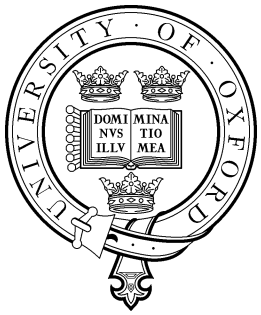
Program Equivalence for Algebraic Effects via Modalities



Cristina Matache
St Cross College
University of Oxford

A thesis submitted for the degree of
MSc in Computer Science
Trinity 2018

Abstract

This dissertation is concerned with the study of program equivalence and algebraic effects as they arise in the theory of programming languages. Algebraic effects represent impure behaviour in a functional programming language, such as input and output, exceptions, nondeterminism etc. all treated in a generic way. Program equivalence aims to identify which programs can be considered equal in some sense. This question has been studied for a long time but has only recently been extended to languages with algebraic effects, which are a newer development. Much work remains to be done in order to understand program equivalence in the presence of algebraic effects. In particular, there is no characterisation of contextual equivalence using a logic.

We define a logic whose formulas express properties of higher-order programs with algebraic effects. We then investigate three notions of program equivalence for algebraic effects: *logical equivalence* induced by the aforementioned logic, *applicative bisimilarity* and *contextual equivalence*. For the programming language used in this dissertation, we prove that they all coincide.

Therefore, the main novel contribution of the dissertation is defining the first logic for algebraic effects whose induced program equivalence coincides with contextual equivalence.

Acknowledgements

I would like to thank my supervisor, Sam Staton, for his guidance and patience without which this project would not have been possible. I am also grateful to Alex Simpson and Niels Voorneveld for insightful discussions about their work. Finally, I want to thank my parents for their support and for making this year in Oxford possible.

Contents

List of Figures

Chapter 1

Introduction

This dissertation is a theoretical study of program equivalence for a higher-order language with algebraic effects. In particular, we are interested in finding a logic of program properties that characterises contextual equivalence. This chapter reviews the history of program equivalence and explains why it is a hard but nevertheless interesting problem. At the same time, we outline the main research questions that led to the present work. Finally, we summarise the contributions of our work and the structure of the dissertation.

1.1 Motivation

Although undecidable in general, program equivalence is a fundamental problem both in the theory of programming languages and in formal verification. Programming language researchers are concerned with theoretical ways of reasoning about program equivalence based on formal semantics, like denotational or operational semantics. We adopt this point of view in the dissertation. Verification combines theory with building automated tools for checking program equivalence whenever possible.

Program equivalence seeks to establish when two programs are interchangeable, or when they behave the same, for some definition of behaviour. For the semantics of programming languages, this is useful because one can regard the meaning of a program as the equivalence class of programs that it belongs to. From a more practical perspective, checking program equivalence can determine whether a program implements a specification, which is itself given as a more familiar program. For example, establishing whether certain compiler optimisations are safe can be done in this way.

The first definition of program equivalence that springs to mind is that two programs are equivalent when they return the same result. In the case of a Turing-complete language, one would also like to take into account the possibility of divergence. Therefore, naively two programs are equivalent if they either return the same result or they both diverge.

This definition has two problems. Firstly, establishing divergence of programs is undecidable due to the halting problem, so program equivalence is in general hard to calculate. Working around this issue is mainly the focus of the formal verification community. Secondly, the above naive definition of program equivalence becomes unclear in the presence of higher-order functions. A higher-order function can receive as arguments and return as results other functions, rather than just ground data, such as natural numbers or booleans.

As an example, consider the following programs:

$$one = \lambda f.\lambda y.(f\ y)$$

$$two = \lambda f.\lambda y.f\ (f\ y)$$

$$two' = \lambda f.\lambda y.f\ (one\ f\ y)$$

where *one* and *two* represent the first two Church numerals. Morally, the functions *two* and two' should be equivalent in the λ-calculus, although they are not syntactically equal. Therefore the problem is: what does it mean for two higher-order functions to be the same?

Over the years, the programming languages community has proposed many answers to this question. At the beginning, they focussed on pure higher-order languages, with no side effects, such as the λ-calculus and PCF [Plo77]. PCF is a simply-typed extension of the λ-calculus with general recursion and a datatype of natural numbers and is therefore Turing-complete. These languages were chosen for their simplicity and because they have had well-understood formal semantics for a long time.

In this context, program equivalence based on denotational semantics was proposed: if two programs have the same denotation, they are equivalent. An alternative notion based on operational semantics is Morris-style contextual equivalence [Mor69]: two programs are equivalent if they have the same *observable* behaviour in all program contexts. In this dissertation, we are not concerned with denotational equality; contextual equivalence will be discussed more in Section 2.1. Notice that it improves on the naive definition of program equivalence because, to test the equality of functions, a context can provide them with arguments and observe their reduction behaviour.

To address some of the shortcomings of denotational equality and contextual equivalence, other notions of program equivalence that use operational semantics have been proposed. Examples include logical equivalence, that is, equivalence induced by satisfaction of formulas in a logic of program properties (e.g. [HM85]), applicative bisimilarity [Abr90] and logical relations [Tai67]. Logical equivalence is the main focus of this dissertation. Notably, it has strong connections with formal verification, where modal logics such as temporal logics [Pnu77] and the modal μ-calculus [Koz83] are used to specify and verify properties of programs. Applicative bisimilarity will be discussed in more detail in Section 2.1, but we will not be concerned with logical relations for program equivalence.

Given the wide variety of program equivalences mentioned so far, a natural question is comparing them: do they coincide, is one included in the other or are they altogether different? This question has been studied as new definitions arose. For example, Plotkin showed that for PCF denotational equality implies contextual equivalence.

Gradually, the questions identified so far have been posed for more complex languages than the λ-calculus or PCF. An interesting addition are impure operations such as input and output, exceptions, nondeterminism, state or continuations, whose semantics has been studied more recently. They are known as *computational effects*. In his influential work, Eugenio Moggi [Mog91] gave a general denotational semantics for computational effects using monads.

Subsequently, Plotkin and Power [PP01, PP02, PP03] initiated a program of research concerned with *algebraic effects*, a subset of computational effects whose behaviour can

be axiomatised by a set of equations. All the example of effects mentioned before are algebraic with the exception of continuations and exception handling. The purpose of algebraic effects is to give a unified, uniform treatment of the semantics of effects and of their combinations [HPP06]. The latter cannot easily be achieved using Moggi's semantics.

In this context, a new question becomes apparent: when are two higher-order programs *exhibiting algebraic effects* equivalent? The natural starting point is to extend the existing notions of program equivalence to languages with algebraic effects, and once again compare the resulting relations. A lot of work has been done in this direction for specific effects, such as nondeterminism (e.g. [Las98]) and probabilistic choice (e.g. [CL14]). Ideally however, a notion of program equivalence should be applicable to *any* algebraic effect.

Several such notions of program equivalence for generic algebraic effects have been developed. Johann, Simpson and Voigtländer [JSV10] study contextual equivalence and a corresponding logical relation. Dal Lago, Gavazzo and Levy [LGL17a] are concerned with applicative bisimilarity. Plotkin and Pretnar [PP08] propose a logic for algebraic effects that is sound with respect to other notions of program equivalence, but not complete in general. Simpson and Voorneveld [SV18] propose a modal logic whose induced program equivalence coincides with applicative bisimilarity, but not with contextual equivalence.

Thus, the main question of the dissertation arises: can we find a logic that *characterises* contextual equivalence for a higher-order language with generic algebraic effects?

1.2 Contributions

The programming language we consider in this dissertation is named ECPS. It is a call-by-value continuation-passing variant of PCF with generic algebraic effects. It is a higher-order Turing-complete language.

Being a continuation-passing language means that functions receive an additional argument which specifies how the computation should proceed once the function terminates. This argument is called a *continuation*. Compared to the direct style of programming, continuation-passing style makes control flow and the order of evaluation explicit. Thanks to these features, continuation-passing languages are often used as intermediate languages inside compilers. So ECPS could be seen as a simple variant of an intermediate language.

In the dissertation, we are concerned with three notions of program equivalence for ECPS: logical equivalence, contextual equivalence and applicative bisimilarity, and with the relationship between them. Most importantly, we wish to define a logic whose induced program equivalence coincides with contextual equivalence.

Any notion of program equivalence needs to satisfy two key properties: being an equivalence relation and being compatible. Compatibility means that equivalent programs can be substituted for a variable in a program equation, thus allowing compositional reasoning about program equivalence. These requirements are both explained in more detail in Section 2.1.

The novel contributions of the dissertation are the following:

1. We study the relationship between ECPS and the language used in the work of Simpson and Voorneveld [SV18]. We provide a correct embedding of their language into ECPS (Theorem 4.3.1).

2. We develop applicative bisimilarity for ECPS and prove it is a compatible equivalence relation (Lemma 5.3.1 and Theorem 5.3.7).

3. We define a logic whose formulas express properties of ECPS programs. We prove that program equivalence induced by the logic coincides with applicative bisimilarity (Theorem 6.2.3). Therefore, logical equivalence is compatible.

4. We present two equivalent definitions of contextual equivalence for ECPS, which are both equivalence relations and compatible. We prove that contextual equivalence coincides with applicative bisimilarity (Theorem 7.2.2). This leads to the main result of the dissertation: logical equivalence coincides with contextual equivalence (Corollary 7.2.3).

1.3 Structure of the Dissertation

Chapter 2 starts with an informal introduction to program equivalence and algebraic effects. It then reviews the work of Simpson and Voorneveld [SV18] since it is the most closely related to this dissertation. Finally, it exemplifies the distinction between contextual equivalence and logical equivalence in the context of their development.

Chapter 3 introduces the language ECPS and its operational semantics. It then reviews two proof techniques, namely coinduction and logical relations.

