those operations another way.

Yet as we conjectured in §2.4, in a way, DAO "just" seems to be a subset of handling, albeit a rather useful one.

There are certainly more things to be understood about the connection between DAO and handling. Firstly, it might be useful to have a language with both operations that are defined and operations that are handled, so that programmers can choose the most appropriate tool for the job. Secondly: if we combine DAO with recursion, then it seems that we can write programs that put an arbitrary number of different operations on the stack at the same time, depending on context. Can we still macro-translate such programs to handling, and what precise sense of handling is required?

Adding effect rows into the mix, as is the case in the "core" language of Zhang and Myers [71][2], may make this a rather nontrivial question, and seems to necessitate name generation for handleable operations. Can we draw some relationship using contextual equivalence? Or even a denotational relationship?

Zhang and Myers [71] raise another obvious question: do our results translate to their (larger) "core" language? What exactly is the relationship between DAO and (an appropriate notion of) delimited continuations? Perhaps our logic becomes trivial when viewed through the lens of delimited continuations.

Richer notions of operations

The handling literature has a richer notion of operations: it is very common there that operations have a *type* arity (rather than finite arity, *corresponds to the return type of the generic effect*) and also a type coarity (*corresponds to the argument of the generic effect*). But we speculate that this thesis already provides the necessary ingredients for *defining* such a richer notion of operations; let us sketch it. There is a circularity like in Chapter 5: operation names can now appear in the arity and coarity of operations. We allow this, as long as there is no infinite path consisting of an operation appearing in the (co)arity of an operation which appears in the (co)arity of an operation, and so forth: this is the same requirement as in §5. Then to give semantics of types, we use equations similar to in §6. They will form a definition of the semantics of types, by the same argument as in §5. The rest is automatic.

Less obvious is how to carry over the reasoning about operation definitions. We are only aware of notions of effect theories in the literature on operations with *ground type* arity/coarity, but ideally this restriction would be lifted to allow a notion of effect theory on operations of any (co)arity — as long as the (co)arities are well-founded, as above. Surely some operation definitions for this more liberal kind of operation admit interesting equations on the (function-valued) arguments and results of those operations.

[2] Recall that in §2.3, we observed that Zhang and Myers [71] also describe a form of defined algebraic operations, although they just call it *handling*.

Ahman [4] gave a fibred notions of operations. And scoped operations (e.g. Pirog et al. [56]) present a richer form of operations on a different axis. Can either of them be combined in any way with DAO?

Richer underlying language

Some handling languages are based on call-by-push-value [48]. It would be interesting to see DAO in this setting; the author suspects little difficulty. We also suspect little trouble with recursion, as we mentioned in §2.1.

Above we mentioned the potential addition of effect rows, which are commonly used with handling, albeit typically without denotational semantics. A direct-style denotational semantics for effect rows with either handling or DAO would surely be very insightful.

Richer notions of theory

The effect theories in this thesis and papers such as Plotkin and Pretnar [62] (basically, Lawvere theories) are nice, but there are things they cannot express. Consider for instance a monotone boolean state, with operations $\underline{\mathrm{getstate}}$ and $\underline{\mathrm{setstate}_1}$, but without $\underline{\mathrm{setstate}_0}$ so that the state can only increase. Then surely $\underline{\mathrm{setstate}_1}$ is idempotent,

$$\underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle);\ \ \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle)\ \ =\ \ \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle)$$

which we can capture in an effect theory. But it is also idempotent "over longer spans of time",

$$\forall M:\qquad \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle);\ \ M;\ \ \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle)\ \ =\ \ \underline{\mathrm{setstate}_1}(\mathrm{return}\ \langle\rangle);\ \ M$$

and this cannot be captured in an effect theory. This kind of optimisation is considered by Kammar and Plotkin [41]; perhaps their work extends to some degree to DAO.

Biernacki et al. [12] developed a logical relations technique with which they derive rather strong results about programs with handlers. Perhaps these results can be translated in some way to DAO.

Coherence and labelled iteration

In this thesis, we gave an account of defined algebraic operations, but also of labelled iteration and the coherence of the semantics of relatively simple lambda calculi. We discuss these more in §3.4 and §4.4, respectively. For more discussion on labelled iteration, we refer to further work by Goncharov et al. [32].

This concludes our conclusion.

BIBLIOGRAPHY

[1] Aczel, P., Adámek, J., Milius, S., and Velebil, J. 2003. Infinite trees and completely iterative theories: a coalgebraic view. *Theoretical Computer Science* 300(1–3):1–45. ISSN 0304-3975. doi:10.1016/S0304-3975(02)00728-4.

[2] Adámek, J., Milius, S., and Velebil, J. 2006. Elgot Algebras. *Logical Methods in Computer Science* 2(5). ISSN 18605974. doi:10.2168/LMCS-2(5:4)2006.

[3] Adámek, J., Milius, S., and Velebil, J. 2011. Elgot theories: a new perspective on the equational properties of iteration. *Mathematical Structures in Computer Science* 21(Special Issue 02):417–480. ISSN 1469-8072. doi:10.1017/S0960129510000496.

[4] Ahman, D. 2017. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2(POPL):7:1–7:29. ISSN 2475-1421. doi:10.1145/3158095.

[5] Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. 2009. Typestate-oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pp. 1015–1022. ACM, New York, NY, USA. ISBN 978-1-60558-768-4. doi:10.1145/1639950.1640073.

[6] Atkey, R. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19(3-4):335–376. ISSN 1469-7653, 0956-7968. doi:10.1017/S095679680900728X.

[7] Bauer, A. and Pretnar, M. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10(4). doi:10.2168/LMCS-10(4:9)2014.

[8] Bauer, A. and Pretnar, M. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84(1):108–123. ISSN 2352-2208. doi:10.1016/j.jlamp.2014.02.001.

[9] Benton, N. and Kennedy, A. 2001. Exceptional syntax. *Journal of Functional Programming* 11(04). ISSN 0956-7968, 1469-7653. doi:10.1017/S0956796801004099.

[10] Benton, N., Kennedy, A., Hofmann, M., and Beringer, L. 2006. Reading, Writing and Relations. In N. Kobayashi, ed., *Programming Languages and Systems*, Lecture Notes in Computer Science, pp. 114–130. Springer Berlin Heidelberg. ISBN 978-3-540-48938-2.

[11] Berghofer, S. and Urban, C. 2007. A Head-to-Head Comparison of de Bruijn Indices and Names. *Electronic Notes in Theoretical Computer Science* 174(5):53–67. ISSN 1571-0661. doi:10.1016/j.entcs.2007.01.018.

[12] Biernacki, D., Piróg, M., Polesiuk, P., and Sieczkowski, F. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2(POPL):8:1–8:30. ISSN 2475-1421. doi:10.1145/3158096.

[13] Biernacki, D. and Polesiuk, P. 2018. Logical relations for coherence of effect subtyping. *Logical Methods in Computer Science* 14(1). ISSN 1860-5974.

[14] Brady, E. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23(5):552–593. ISSN 0956-7968, 1469-7653. doi:10.1017/S095679681300018X.

[15] Brady, E. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 133–144. ACM, New York, NY, USA. ISBN 978-1-4503-2326-0. doi:10.1145/2500365.2500581.

[16] Brady, E. 2015. Resource-Dependent Algebraic Effects. In J. Hage and J. McCarthy, eds., *Trends in Functional Programming*, Lecture Notes in Computer Science, pp. 18–33. Springer International Publishing. ISBN 978-3-319-14675-1.

[17] Breazu-Tannen, V., Coquand, T., Gunter, C.A., and Scedrov, A. 1991. Inheritance as implicit coercion. *Information and Computation* 93(1):172–221. ISSN 0890-5401. doi:10.1016/0890-5401(91)90055-7.