Chapters 4 to 7 contain the novel technical content. Chapter 4 is concerned with justifying the use of the ECPS language. It gives a translation from the programming language used by Simpson and Voorneveld into ECPS and proves the translation correct.

The following three chapters study program equivalence. Chapter 5 defines applicative bisimilarity and proves its main properties. Chapter 6 introduces a logic for ECPS and proves that logical equivalence coincides with applicative bisimilarity. Finally, Chapter 7 develops contextual equivalence and proves it coincides with logical equivalence.

The last chapter reviews the material in the dissertation and surveys previous work. It then sketches several directions for future work and closes with some personal remarks.

Chapters 2 through 7 all end with an accessible summary of their most important points. For an overview of the dissertation, one can consult the “Chapter Summaries”. Almost all the mathematical proofs completed as part of the project appear in the dissertation. To facilitate reading, routine or overlong proofs appear in the appendices rather than in the main body of the text. Therefore, Chapters 4 to 7 all have a corresponding appendix.

Chapter 2

Background and Explanation of Problem

This chapter starts with an informal discussion of program equivalence and algebraic effects. It then reviews in some detail the work of Simpson and Voorneveld [SV18] on program equivalence for algebraic effects. The syntax and operational semantics of a programming language with algebraic effects named EPCF is introduced. Two forms of program equivalence are discussed: logical equivalence and applicative bisimilarity, and they are compared to contextual equivalence.

2.1 Program Equivalence

As Pitts observes [Pit11], for a notion of program equivalence, or equality, to be useful it should be a congruence, that is, satisfy two key properties: being an *equivalence relation* and being *compatible*. The former allows reasoning through a chain of equations in order to establish that two programs P_1 and P_n are equal:

$$P_1 \simeq P_2 \simeq \ldots \simeq P_n.$$

Assuming that programs can take parameters, $P(x)$, compatibility means that we can substitute equivalent programs for a parameter in an equation:

$$P_1(x) \simeq P_2(x) \text{ and } Q_1 \simeq Q_2 \implies P_1(Q_1) \simeq P_2(Q_2).$$

This property is important because it allows us to reason compositionally about programs. In order to decide whether two programs are equivalent, it suffices to investigate whether its subphrases are equivalent. When the parameter is allowed to be a function or a process, rather than just ground data, like booleans or integers, compatibility becomes even more important but also harder to establish.

One of the most intuitive notions of program equivalence is Morris-style *contextual equivalence* [Mor69]. Two programs are contextually equivalent if and only if they have the same observable behaviour in all program contexts:

$$P_1 \simeq P_2 \quad \iff \quad \forall C.\ \mathfrak{O}(C[P_1]) = \mathfrak{O}(C[P_2]).$$

In the case of the untyped λ-calculus, a possible definition for the "observable behaviour" of program P, $\mathfrak{O}(P)$, is whether or not P terminates, written $P{\Downarrow}$. Therefore $\mathfrak{O}(C[P_1]) = \mathfrak{O}(C[P_2])$ becomes:

$$C[P_1]{\Downarrow} \iff C[P_2]{\Downarrow}.$$

However, contextual equivalence is difficult to establish for particular programs because of the quantification over all contexts. Therefore, Abramsky [Abr90] proposed another notion of equivalence for the untyped λ-calculus named *applicative bisimilarity*.

Bisimilarity was first defined by Milner [Mil80] for the process calculus CCS which models concurrency. The main idea is that two processes are bisimilar if whenever one of them can advance by one step, the other can perform a matching step, and the two resulting processes are again bisimilar. The circularity of this definition of bisimilarity suggests that it can be defined coinductively as the greatest relation with a certain property.

In the case of the λ-calculus, the steps that programs can take are β-reduction steps. Therefore, applicative similarity, the one-sided version of bisimilarity, is defined as:

The greatest relation $\precsim$, such that $P_1 \precsim P_2$ implies

$$P_1 \longrightarrow^* \lambda x.P_1' \implies$$

$$\exists P_2' \text{ such that } P_2 \longrightarrow^* \lambda x.P_2' \text{ and for any value } v,\ P_1'[v/x] \precsim P_2'[v/x].$$

Applicative bisimilarity is defined analogously.

For the λ-calculus, applicative bisimilarity coincides with contextual equivalence e.g. [Abr90, Pit11]. Therefore, checking whether two λ-terms are bisimilar is a sound and complete proof technique for establishing contextual equivalence, easier to use in practice.

Another approach to program equivalence is to define a logic $\mathcal{L}$ whose formulas ϕ represent program properties. In this setting, two programs are equivalent if and only if they satisfy the same formulas in this logic:

$$P_1 \simeq P_2 \quad \iff \quad (\forall \phi \text{ in } \mathcal{L}.\ P_1 \models \phi \iff P_2 \models \phi).$$

An example of a logic that describes program properties is Hennessy-Milner logic [HM85], which concerns CCS processes. In fact, bisimilarity for CCS coincides with the equivalence induced by Hennessy-Milner logic.

In this dissertation, we are most interested in logical equivalence, so we will consider some informal examples of formulas.

Example 2.1.1. Consider a call-by-value simply-typed λ-calculus with natural numbers. A logical formula could be:

$$\phi = \{3\} \mapsto \{2\}.$$

A function f satisfies ϕ if, given argument 3, the expression $(f\ x)$ reduces to 2. Consider for example the following function:

$$f = \lambda n.\ \textbf{if } n = 3 \textbf{ then pred } n \textbf{ else succ } n.$$

We can see that f indeed satisfies ϕ, but f does not satisfy $\{4\} \mapsto \{3\}$.

2.2 Algebraic Effects and Logical Properties

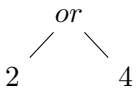
In general, programming languages that are purely functional do not provide "impure" operations such as input and output, nondeterministic or probabilistic choice, global state etc. These features are known as algebraic effects. To include them in a programming language, it suffices to add relevant operations to the language.

As an example consider nondeterministic choice. This can be implemented by adding an operation $or(-,-)$, where, in the term $or(t_1, t_2)$, an external agent chooses nondeterministically whether to execute term t_1 or t_2.

Example 2.2.1. Consider the simply-typed λ-calculus from the previous example extended with or. Now there is more than one value that a term may reduce to. For example:

$$(g\ 3) \quad \text{where } g = \lambda n.\ or(\mathbf{pred}\ n,\ \mathbf{succ}\ n)$$

may reduce to either 2 or 4. The reduction behaviour of $(g\ 3)$ could be represented as a tree:



Recall the formula $\{3\} \mapsto \{2\}$ from the previous section. We can interpret it either as: "the function *always* returns 2" or "*may* return 2". As a result we have two new formulas: $\phi_1 = \{3\} \mapsto \Box\{2\}$ and $\phi_2 = \{3\} \mapsto \Diamond\{2\}$. We can see that g satisfies the latter but not the former.

Example 2.2.2. As another example, consider a higher-order function:

$$h = \lambda f.\ \lambda n.\ or(f\ (\mathbf{pred}\ n),\ f\ (\mathbf{succ}\ n)) : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}).$$

If we apply it to arguments g and 2, and then g and 4, the trees of $(h\ g\ 2)$ and $(h\ g\ 4)$ respectively are:

Now consider the formula:

$$\psi = (\{3\} \mapsto \Diamond\{2\}) \mapsto (\{2,4\} \mapsto \Diamond\{2\}).$$

It says that, given a function f that satisfies $\{3\} \mapsto \Diamond\{2\}$, h returns another function which when given either 2 or 4 as argument *may* return 2. Function h satisfies ψ because in either case it may call $(f\ 3)$, which we know may return 2.

2.3 Program Equivalence for Algebraic Effects

The notions of program equivalence presented in Section 2.1 have been extended recently to programming languages with algebraic effects. Johann, Simpson and Voigtländer [JSV10] study contextual equivalence for a polymorphic language with recursion and generic effects. They characterise contextual equivalence using a logical relation and thus prove some of its fundamental properties.

Dal Lago, Gavazzo and Levy [LGL17a] give an abstract account of applicative bisimilarity for an untyped λ-calculus with generic algebraic effects. They show that applicative bisimilarity is included in contextual equivalence, but they note that this inclusion is strict in general.

Simpson and Voorneveld [SV18] consider a simply-typed programming language with recursion and generic algebraic effects. They propose a modal logic in which formulas expressing program behaviour are very similar in spirit to the example formulas we have seen so far. They also define applicative bisimilarity following [LGL17a]. The logical equivalence induced by the modal logic is then proved to coincide with applicative bisimilarity.

In the conclusion of their paper, Simpson and Voorneveld observe that logical equivalence is included in contextual equivalence. However, contextual equivalence equates more programs than logical equivalence does. Therefore, an open research direction is finding a logic that characterises contextual equivalence. This is the main goal of this dissertation.

2.4 PCF with Effects – EPCF

The programming language used in the work of Simpson and Voorneveld [SV18] is a call-by-value, simply-typed λ-calculus with recursion, a datatype of natural numbers and algebraic effects. Therefore, the language is a variant of Plotkin's PCF [Plo77] extended with algebraic effects; in this work we will refer to it as EPCF.

In order to simplify their proofs, Simpson and Voorneveld formulate EPCF as fine-grained call-by-value [LPT03]. This means that there is a distinction between terms that are values and terms that are computations; they form separate syntactic categories. For example, $\lambda x{:}\mathbb{N}.S(x)$ and 3 are values because they cannot reduce, while $(\lambda x{:}\mathbb{N}.S(x))\ 3$ is a computation. Here $S(x)$ represents the successor of x. The fine-grained call-by-value formulation is equivalent to the usual call-by-value formulation.

Definition 2.4.1 (EPCF)**.** *Types and environments:*

$$\tau, \rho \coloneqq \mathbb{1} \mid \mathbb{N} \mid \rho \to \tau$$

$$\Gamma \coloneqq \emptyset \mid \Gamma, x : \tau.$$

Values and computations are defined by the grammar:

$$V, W \coloneqq \star \mid Z \mid S(V) \mid \lambda x{:}\tau.M \mid x$$

$$M, N \coloneqq V\ W \mid \mathbf{return}\ V \mid \mathbf{let}\ M \Rightarrow x\ \mathbf{in}\ N \mid \mathbf{fix}\ V \mid \mathbf{case}\ V\ \mathbf{in}\ \{Z \Rightarrow M,\ S(x) \Rightarrow N\}.$$

The ground types are unit $\mathbb{1}$ and natural numbers $\mathbb{N}$. There is a countably infinite set of variables ranged over by x. The environment $\Gamma, x : \tau$ assumes that x does not appear in Γ.

Terms V, W represent values, and M, N represent computations, that is, terms which can be evaluated. The intuitive semantics of computations is the following: **return** V immediately returns the value V. The construct **let** $M \Rightarrow x$ **in** N is a sequencing operation: first it evaluates M, if this returns a value V, V is substituted for x in N, then $N[V/x]$ is evaluated. The computation **fix** V calculates the fixed point of the function V. The **case** V construct branches according to whether the natural number V is zero or a successor.

The language EPCF incorporates effects in a general way. Instead of specifying all the effect operations in the language, the definition of EPCF is parametrised by a set of effect operations Σ. The set Σ can be instantiated in turn for nondeterminism, probabilistic choice, global store, input and output etc. This is done in a series of examples at the end of the section.

Each operation $\sigma \in \Sigma$ has an arity which specifies what arguments the operation takes and what type the resulting computation has. The possible arities are:

$$\alpha^n \to \alpha \qquad \mathbb{N} \times \alpha^n \to \alpha \qquad \alpha^{\mathbb{N}} \to \alpha \qquad \mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha$$

where α can be regarded as a type variable. They should be interpreted as follows: $\sigma : \mathbb{N} \times \alpha^n \to \alpha$ is an operation that takes a natural number V and n computations of type α, $M_0, M_1, \ldots, M_{n-1}$. The resulting computation $\sigma(V; M_0, M_1, \ldots, M_{n-1})$ has type α. The expression $\alpha^{\mathbb{N}}$ represents a function from natural numbers to the type α.

Definition 2.4.1 (EPCF – continued)**.** *Fix a set of effect operations Σ, with associated arities. The grammar of computations is extended as follows:*

$$M, N ::= \ldots \mid \sigma(M_0, M_1, \ldots, M_{n-1}) \mid \sigma(V; M_0, M_1, \ldots, M_{n-1}) \mid \sigma(V) \mid \sigma(V; W).$$

The typing relations $\Gamma \vdash V : \tau$ and $\Gamma \vdash M : \tau$ are the least relations closed under the rules in Figure 2.1. The judgement $\Gamma \vdash V : \tau$ should be read as V has type τ in environment Γ.

Substitution of values for free variables inside values, $W[V/x]$, and inside computations, $M[V/x]$, is defined by recursion on the structure of W and M in a standard way. An example for effect operations is:

$$\sigma(W; M_0, M_1, \ldots, M_{n-1})[V/x] = \sigma(W[V/x]; M_0[V/x], M_1[V/x], \ldots, M_{n-1}[V/x]).$$

We use the notation $\mathit{Val}(\tau)$ for the set of closed values of type τ and $\mathit{Comp}(\tau)$ for the set of closed computations of type τ. We denote natural numbers $S^n(Z)$ by $\overline{n}$. Everywhere in the dissertation we consider terms up to α-conversion.

Below are examples of effect operations from [SV18] which will be used throughout the dissertation:

Example 2.4.2 (Pure functional computation)**.** In this case, the language has no effects so the set Σ is empty.

Example 2.4.3 (Nondeterminism)**.** There is one effect operation representing binary choice $or : \alpha^2 \to \alpha$ so $\Sigma = \{or\}$. It takes as arguments two computations of type α and chooses to run one of them. The choice is determined by an external agent.

Example 2.4.4 (Probabilistic choice)**.** Define $\Sigma = \{p\text{-}or\}$ where $p\text{-}or : \alpha^2 \to \alpha$ is a binary choice operator. With probability 0.5 it executes the first computation, otherwise the second computation.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}(\textsc{var}) \quad \frac{}{\Gamma \vdash \star : \mathbb{1}}(\textsc{unit}) \quad \frac{}{\Gamma \vdash Z : \mathbb{N}}(\textsc{zero}) \quad \frac{\Gamma \vdash V : \mathbb{N}}{\Gamma \vdash S(V) : \mathbb{N}}(\textsc{succ})$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathbf{return}\ V : \tau}(\textsc{ret}) \quad \frac{\Gamma, x : \tau \vdash M : \rho}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \rho}(\textsc{lbd}) \quad \frac{\Gamma \vdash V : \tau \to \rho \qquad \Gamma \vdash W : \tau}{\Gamma \vdash V\ W : \rho}(\textsc{app})$$

$$\frac{\Gamma \vdash V : (\tau \to \rho) \to (\tau \to \rho)}{\Gamma \vdash \mathbf{fix}\ V : \tau \to \rho}(\textsc{fix}) \quad \frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \tau \vdash N : \rho}{\Gamma \vdash \mathbf{let}\ M \Rightarrow x\ \mathbf{in}\ N : \rho}(\textsc{let})$$

$$\frac{\Gamma \vdash V : \mathbb{N} \qquad \Gamma \vdash M : \tau \qquad \Gamma, x : \mathbb{N} \vdash N : \tau}{\mathbf{case}\ V\ \mathbf{in}\ \{Z \Rightarrow M,\ S(x) \Rightarrow N\} : \tau}(\textsc{case})$$

$$\frac{\sigma : \alpha^n \to \alpha \qquad \Gamma \vdash M_i : \tau}{\Gamma \vdash \sigma(M_0, M_1, \ldots, M_{n-1}) : \tau}(\textsc{op}1) \quad \frac{\sigma : \alpha^{\mathbb{N}} \to \alpha \qquad \Gamma \vdash V : \mathbb{N} \to \tau}{\Gamma \vdash \sigma(V) : \tau}(\textsc{op}2)$$

$$\frac{\sigma : \mathbb{N} \times \alpha^n \to \alpha \qquad \Gamma \vdash V : \mathbb{N} \qquad \Gamma \vdash M_i : \tau}{\Gamma \vdash \sigma(V; M_0, M_1, \ldots, M_{n-1}) : \tau}(\textsc{op}3)$$

$$\frac{\sigma : \mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha \qquad \Gamma \vdash V : \mathbb{N} \qquad \Gamma \vdash W : \mathbb{N} \to \tau}{\Gamma \vdash \sigma(V; W) : \tau}(\textsc{op}4)$$

Figure 2.1: Typing judgements for EPCF [SV18].

Example 2.4.5 (Global store)**.** Fix a finite set of locations $\mathbb{L}$ for storing natural numbers. For each $l \in \mathbb{L}$, Σ contains the following operations:

$$lookup_l : \alpha^{\mathbb{N}} \to \alpha$$

$$update_l : \mathbb{N} \times \alpha \to \alpha.$$

The intuition is the following: the computation $lookup_l(V)$ looks up the number at location l then passes it to the function V; $update_l(\overline{n}, M)$ writes n to location l then runs the computation M.

Example 2.4.6 (Input/output)**.** In this case, $\Sigma = \{read, write\}$ where

$$read : \alpha^{\mathbb{N}} \to \alpha$$

$$write : \mathbb{N} \times \alpha \to \alpha.$$

The computation $read(V)$ reads a natural number from the input channel and passes it to the function V, which then executes. The computation $write(\overline{n}, M)$ outputs the number n then continues as M.

These operations seem very similar to those for global store. One difference is that updating location l with value n then immediately looking up the value will always yield $\overline{n}$. This is not the case for I/O operations *write* and *read*. There are no guarantees about the values on the output and input channels.

2.5 Operational Semantics of EPCF

Simpson and Voorneveld define an operational semantics for EPCF where computations evaluate to trees, following Plotkin and Power [PP01].

Definition 2.5.1. *The operational semantics uses evaluation stacks S to implement sequencing. They are defined as:*

$$S ::= id \mid S \circ (\textbf{let } (-) \Rightarrow x \textbf{ in } M).$$

Define the operation of 'filling in the hole' of a stack with a closed computation as:

$$id\{N\} = N$$

$$(S \circ (\textbf{let } (-) \Rightarrow x \textbf{ in } M))\{N\} = S\{\textbf{let } N \Rightarrow x \textbf{ in } M\}.$$

Write $Stack(\tau, \rho)$ for the set of stacks S which when given a computation $N \in Comp(\tau)$, return a computation $S\{N\} \in Comp(\rho)$.

The operational semantics consists of two relations, one between closed computations, and one between configurations (S, M), where $M \in Comp(\tau)$ and $S \in Stack(\tau, \rho)$.

$$(\lambda x{:}\tau.M)\ V \rightsquigarrow M[V/x]$$

$$\textbf{fix } F \rightsquigarrow \textbf{return } \lambda x{:}\tau.\textbf{let } F\ (\lambda y{:}\tau.\textbf{let fix } F \Rightarrow z \textbf{ in } z\ y) \Rightarrow w \textbf{ in } w\ x$$

$$\textbf{case } Z \textbf{ in } \{Z \Rightarrow M,\ S(x) \Rightarrow N\} \rightsquigarrow M$$

$$\textbf{case } S(V) \textbf{ in } \{Z \Rightarrow M,\ S(x) \Rightarrow N\} \rightsquigarrow N[V/x]$$

$$(S,\ \textbf{let}\ N \Rightarrow x\ \textbf{in}\ M) \rightarrowtail (S \circ \textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M,\ N)$$

$$(S \circ \textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M,\ \textbf{return}\ V) \rightarrowtail (S,\ M[V/x])$$

$$(S, M) \rightarrowtail (S, M')\ \textit{if}\ M \rightsquigarrow M'$$

Denote by $\rightarrowtail^*$ *the reflexive-transitive closure of* $\rightarrowtail$.