[18] Curien, P.L. and Ghelli, G. 1992. Coherence of subsumption, minimum typing and typechecking in F$\leq$. *Mathematical Structures in Computer Science* 2(1):55–91. ISSN 1469-8072, 0960-1295. doi:10.1017/S0960129500001134.

[19] Danvy, O. and Nielsen, L.R. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pp. 162–174. ACM, New York, NY, USA. ISBN 978-1-58113-388-2. doi:10.1145/773184.773202.

[20] De Bruijn, N.G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75(5):381–392. ISSN 1385-7258. doi:10.1016/1385-7258(72)90034-0.

[21] Dunfield, J. and Krishnaswami, N.R. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 429–442. ACM, New York, NY, USA. ISBN 978-1-4503-2326-0. doi:10.1145/2500365.2500582.

[22] Dybjer, P. 1991. Inductive Sets and Families in Martin-Löf's Type Theory. In G. Huet and G. Plotkin, eds., *Logical Frameworks*, pp. 280–306. Cambridge University Press, New York, NY, USA. ISBN 978-0-521-41300-8.

[23] Elgot, C.C. 1975. Monadic Computation and Iterative Algebraic Theories. In H.E. Rose and J.C. Shepherdson, eds., *Studies in Logic and the Foundations of Mathematics*, volume 80 of *Logic Colloquium '73 Proceedings of the Logic Colloquium*, pp. 175–230. Elsevier.

[24] Filinski, A. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pp. 446–457. ACM, New York, NY, USA. ISBN 0-89791-636-0. doi:10.1145/174675.178047.

[25] Fiore, M., Plotkin, G., and Turi, D. 1999. Abstract syntax and variable binding. In *Proceedings of the 14th Symposium on Logic in Computer Science*, pp. 193–202. doi:10.1109/LICS.1999.782615.

[26] Forster, Y., Kammar, O., Lindley, S., and Pretnar, M. 2017. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1(ICFP):13:1–13:29. ISSN 2475-1421. doi:10.1145/3110257.

[27] Foundation, F.S. 2019. Emacs Lisp manual for Emacs 26.2.

[28] Frisch, A. 2013. Static exceptions. Web page on https://www.lexifi.com/ocaml/static-exceptions/ .

[29] Führmann, C. 1999. Direct Models of the Computational Lambda-calculus. *Electronic Notes in Theoretical Computer Science* 20:245–292. ISSN 1571-0661. doi:10.1016/S1571-0661(04)80078-1.

[30] Geron, B. and Levy, P.B. 2016. Iteration and Labelled Iteration. In *MFPS 2016*, volume 325 of *Electronic Notes in Theoretical Computer Science*, pp. 127–146. Elsevier, Pittsburgh. doi:10.1016/j.entcs.2016.09.035.

[31] Goncharov, S., Rauch, C., and Schröder, L. 2015. Unguarded Recursion on Coinductive Resumptions. In *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pp. 183–198. Elsevier. doi:10.1016/j.entcs.2015.12.012.

[32] Goncharov, S., Rauch, C., and Schröder, L. 2018. A Metalanguage for Guarded Iteration. In B. Fischer and T. Uustalu, eds., *Theoretical Aspects of Computing – ICTAC 2018*, Lecture Notes in Computer Science, pp. 191–210. Springer International Publishing. ISBN 978-3-030-02508-3.

[33] Goncharov, S., Schröder, L., and Mossakowski, T. 2009. Kleene Monads: Handling Iteration in a Framework of Generic Effects. In A. Kurz, M. Lenisa, and A. Tarlecki, eds., *Algebra and Coalgebra in Computer Science*, number 5728 in Lecture Notes in Computer Science, pp. 18–33. Springer Berlin Heidelberg. ISBN 978-3-642-03740-5. doi:10.1007/978-3-642-03741-2_3.