The reduction rule for the fixed point **fix** F is somewhat complicated by the syntactic restrictions imposed by fine-grained call-by-value. Intuitively, we can think of **fix** F as reducing to the thunk of $F(\textbf{fix}\ F)$.

By inspecting the reduction relation $\rightarrowtail$ we can see that it is deterministic. There are two ways $\rightarrowtail$ can get stuck. If $(S, M) \rightarrowtail^* (id, \textbf{return}\ V)$; in this case there is nothing left to do so the computation should terminate. Or if $(S, M) \rightarrowtail^* (S', \sigma(\ldots))$. In this case, an effect operation should take place and the execution should continue from S' with the computation chosen by the effect operation. It is also possible that $\rightarrowtail$ never terminates due to the presence of recursion.

This suggests that a computation of type τ should evaluate to an *effect tree* with leaves values of type τ. Denote the set of all such trees by $\mathit{Trees}(\tau)$. A possibly infinite tree in $\mathit{Trees}(\tau)$ can have:

- a leaf labelled by $\bot$, which signifies nontermination of $\rightarrowtail$;
- a leaf labelled by a value $V \in \mathit{Val}(\tau)$;
- a node labelled σ where $(\sigma : \alpha^n \to \alpha) \in \Sigma$; this has children $t_0, t_1, \ldots t_{n-1}$;
- a node labelled σ where $(\sigma : \alpha^{\mathbb{N}} \to \alpha) \in \Sigma$; this has infinitely many children $t_0, t_1, \ldots$, one for each natural number;
- a node labelled σ_m where $(\sigma : \mathbb{N} \times \alpha^n \to \alpha) \in \Sigma$; the label m is the natural number that σ takes as an argument; the node has children $t_0, t_1, \ldots t_{n-1}$;
- a node labelled σ_m where $(\sigma : \mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha) \in \Sigma$; this has children $t_0, t_1, \ldots$.

Before defining the tree associated to each computation, we need to define a partial order on $\mathit{Trees}(\tau)$ as follows:

$$tr_1 \leq tr_2 \quad \Longleftrightarrow \quad tr_1 \text{ can be obtained from } tr_2 \text{ by replacing some of its subtrees by } \bot.$$

This ordering endows $\mathit{Trees}(\tau)$ with an ω-CPO structure [Fio17] with least element $\bot$. This means that every increasing chain $tr_1 \leq tr_2 \leq \ldots$ has a least upper bound $\bigsqcup_{n \in \mathbb{N}} tr_n$.

Definition 2.5.1 (Continued)**.** *Define a family of functions*

$$|-,-|_{(-)} : \mathit{Stack}(\tau, \rho) \times \mathit{Comp}(\tau) \times \mathbb{N} \longrightarrow \mathit{Trees}(\rho).$$

The tree $|S, M|_n$ *represents the unfolding of computation* M *for* n *steps starting in stack*

S. The formal definition is:

$$|S, M|_0 = \bot$$

$$|S, M|_{n+1} = \begin{cases} V & \text{if } S = id \text{ and } M = \mathbf{return}\ V \\ |S', M'|_n & \text{if } (S, M) \rightarrowtail (S', M') \\ \sigma(|S, M_0|_n, \ldots, |S, M_{k-1}|_n) & \text{if } \sigma : \alpha^k \to \alpha \\ & \text{and } M = \sigma(M_0, M_1, \ldots, M_{k-1}) \\ \sigma(|S, V\ \overline{0}|_n, |S, V\ \overline{1}|_n, \ldots) & \text{if } \sigma : \alpha^{\mathbb{N}} \to \alpha \text{ and } M = \sigma(V) \\ \sigma_m(|S, M_0|_n, \ldots, |S, M_{k-1}|_n) & \text{if } \sigma : \mathbb{N} \times \alpha^k \to \alpha \\ & \text{and } M = \sigma(\overline{m}; M_0, M_1, \ldots, M_{k-1}) \\ \sigma_m(|S, V\ \overline{0}|_n, |S, V\ \overline{1}|_n, \ldots) & \text{if } \sigma : \mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha \text{ and } M = \sigma(\overline{m}; V) \\ \bot & \text{otherwise.} \end{cases}$$

From this definition we can see that $|S, M|_n \leq |S, M|_{n+1}$. Therefore, the effect tree associated with a computation is defined as:

$$|-| : Comp(\tau) \longrightarrow Trees(\tau)$$

$$|M| = \bigsqcup_{n \in \mathbb{N}} |id, M|_n.$$

We call $|M|$ a computation tree.

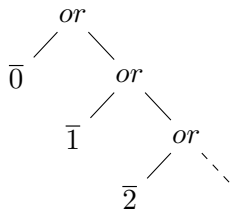
The intuitive interpretation of computation trees is that a path through the tree represents a potential execution path of the program. However, the operational semantics is not aware of this interpretation. In this sense, the effect operations are purely formal as in [PP01]. This will be illustrated in Example 2.5.5 below.

Example 2.5.2 (Pure functional computation)**.** In this case, there are no effect operations so all computation trees are leaves.

Example 2.5.3 (Nondeterminism)**.** Consider a computation that generates a natural number nondeterministically (from [SV18]):

$$?nat = \mathbf{let\ fix}\ (\lambda f{:}\mathbb{1}{\to}\mathbb{N}.or(\lambda y{:}\mathbb{1}.Z,\ \lambda y{:}\mathbb{1}.\mathbf{let}\ f\ y \Rightarrow u\ \mathbf{in}\ S(u))) \Rightarrow w\ \mathbf{in}\ w\ \star.$$

Its computation tree is:

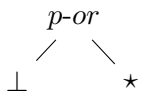


Thus, the notion of a tree whose nodes are nondeterministic choice points, discussed informally in Section 2.2, is formalised by a computation tree.

Example 2.5.4 (Probabilistic choice)**.** Define the following computation whose execution never terminates:

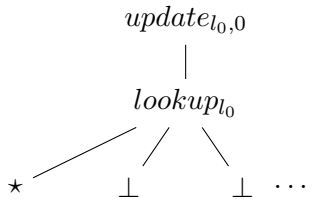
$$loop = \textbf{let } (\textbf{fix } \lambda f{:}\mathbb{N}{\to}\mathbb{1}.\textbf{return } f) \Rightarrow g \textbf{ in } g\ Z.$$

The computation tree of $p\text{-}or(loop, \textbf{return } \star) : \mathbb{1}$ is therefore:



Example 2.5.5 (Global store)**.** Consider a location $l_0 \in \mathbb{L}$. The following computation writes $\overline{0}$ to location l_0 then immediately reads this value:

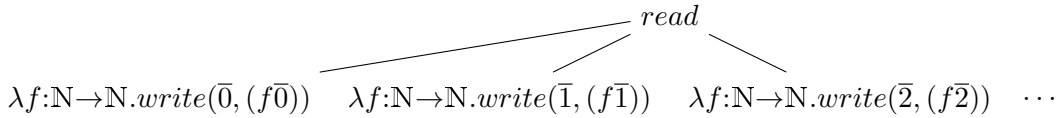
$$|update_{l_0}(\overline{0};\ lookup_{l_0}(\lambda x{:}\mathbb{N}.\textbf{case } x \textbf{ in } \{Z \Rightarrow \textbf{return } \star,\ S(y) \Rightarrow loop\}))| =$$



According to the intuitive interpretation that we give to the *lookup* and *update* operations only the path that returns a $\star$ can occur. However, the tree contains a path for each natural number that could be in l_0 because the $|-|$ function treats operation symbols as syntax without any interpretation. If we replaced *lookup* and *update* with the *read* and *write* operations from I/O then all paths would be relevant.

Example 2.5.6 (Input/output)**.** The following computation reads a natural number from the input channel, then returns a function whose behaviour depends on this number:

$$read(\lambda x{:}\mathbb{N}.\textbf{return } \lambda f{:}\mathbb{N}{\to}\mathbb{N}.write(x;\ (f\ x))) \quad : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$$



A path through the tree is not only a possible execution path of the computation, it also corresponds to an I/O trace.

2.6 Summary of Results about EPCF

The main contribution of Simpson's and Voorneveld's work [SV18] is that they define a modal logic whose formulas represent properties of EPCF programs. We will refer to it as EPCF logic. Moreover, they give a general analysis of the modalities involved.

In EPCF logic, there are two kinds of formulas all of which are attached an EPCF type. A formula $\phi : \tau$ describes a value of type τ, while $\Phi : \tau$ describes a computation of type τ. The definition of the logic starts from a set of basic formulas at each type which is then closed under negation, arbitrary conjunctions and disjunctions.

The basic formulas for values of type $\mathbb{N}$ are:

$$\{n\} \text{ where } n \in \mathbb{N}.$$

A closed value $W : \mathbb{N}$ satisfies $\{n\}$, written $W \models \{n\}$, if and only if $W = \overline{n}$.

For values of function types $\tau \to \rho$, the basic formulas are:

$$\phi \mapsto \Phi$$

where ϕ is a value formula of type τ and Φ is a computation formula of type ρ. The satisfaction $W \models \phi \mapsto \Phi$ holds if and only if:

$$\forall V \models \phi.\ (W\ V) \models \Phi.$$

A value formula $\phi \mapsto \Phi$ tests the behaviour of a function when it is being applied. This behaviour is the fundamental property of a function, hence the choice of value formula is reasonable.

Finally, computation formulas make use of a set of *modalities* $\mathcal{O}$. The set $\mathcal{O}$ contains sets of effect trees of type $\mathbb{1}$, that is, $\mathcal{O} \subseteq \mathcal{P}(\mathit{Trees}(\mathbb{1}))$. A basic computation formula of type τ is:

$$o\phi$$

where $o \in \mathcal{O}$ and ϕ is a value formula of type τ. We can see that o lifts a formula for values, ϕ, to a formula for computations. This is why o is named a *modality*.

For a tree $tr \in \mathit{Trees}(\tau)$ and a value formula $\phi : \tau$ denote by:

$$tr[\models \phi]$$

the tree in $\mathit{Trees}(\mathbb{1})$ obtained by replacing the leaves V of tr by $\star$ if $V \models \phi$ and by $\bot$ otherwise. We can now define satisfaction of computation formulas as:

$$M \models o\phi \quad \iff \quad |M|[\models \phi] \in o.$$

A computation formula $o\phi$ tests whether the possible return values of a computation satisfy ϕ and also tests the shape of the effect tree of the computation. These two pieces of information form the *observable behaviour* of a computation, in the sense of Section 2.1.

The definition of $\mathcal{O}$ depends on the effects present in the language so we will look at the nondeterminism example. The other effects are treated in more detail in Chapter 5, in the context of ECPS.

Example 2.6.1 (Nondeterminism)**.** Define $\mathcal{O} = \{\Diamond, \Box\}$ where:

$$\Diamond = \{tr \in \mathit{Trees}(\mathbb{1}) \mid tr \text{ has some } \star \text{ leaf }\}$$

$$\Box = \{tr \in \mathit{Trees}(\mathbb{1}) \mid tr \text{ has finite height and every leaf is a } \star\}.$$

The formula $\Diamond\phi$ says that a computation may return a value satisfying ϕ, whereas $\Box\phi$ asserts that it must return such a value. We can see that $\Box$ and $\Diamond$ which we discussed informally in Examples 2.2.1 and 2.2.2 are now defined as modalities.

In fact, the formulas used in those examples are valid formulas in EPCF logic:

$$\phi_1 = \{3\} \mapsto \Box\{2\}$$

$$\phi_2 = \{3\} \mapsto \Diamond\{2\}$$

$$\psi = (\{3\} \mapsto \Diamond\{2\}) \mapsto ((\{2\} \vee \{4\}) \mapsto \Diamond\{2\})$$

and the example functions:

$$g = \lambda n{:}\mathbb{N}.or(\textbf{pred}\ n,\ \textbf{succ}\ n)$$

$$h = \lambda f{:}\mathbb{N}{\to}\mathbb{N}.\ \lambda n{:}\mathbb{N}.\ or(f\ (\textbf{pred}\ n),\ f\ (\textbf{succ}\ n))$$

are valid EPCF terms for suitable encodings of **succ** and **pred**. We can see that $(id, (g\ \overline{3}))$ may return either $\overline{2}$ or $\overline{4}$ so using the definition of logical satisfaction we indeed obtain:

$$g \not\models \phi_1 \quad \text{and} \quad g \models \phi_2.$$

Using the same definition we can also see that $h \models \psi$.

Simpson and Voorneveld [SV18] are concerned with two notions of program equivalence: applicative bisimilarity and logical equivalence induced by satisfaction in EPCF logic. Applicative bisimilarity is a family of relations defined between well-typed EPCF terms. The definition is technically involved so we omit it. However, some key features of applicative bisimilarity are:

- Two natural numbers are bisimilar if and only if they are equal.
- Two function values are bisimilar if and only if for all arguments they yield bisimilar computations. This condition is the same as in the informal explanation of similarity from Section 2.1.
- To specify when two computations are bisimilar, their effect trees are inspected. Roughly speaking, M and N are bisimilar if: the possible return values of N approximate those of M well enough to preserve the properties of the effect tree of M, and vice-versa. The set of modalities $\mathcal{O}$ is used express properties of computation trees and to lift bisimilarity of values to a relation between trees.

Because the definition of bisimilarity depends on the set of modalities $\mathcal{O}$, it is named applicative $\mathcal{O}$-bisimilarity.

Definition 2.6.2. *Logical equivalence between well-typed EPCF terms is defined as:*

$$V \equiv_{EPCF} W \quad \Longleftrightarrow \quad \forall \phi : \tau.\ V \models \phi \Longleftrightarrow W \models \phi$$

$$M \equiv_{EPCF} N \quad \Longleftrightarrow \quad \forall \Phi : \tau.\ M \models \Phi \Longleftrightarrow N \models \Phi.$$

It is easy to prove that logical equivalence and applicative bisimilarity for EPCF are equivalence relations. In order to prove they are *compatible*, Simpson and Voorneveld identify two sufficient conditions that the set of modalities $\mathcal{O}$ should satisfy in general:

- Each modality $o \in \mathcal{O}$ needs to be Scott-open. This condition will be defined later (Definition 5.3.5). It refers to the Scott-topology on $\mathit{Trees}(\mathbb{1})$.
- The set $\mathcal{O}$ needs to be decomposable. This roughly means that, for a valid computation tree, its subtrees can also be described using the modalities in $\mathcal{O}$. Decomposability is a notion introduced by Simpson and Voorneveld whose definition is very technical so we omit it.

They show that for all the example effects in this dissertation $\mathcal{O}$ is decomposable and contains only Scott-open modalities. Assuming that $\mathcal{O}$ is a decomposable set of Scott-open modalities, Simpson and Voorneveld prove the following theorems:

Theorem 2.6.3. *Applicative $\mathcal{O}$-bisimilarity is compatible.*

Theorem 2.6.4. *Logical equivalence induced by EPCF logic coincides with applicative $\mathcal{O}$-bisimilarity. Hence, logical equivalence is compatible.*

Thus, they obtain a logical characterisation of applicative bisimilarity for EPCF.

2.7 Problem: Contextual vs. Logical Equivalence

Simpson and Voorneveld [SV18] briefly note that, in EPCF, contextual equivalence equates more programs than logical equivalance and bisimilarity do. In this section we look at example programs, originally due to Lassen [Las98] [1], that are contextually equivalent but not logically equivalent.

Consider a computation $min(v, w)$ which returns the minimum of the natural numbers v and w. This can be encoded in EPCF as:

$$
\begin{array}{l}
min(v,w) = \\
\quad \textbf{let } (\textbf{fix } \lambda f : \mathbb{N}{\to}\mathbb{N}{\to}\mathbb{N}. \\
\qquad\qquad \textbf{return } \lambda x : \mathbb{N}. \\
\qquad\qquad\quad \textbf{return } \lambda y : \mathbb{N}. \\
\qquad\qquad\qquad \textbf{case } x \textbf{ in } \{Z \Rightarrow \textbf{return } v, \\
\qquad\qquad\qquad\qquad\qquad S(z_0) \Rightarrow \textbf{case } y \textbf{ in } \{Z \Rightarrow \textbf{return } w, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S(z_1) \Rightarrow (\textbf{let } f\ z_0 \Rightarrow h \textbf{ in } h\ z_1)\}\} \\
\qquad) \Rightarrow g \textbf{ in let } g\ v \Rightarrow g' \textbf{ in } g'\ w\,.
\end{array}
$$

Example 2.7.1 (Nondeterminism)**.** Recall the computation $?nat$, from Example 2.5.3, which produces a natural number nondeterministically. Define:

$$M = \textbf{return } \lambda x{:}\mathbb{1}.?nat \quad : \mathbb{1} \to \mathbb{N}$$

$$P = \textbf{let } ?nat \Rightarrow x \textbf{ in } (\textbf{return } \lambda z{:}\mathbb{1}.\textbf{let } ?nat \Rightarrow y \textbf{ in } min(x,y)) \quad : \mathbb{1} \to \mathbb{N}.$$

Consider the following computation formula in EPCF logic of type $\mathbb{1} \to \mathbb{N}$:

$$\Phi = \Diamond(true \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\}).$$

[1] I am grateful to Niels Voorneveld for pointing out this example.

A computation that satisfies this formula is one such that: at least one of its possible return values is a function which, when given argument $\star$, *may* return *any* natural number. Note that for type $\mathbb{1}$, the only formulas are $true$ and $false$, represented as the empty conjunction and disjunction respectively, and the only value is $\star$.

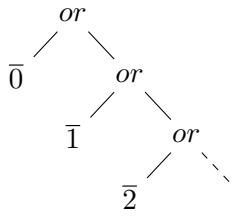
We can show that $M \models \Phi$. We need to check that $|M|[\models true \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\}] \in \Diamond$. Computation M returns immediately so $|M|$ is just a leaf labelled $\lambda x{:}\mathbb{1}.?nat$. The set $\Diamond$ contains the trees with at least one $\star$ leaf. So it is sufficient to check:

$$\lambda x{:}\mathbb{1}.?nat \models true \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\}$$

that is,

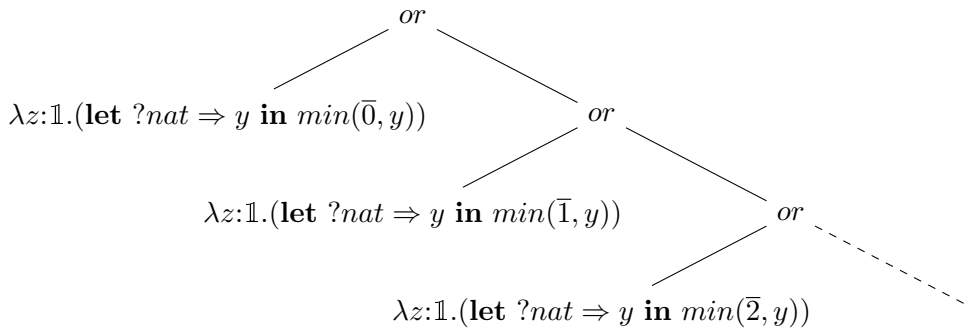
$$?nat \models \wedge_{n\in\mathbb{N}}\Diamond\{n\}.$$

Recall that the computation tree of $?nat$ is:



It has a leaf labelled with each natural number, therefore $?nat \models \wedge_{n\in\mathbb{N}}\Diamond\{n\}$ is true.