[34] Gosling, J., Joy, B., Steele, G.L., Bracha, G., and Buckley, A. 2014. *The Java® language specification*. Addison-Wesley, Upper Saddle River, NJ. ISBN 978-0-13-390069-9.

[35] Hillerström, D. and Lindley, S. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pp. 15–27. ACM, New York, NY, USA. ISBN 978-1-4503-4435-7. doi:10.1145/2976022.2976033.

[36] Jones, M.P. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In J. Jeuring and E. Meijer, eds., *Advanced Functional Programming*, volume 925, pp. 97–136. Springer. doi:10.1007/3-540-59451-5_4.

[37] Kakutani, Y. 2002. Duality between Call-by-Name Recursion and Call-by-Value Iteration. In J. Bradfield, ed., *Computer Science Logic*, number 2471 in Lecture Notes in Computer Science, pp. 506–521. Springer Berlin Heidelberg. ISBN 978-3-540-44240-0. doi:10.1007/3-540-45793-3_34.

[38] Kamareddine, F. and Ríos, A. 1995. A λ-calculus à la de Bruijn with explicit substitutions. In M. Hermenegildo and S.D. Swierstra, eds., *Programming Languages: Implementations, Logics and Programs*, number 982 in Lecture Notes in Computer Science, pp. 45–62. Springer Berlin Heidelberg. ISBN 978-3-540-60359-7. doi: 10.1007/BFb0026813.

[39] Kammar, O. 2014. *Algebraic theory of type-and-effect systems*. PhD thesis, University of Edinburgh.

[40] Kammar, O., Lindley, S., and Oury, N. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 145–158. ACM, New York, NY, USA. ISBN 978-1-4503-2326-0. doi:10.1145/2500365.2500590.

[41] Kammar, O. and Plotkin, G.D. 2012. Algebraic Foundations for Effect-dependent Optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pp. 349–360. ACM, New York, NY, USA. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103698.

[42] Kennedy, A. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pp. 177–190. ACM, New York, NY, USA. ISBN 978-1-59593-815-2. doi:10.1145/1291151.1291179.

[43] Kozen, D. and Mamouras, K. 2013. Kleene Algebra with Products and Iteration Theories. In S.R.D. Rocca, ed., *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 415–431. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany. ISBN 978-3-939897-60-6. doi: 10.4230/LIPIcs.CSL.2013.415.

[44] Kozen, D. and Mamouras, K. 2014. Kleene Algebra with Equations. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, eds., *Automata, Languages, and Pro-*

gramming, number 8573 in Lecture Notes in Computer Science, pp. 280–292. Springer Berlin Heidelberg. ISBN 978-3-662-43950-0. doi:10.1007/978-3-662-43951-7_24.

[45] Laird, J. 2005. The Elimination of Nesting in SPCF. In P. Urzyczyn, ed., *Typed Lambda Calculi and Applications*, number 3461 in Lecture Notes in Computer Science, pp. 234–245. Springer Berlin Heidelberg. ISBN 978-3-540-25593-2. doi:10.1007/11417170_18.

[46] Leijen, D. 2016. Algebraic Effects for Functional Programming. Microsoft Technical Report MSR-TR-2016-29, Microsoft Research.

[47] Leijen, D. 2017. Type Directed Compilation of Row-typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 486–499. ACM, New York, NY, USA. ISBN 978-1-4503-4660-3. doi:10.1145/3009837.3009872.

[48] Levy, P.B. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19(4):377–414. ISSN 1388-3690, 1573-0557. doi:10.1007/s10990-006-0480-6.

[49] Longley, J. 2015. The recursion hierarchy for PCF is strict. Informatics Research Report EDI-INF-RR-1421, School of Informatics, University of Edinburgh.

[50] Lucassen, J.M. and Gifford, D.K. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 47–57. ACM New York, San Diego, California, USA. ISBN 0-89791-252-7. doi:10.1145/73560.73564.

[51] Marlow, S. 2010. Haskell 2010 language report.