On the other hand, $P \not\models \Phi$. To see this note that $|P|$ is:



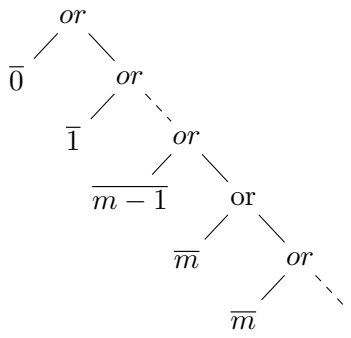
In order for $|P|[\models true \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\}] \in \Diamond$ to be true we need to find some $m \in \mathbb{N}$ such that:

$$\lambda z{:}\mathbb{1}.(\textbf{let } ?nat \Rightarrow y \textbf{ in } min(\overline{m}, y)) \models true \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\}$$

that is,

$$\textbf{let } ?nat \Rightarrow y \textbf{ in } min(\overline{m}, y) \models \wedge_{n\in\mathbb{N}}\Diamond\{n\}.$$

By contradiction, assume that such an m exists. The tree of $\textbf{let } ?nat \Rightarrow y \textbf{ in } min(\overline{m}, y)$ has as leaves all the numbers from 0 to m, but none greater than m:



So it is false that **let** $?nat \Rightarrow y$ **in** $min(\overline{m}, y) \models \Diamond\{m+1\}$. Therefore $P \not\models \Phi$ so M and P are not logically equivalent.

However, M and P are contextually equivalent. Since we have not defined contextual equivalence for EPCF rigorously, we only argue informally.

A context C can compare computations M and P by supplying their return values, which are functions, with arguments and observing the possible results. The context is a syntactic device, hence it is finite. So it can only perform this test a finite number of times.

Computation M returns a function that can generate any natural number, whereas P returns a bounded number generator but *the bound is arbitrarily large*. Because the context can only check for a finite set of natural numbers, say all smaller than k, P and M appear to be equivalent. It is always possible that P returns a bounded number generator with bound larger than k.

In contrast, logical equivalence can test for an infinite number of outcomes at once. This is achieved in the formula Φ by using the infinite conjunction $\wedge_{n\in\mathbb{N}}\Diamond\{n\}$.

2.8 Chapter Summary

Section 2.1 discussed the notions of equivalence relation and compatibility in connection with program equivalence. These properties are useful when establishing equivalence of two particular programs because they allow chaining equations and substituting equals for equals inside an equation. Three kinds of program equivalence were then outlined: contextual equivalence, applicative bisimilarity and equivalence induced by a logic of program properties. These are studied in the rest of the dissertation.

In Section 2.2, algebraic effects were briefly discussed. They specify behaviour such as nondeterminism, probabilistic choice, global state and I/O in a generic way. These behaviours are all triggered by a set of operations, for example, a binary choice operator *or* for nondeterminism. Because an effectful program has multiple possible execution paths, its execution can be pictured as a tree where the nodes are effect operations and the leaves are return values.

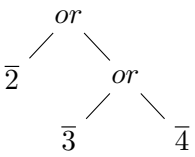
Next, a programming language named EPCF was introduced. It is a call-by-value extension of the simply-typed λ-calculus with recursion, natural numbers and algebraic effects. EPCF makes a syntactic distinction between values, terms which cannot reduce, and computations, which can reduce. Its operational semantics maps a computation to a

tree. For example the computation M where

$$F = \lambda f{:}\mathbb{N}{\to}\mathbb{N}.or(f\ \overline{1},\ or(f\ \overline{2},\ f\ \overline{3}))$$

$$M = F\ (\lambda n{:}\mathbb{N}.\mathbf{return}\ S(n))$$

has tree:



Simpson and Voorneveld [SV18] proposed a modal logic that expresses properties of EPCF programs, named EPCF logic (Section 2.6). For each effect, there is a set of *modalities* that express properties of computations which exhibit that effect. For nondeterminism these are $\Box$ and $\Diamond$. For example, $\Phi_1 = \Box(\{2\} \vee \{3\} \vee \{4\})$ says that computation M *always* returns a result from the set $\{2, 3, 4\}$, whereas $\Phi_2 = \Diamond\{2\}$ says that M *may* return 2.

For functions, logical formulas have the form $\phi \mapsto \Phi$. This says that, if the argument of the function satisfies ϕ, then the resulting application satisfies Φ. For example, F satisfies the following property: $F \models (\{1\} \mapsto \Box\{2\}) \mapsto \Diamond\{2\}$.

Simpson and Voorneveld defined applicative bisimilarity for EPCF using modalities and proved it compatible. They showed that program equivalence induced by EPCF logic coincides with applicative bisimilarity but not with contextual equivalence. Section 2.7 described two EPCF programs exhibiting nondeterminism that are contextually equivalent but not logically equivalent. The problem is that a context can only test a function on a finite number of arguments, whereas a logical formula can test it on infinitely many arguments, using infinitary connectives. Thus, we have identified the main problem of the dissertation: finding a logic that characterises contextual equivalence for a higher-order language with algebraic effects.

Chapter 3

Introducing the ECPS Language

This chapter introduces the programming language ECPS and its operational semantics. ECPS will be used in the rest of the dissertation to study program equivalence. Moreover, a few results that will be used later are outlined: a coinduction proof principle and general intuitions about logical relations.

3.1 A New Language – ECPS

To make it easier to formulate a logic that characterises contextual equivalence, we introduce a new programming language ECPS. It is a variant of EPCF in which programs are written in continuation-passing style (CPS) [Rey93]. This means that functions carry an additional argument, namely the *continuation* to which they pass their result. The intuition is that a continuation specifies how the execution should proceed once a function has finished.

Given a fixed return type R, a continuation has type $\alpha \to R$. It is waiting for an argument of type α to produce a return value of type R. Consider for example a function that adds two natural numbers. Usually, it has type $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$. In continuation-passing style, this function would look like:

$$\texttt{addc} : \texttt{nat} \to \texttt{nat} \to (\texttt{nat} \to R) \to R$$

$$\texttt{addc}\ n\ m\ \mathit{cont} = \mathit{cont}\ (n + m).$$

Instead of directly returning the result $n + m$, the function `addc` passes it to the continuation *cont*.

The key property of ECPS that allows the formulation of the new logic, which will be introduced in Section 6.1, is that functions do not have a return type. Once a function has been applied, the resulting computation is expected to run forever. In other words, the return type R of continuations is chosen to be $\bot$. Thus, programs no longer return values that we can observe. We might however observe termination, which is now treated as an effect, or other side effects such as output values.

Definition 3.1.1 (ECPS)**.** *The types are defined by the following grammar:*

$$A, B \coloneqq \neg(A_1, \ldots, A_n) \mid \texttt{nat} \mid \texttt{unit}.$$

$$\frac{}{\Gamma, x : A \vdash x : A}(\text{VAR}) \quad \frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash t}{\Gamma \vdash \lambda(x_1, \ldots, x_n){:}(A_1, \ldots, A_n).t : \neg(A_1, \ldots, A_n)}(\text{LBD})$$

$$\frac{\Gamma \vdash v : \neg(A_1, \ldots, A_n) \qquad \Gamma \vdash w_1 : A_1 \qquad \ldots \qquad \Gamma \vdash w_n : A_n}{\Gamma \vdash v\ (w_1, \ldots, w_n)}(\text{APP})$$

$$\frac{}{\Gamma \vdash \texttt{zero} : \texttt{nat}}(\text{ZERO}) \quad \frac{\Gamma \vdash v : \texttt{nat}}{\Gamma \vdash \texttt{succ}(v) : \texttt{nat}}(\text{SUCC}) \quad \frac{}{\Gamma \vdash \star : \texttt{unit}}(\text{UNIT})$$

$$\frac{\Gamma \vdash v : \texttt{nat} \qquad \Gamma \vdash t \qquad \Gamma, x : \texttt{nat} \vdash s}{\Gamma \vdash \texttt{case}\ v\ \texttt{in}\ \{\texttt{zero} \Rightarrow t,\ \texttt{succ}(x) \Rightarrow s\}}(\text{CASE})$$

$$\frac{\Gamma, x : \neg(\overrightarrow{A}) \vdash v : \neg(\overrightarrow{A}) \qquad \Gamma \vdash \overrightarrow{w} : \overrightarrow{A}}{\Gamma \vdash (\mu x.v)(\overrightarrow{w})}(\text{MU})$$

$$\frac{\Gamma, x : \texttt{nat} \vdash t \qquad \Gamma \vdash v : \texttt{nat}}{\Gamma \vdash \sigma(v, x.t)}\sigma \in \Sigma\ (\text{OP}) \quad \frac{}{\Gamma \vdash \downarrow}(\text{STOP})$$

Figure 3.1: Typing judgements for ECPS.

Fix a set Σ of effect operations σ, each with arity $\texttt{nat} \times \alpha^{\texttt{nat}} \to \alpha$, where α stands for a computation. Values and computations are defined respectively as:

$$v, w \coloneqq \texttt{zero} \mid \texttt{succ}(v) \mid \star \mid \lambda(x_1, \ldots, x_n){:}(A_1, \ldots, A_n).t \mid x$$

$$t, u \coloneqq v(w_1, \ldots, w_n) \mid (\mu x.v)(\overrightarrow{w}) \mid \sigma(v, x.t) \mid \downarrow \mid \texttt{case}\ v\ \texttt{in}\ \{\texttt{zero} \Rightarrow t,\ \texttt{succ}(x) \Rightarrow u\}.$$

There are two typing relations, one for values $\Gamma \vdash v : \tau$, and one for computations which do not have a type, $\Gamma \vdash t$. These are the least relations closed under the rules in Figure 3.1.

ECPS is a fine-grained call-by-value language; it makes a distinction between values and computations. Ignoring effects, ECPS is in fact a fragment of Levy's Jump-With-Argument programming language [LL07].

The type $\neg(A_1, \ldots, A_n)$ is the type of a function which takes n arguments of types $A_1, \ldots, A_n$ respectively and returns a computation. This computation is not expected to terminate so we can think of $\neg(A_1, \ldots, A_n)$ as $(A_1, \ldots, A_n) \to \bot$. We can also think of $k : \neg A$ as a continuation of type $A \to R$, where the return type of all continuations R has been set to $\bot$. The base types `nat` and `unit` are the same as in EPCF.

In $\mu x.v$, the expression v is constrained by the typing rules to be a function. Thus, intuitively $\mu x.v$ is a recursive definition of the function v, where x represents v and can appear free inside v. Computation $(\mu x.v)(\overrightarrow{w})$ arises by applying this function to $\overrightarrow{w}$.

Example 3.1.2. We can now implement the function `addc` in ECPS using continuations and recursion. The type of this function is:

$$\texttt{addc} : \neg(\texttt{nat},\ \texttt{nat},\ \neg\texttt{nat})$$

$$
\begin{array}{l}
\mathtt{addc} = \lambda(x, y, k) : (\mathtt{nat},\ \mathtt{nat},\ \neg\mathtt{nat}). \\
\quad (\mu f.\ \lambda(x', y', k') : (\mathtt{nat},\ \mathtt{nat},\ \neg\mathtt{nat}). \\
\qquad \mathtt{case}\ x'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow k'\ y', \\
\qquad\qquad \mathtt{succ}(x'') \Rightarrow \mathtt{case}\ y'\ \mathtt{in} \\
\qquad\qquad\qquad \{\mathtt{zero} \Rightarrow k'\ x', \\
\qquad\qquad\qquad\ \mathtt{succ}(y'') \Rightarrow f\ (x'', y'',\ \lambda z : \mathtt{nat}.\ k'\ \mathtt{succ}(\mathtt{succ}(z)))\}\} \\
\quad)\ (x, y, k)\,.
\end{array}
$$

The behaviour of this function can be explained intuitively as follows: if x is zero then the result of the addition is y, so y is passed to the current continuation k. Similarly if y is zero. If x and y are both greater than zero, add $x - 1$ and $y - 1$ and pass the result to continuation $\lambda z : \mathtt{nat}.\ k'\ \mathtt{succ}(\mathtt{succ}(z))$. This continuation adds two to $(x - 1) + (y - 1)$, then passes the result to the current continuation k. This explanation might become clearer if read in conjunction with the operational semantics from the next section.

ECPS does not have a **let** constructor for sequencing. Sequencing can be achieved instead by manipulating the continuation passed to a program.

As discussed above, termination in ECPS is an effect. Its associated effect operation is $\downarrow$, which does not take any arguments.

The different arities for effect operations from EPCF are conflated into the most general one: $\mathtt{nat} \times \alpha^{\mathtt{nat}} \to \alpha$. Therefore, operation σ takes as arguments a natural number v and a function from a natural number x to a computation t, and returns a computation $\sigma(v, x.t)$.

Substitution of values for free variables, $v[w/x]$ and $t[w/x]$, is defined in a standard way by recursion on the structure of v and t. We will use $\overline{n}$ to denote the natural number $\mathtt{succ}^n(\mathtt{zero})$. Let $(\vdash)$ be the set of closed computations and $(\vdash A)$ the set of closed values of type A.

3.2 Operational Semantics of ECPS

Next, we present the operational semantics of ECPS. It does not need to use stacks because any control flow is explicitly encoded inside a computation using continuations.

Definition 3.2.1. *The operational semantics is given by two families of relations on closed computation terms*

$$(\longrightarrow) \subseteq (\vdash) \times (\vdash)$$

$$(\xrightarrow{\sigma(v)}) \subseteq (\vdash) \times (\vdash)^{\mathbb{N}} \quad \textit{for any } \sigma \in \Sigma \textit{ and } \vdash v : \mathtt{nat}$$

defined as:

$$\sigma(v, x.t) \xrightarrow{\sigma(v)} (t[\overline{n}/x])_{n \in \mathbb{N}}$$

$$(\lambda(\overrightarrow{x}){:}(\overrightarrow{A}).t)\ (\overrightarrow{w}) \longrightarrow t[\overrightarrow{w}/\overrightarrow{x}]$$

$$(\mu x.v)\ (\overrightarrow{w}) \longrightarrow (v[(\lambda(\overrightarrow{y}){:}(\overrightarrow{A}).(\mu x.v)(\overrightarrow{y}))/x])\ (\overrightarrow{w})$$

$$\mathtt{case}\ \mathtt{zero}\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \longrightarrow s$$

$$\mathtt{case}\ \mathtt{succ}(v)\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \longrightarrow t[v/x].$$

Denote by $\longrightarrow^*$ the reflexive-transitive closure of $\longrightarrow$.

We can see that the $\longrightarrow$ reduction relation can only get stuck when encountering an effect operation $\sigma(\ldots)$ or $\downarrow$. It might also be the case that $\longrightarrow$ never terminates.

There are no reduction rules of any kind for $\downarrow$ since it signifies termination. For $\sigma \in \Sigma$, the $\xrightarrow{\sigma(v)}$ reduction rule has on its right-hand-side a set of computations, one for each natural number. Therefore, repeated applications of this rule lead to the construction of an infinitely branching tree.

Given this observation we can define effect trees for ECPS. Denote the set of all effect trees by Trees_Σ. A tree in this set can have:

- a leaf labelled $\bot$, which signifies nontermination of $\longrightarrow$;
- a leaf labelled $\downarrow$, which signifies termination;
- nodes labelled σ_n, where $\sigma \in \Sigma$ and $n \in \mathbb{N}$; such a node has an infinite number of children $t_0, t_1, \ldots$.

We can define a partial order on Trees_Σ which makes it an ω-CPO. This is similar to the order on EPCF trees:

$$tr_1 \leq tr_2 \quad \Longleftrightarrow \quad tr_1 \text{ can be obtained by replacing subtrees of } tr_2 \text{ by } \bot.$$

Using this order, we can give a domain theoretic definition of the tree associated to a closed computation. The tree might have infinite depth and width.

Definition 3.2.2 (Computation trees for ECPS)**.** *Define a family of maps*

$$[\![-]\!]_{(-)} : (\vdash) \times \mathbb{N} \longrightarrow \mathit{Trees}_\Sigma$$

$$[\![t]\!]_0 = \bot$$

$$[\![t]\!]_{n+1} = \begin{cases} [\![s]\!]_n & \text{if } t \longrightarrow s \\ \sigma_m([\![s[\overline{0}/x]]\!]_n, \ldots, [\![s[\overline{k}/x]]\!]_n, \ldots) & \text{if } t \xrightarrow{\sigma(\overline{m})} (s[\overline{k}/x])_{k\in\mathbb{N}} \\ \downarrow & \text{if } t = \downarrow \\ \bot & \text{otherwise.} \end{cases}$$

We can see that $[\![t]\!]_n \leq [\![t]\!]_{n+1}$ so we can define $[\![-]\!] : (\vdash) \longrightarrow \mathit{Trees}_\Sigma$ as the least upper bound of the chain $\{[\![t_n]\!]\}_{n\in\mathbb{N}}$:

$$[\![t]\!] = \bigsqcup_{n\in\mathbb{N}} [\![t]\!]_n.$$

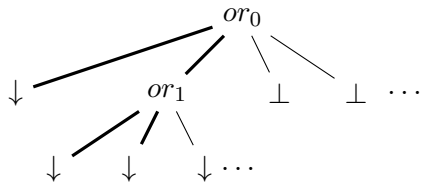
Example 3.2.3 (Pure functional computation)**.** In this case, the signature Σ is empty and the only effect operation is $\downarrow$. Therefore, computation trees can only be leaves: $\downarrow$ for a computation that terminates and $\bot$ for one that does not. For example:

$$[\![\mathit{loop}]\!] = [\![(\mu f.\lambda x{:}\mathtt{nat}.(f\ x))\ \mathtt{zero}]\!] = \bot.$$

Example 3.2.4 (Nondeterminism)**.** Define $\Sigma = \{or\}$. All effects have arity $\mathtt{nat} \times \alpha^{\mathtt{nat}} \to \alpha$. The intuitive interpretation of $or(v, x.t)$ is that it ignores v and performs a nondeterministic choice between $t[\overline{0}/x]$ and $t[\overline{1}/x]$.

For example, consider the computation tree of:

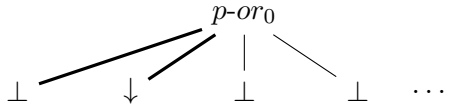
$$[\![or(\overline{0},\ x.\mathtt{case}\ x\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow \downarrow,\ \mathtt{succ}(y) \Rightarrow \mathtt{case}\ y\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow or(\overline{1}, z.\downarrow),\ \mathtt{succ}(w) \Rightarrow loop\}\})]\!] =$$



As far as the intuitive interpretation of this computation is concerned, the indices 0 and 1 are irrelevant, and only the paths highlighted in bold can occur. However, $[\![-]\!]$ treats *or* as uninterpreted syntax, hence the paths that can never occur are still present in the effect tree.