[52] McBride, C. and McKinna, J. 2004. Functional Pearl: I Am Not a Number–I Am a Free Variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pp. 1–9. ACM, New York, NY, USA. ISBN 1-58113-850-4. doi:10.1145/1017472.1017477.

[53] Milius, S. and Litak, T. 2013. Guard Your Daggers and Traces: On The Equational Properties of Guarded (Co-)recursion. *Electronic Proceedings in Theoretical Computer Science* 126:72–86. ISSN 2075-2180. doi:10.4204/EPTCS.126.6.

[54] Ohori, A. 1989. A Simple Semantics for ML Polymorphism. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pp. 281–292. ACM, New York, NY, USA. ISBN 978-0-89791-328-7. doi:10.1145/99370.99393.

[55] Orchard, D. and Petricek, T. 2014. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pp. 13–24. ACM, New York, NY, USA. ISBN 978-1-4503-3041-1. doi:10.1145/2633357.2633368.

[56] Piróg, M., Schrijvers, T., Wu, N., and Jaskelioff, M. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pp. 809–818. ACM, New York, NY, USA. ISBN 978-1-4503-5583-4. doi:10.1145/3209108.3209166.

[57] Pitts, A.M. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* 10(03):321–359. ISSN 1469-8072.

[58] Plotkin, G. and Power, J. 2001. Adequacy for Algebraic Effects. In F. Honsell, M. Miculan, G. Goos, J. Hartmanis, and J. van Leeuwen, eds., *Foundations of Software Science and Computation Structures*, volume 2030, pp. 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-41864-1.

[59] Plotkin, G. and Power, J. 2001. Semantics for Algebraic Operations. *Electronic Notes in Theoretical Computer Science* 45:332–345. ISSN 1571-0661. doi:10.1016/S1571-0661(04)80970-8.

[60] Plotkin, G. and Power, J. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11(1):69–94. ISSN 0927-2852, 1572-9095. doi:10.1023/A:1023064908962.

[61] Plotkin, G. and Pretnar, M. 2008. A Logic for Algebraic Effects. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science*, pp. 118–129. IEEE. ISBN 978-0-7695-3183-0. doi:10.1109/LICS.2008.45.

[62] Plotkin, G.D. and Pretnar, M. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9(4). ISSN 18605974. doi:10.2168/LMCS-9(4:23)2013.

[63] Power, J. and Robinson, E. 1997. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science* 7(5):453–468. ISSN 1469-8072, 0960-1295.

[64] Pretnar, M. 2010. *The logic and handling of algebraic effects*. PhD thesis, University of Edinburgh.

[65] Reynolds, J.C. 2000. The Meaning of Types: From Intrinsic to Extrinsic Semantics. Technical Report RS-00-32, DAIMI, Department of Computer Science, University of Aarhus.

[66] Sperber, M., Dybvig, R.K., Flatt, M., Straaten, A.V., Findler, R., and Matthews, J. 2009. Revised Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19(S1):1–301. ISSN 1469-7653, 0956-7968. doi:10.1017/S0956796809990074.

[67] Stallman, R.M. 1981. EMACS: The Extensible, Customizable Self-Documenting Display Editor. AI Memo AIM-519a, Massachusetts Institute for Technology.

[68] Staton, S. 2010. Completeness for Algebraic Theories of Local State. In D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, and L. Ong, eds., *Foundations of Software Science and Computational Structures*, volume 6014, pp. 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-12031-2. doi:10.1007/978-3-642-12032-9_5.

[69] Steele Jr., G.L. and Sussman, G.J. 1978. The revised report on SCHEME: a dialect of LISP. Artificial Intelligence Project Memo AIM-452, Massachusetts Institute for Technology.

[70] Tait, W.W. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32(02):198–212. ISSN 1943-5886. doi:10.2307/2271658.

[71] Zhang, Y. and Myers, A.C. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3(POPL):5:1–5:29. ISSN 2475-1421. doi:10.1145/3290318.