Example 3.2.5 (Probabilistic choice)**.** Define $\Sigma = \{p\text{-}or\}$. Intuitively, the operation $p\text{-}or(v, x.t)$ chooses between $t[\overline{0}/x]$ and $t[\overline{1}/x]$ with probability 0.5. The rest of the branches can never occur. The following computation:

$$[\![p\text{-}or(\overline{0},\ x.\mathtt{case}\ x\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow loop,\ \mathtt{succ}(y) \Rightarrow \mathtt{case}\ y\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow \downarrow,\ \mathtt{succ}(z) \Rightarrow loop\}\})]\!] =$$

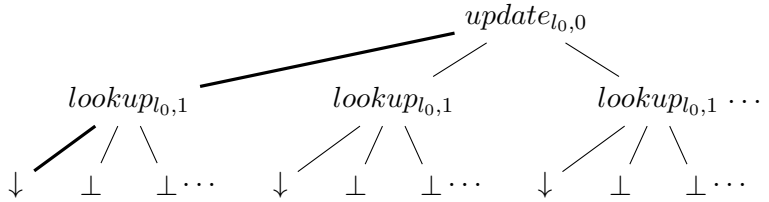


is analogous to the EPCF computation $p\text{-}or(loop, \mathbf{return}\ \star)$ from Example 2.5.4, irrespective of the index of *p-or*, here 0.

Example 3.2.6 (Global store)**.** There is a finite set of locations $\mathbb{L}$ that can store natural numbers and $\Sigma = \{lookup_l,\ update_l \mid l \in \mathbb{L}\}$. The intuitive interpretation of $lookup_l(v, x.t)$ is that it ignores v, it looks up the value at location l, if this is $\overline{n}$ it continues with $t[\overline{n}/x]$. For $update_l(v, x.t)$ the intuition is: write the number v in location l then continue with the computation $t[\overline{0}/x]$.

The EPCF computation tree from Example 2.5.5 can be adapted here:

$$[\![update_{l_0}(\overline{0},\ x.lookup_{l_0}(\overline{1},\ x'.\mathtt{case}\ x'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow \downarrow,\ \mathtt{succ}(y) \Rightarrow loop\}))]\!] =$$

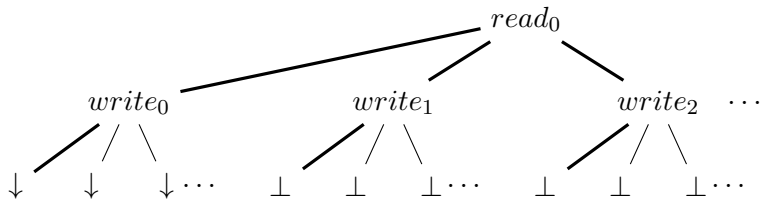


Only the path in bold can occur in the computation above.

Example 3.2.7 (Input/output)**.** Define $\Sigma = \{read, write\}$. Intuitively, the computation $read(v, x.t)$ ignores v, accepts as input a number $\overline{n}$ and continues with $t[\overline{n}/x]$. The computation $write(v, x.t)$ writes v to the output channel then continues with computation $t[\overline{0}/x]$.

Below is a computation that inputs a number then outputs it immediately. Only the paths in bold can occur:

$$[\![read(\overline{0},\ x.write(x,\ x'.\texttt{case}\ x\ \texttt{in}\ \{\texttt{zero} \Rightarrow \downarrow,\ \texttt{succ}(y) \Rightarrow loop\}))]\!] =$$



3.3 A Coinduction Proof Principle

Coinduction is a proof techniques that will be used in the following chapter. We give an abstract overview of a coinduction proof principle using the basic notions of a category, functor, terminal object and coalgebra, all of which can be found in an introduction to category theory such as [AT18]. The following definition appears in [JR11]:

Definition 3.3.1. *Let T : **Set** $\Longrightarrow$ **Set** be a functor. Take two T-coalgebras $(X,\ a_X : X \longrightarrow T(X))$ and $(Y,\ a_Y : Y \longrightarrow T(Y))$. A* T-bisimulation *between $(X,\ a_X)$ and $(Y,\ a_Y)$ is a relation $\mathcal{R} \subseteq X \times Y$ for which there exists a T-coalgebra structure $g : \mathcal{R} \longrightarrow T(\mathcal{R})$ such that the two projection functions $\pi_1 : \mathcal{R} \longrightarrow X$ and $\pi_2 : \mathcal{R} \longrightarrow Y$ are T-coalgebra morphisms:*

$$\begin{array}{ccccc} X & \xleftarrow{\pi_1} & \mathcal{R} & \xrightarrow{\pi_2} & Y \\ \downarrow{\scriptstyle a_X} & & \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle a_Y} \\ T(X) & \xleftarrow{T(\pi_1)} & T(\mathcal{R}) & \xrightarrow{T(\pi_2)} & T(Y) \end{array}$$

Use the following notation:

$$x \cong y \quad \Longleftrightarrow \quad \text{there exists a } T\text{-bisimulation } \mathcal{R} \subseteq X \times Y \text{ with } (x, y) \in \mathcal{R}.$$

We can now formulate the following coinduction proof principle:

Proposition 3.3.2 (From [JR11])**.** *Consider the final T-coalgebra $(Z,\ c : Z \longrightarrow T(Z))$, if it exists. Let $\alpha_X : X \longrightarrow Z$ and $\alpha_Y : Y \longrightarrow Z$ be coalgebra morphisms. For all $x \in X$ and $y \in Y$:*

$$\text{if } x \cong y \text{ then } \alpha_X(x) = \alpha_Y(y).$$

Proof. It suffices to show that the following diagram commutes, where $\mathcal{R} \subseteq X \times Y$ is a bisimulation and $(x, y) \in \mathcal{R}$:

$$\begin{array}{ccccccccc} Z & \xleftarrow{\alpha_X} & X & \xleftarrow{\pi_1} & \mathcal{R} & \xrightarrow{\pi_2} & Y & \xrightarrow{\alpha_Y} & Z \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle a_X} & & \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle a_Y} & & \downarrow{\scriptstyle c} \\ T(Z) & \xleftarrow{T(\alpha_X)} & T(X) & \xleftarrow{T(\pi_1)} & T(\mathcal{R}) & \xrightarrow{T(\pi_2)} & T(Y) & \xrightarrow{T(\alpha_Y)} & T(Z) \end{array}$$

This is true because all the small squares commute. Then both $\alpha_X \circ \pi_1$ and $\alpha_Y \circ \pi_2$ are coalgebra morphisms into Z. By finality of Z, they must be equal. □

3.4 Logical Relations

"Logical relations" is a proof technique that involves defining a family of relations by induction on the types of a programming language. A relation for type τ contains only pairs of terms of type τ. The relation is *logical* if, given related functions f_1 and f_2 of type $\rho \to \tau$ and related arguments $x_1 : \rho$ and $x_2 : \rho$, the terms $f_1\ x_1$ and $f_2\ x_2$ are related.

Logical predicates, the unary version of logical relations, have been used to prove strong normalisation of the simply-typed λ-calculus [Tai67], [GTL89, Chapter 6]. Other versions of logical relations have been used, for example, to characterise program equivalence [Ahm06] and to prove compiler correctness [BH09]. These are examples of syntactic logical relations, based on the operational semantics of a language. These are the kinds of relations used in the next chapter to prove that a continuation-passing translation of EPCF into ECPS is correct.

There are also logical relations based on denotational models (e.g. [Pit96, Fio17]). One of their applications is proving that a denotational model of a programming language in computationally adequate. Adequacy, means that denotational equality is a sound technique for establishing contextual equivalence of programs.

The particular flavour of logical relations we will need is *step-indexed biorthogonal* logical relations. As Jaber and Tabareau explain [JT11], biorthogonality allows us to define which terms should be related by specifying their interaction with program contexts.

A biorthogonal logical relation contains a collection of relations on values, $\mathcal{V}_\tau$, for each type τ. There is a collection of relations on program contexts defined using the relations on values:

$$\mathcal{C}_\tau = \{(C_1, C_2) \mid \forall (v_1, v_2) \in \mathcal{V}_\tau.\ \mathfrak{O}(C_1[v_1], C_2[v_2])\}.$$

And finally, a collection of relations on terms defined using the relations on contexts:

$$\mathcal{T}_\tau = \{(t_1, t_2) \mid \forall (C_1, C_2) \in \mathcal{C}_\tau.\ \mathfrak{O}(C_1[t_1], C_2[t_2])\}.$$

The notation $\mathfrak{O}(C_1[t_1], C_2[t_2])$ stands for an observation about programs $C_1[t_1]$ and $C_2[t_2]$. The notion of observation is chosen on a case-by-case basis, but it usually involves the reduction behaviour of the terms under consideration.

Step-indexing was introduced to deal with programming languages with recursion. This approach, instead of defining a single relation for type τ, defines a family of relations for type τ indexed by natural numbers. Intuitively, terms in $\mathcal{T}_\tau^n$ are allowed to reduce at most n steps. The natural number indices help to break the vicious circle introduced by recursion in proofs about the logical relation.

Step-indexing and biorthogonality can be combined in a straightforward way as explained for example by Pitts [Pit10]. Because EPCF contains recursion, and because computations can only be evaluated in a stack, it is useful to use both step-indexing and biorthogonality when proving correctness of the translation from EPCF to ECPS. This will be explained in the next chapter.

3.5 Chapter Summary

This chapter introduced a new language ECPS in Sections 3.1 and 3.2. ECPS is a continuation-passing variant of EPCF which will be used in the rest of the dissertation to study program equivalence.

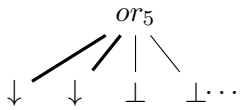
Being a continuation-passing language means that functions receive an additional argument, which specifies how the computation should proceed once the function has terminated. For example, the successor function in ECPS is:

$$f = \lambda(n,k){:}(\mathtt{nat}, \neg\mathtt{nat}).(k\ \mathtt{succ}(n)) : \neg(\mathtt{nat}, \neg\mathtt{nat}). \tag{3.5.1}$$

Here $k : \neg\mathtt{nat}$ is a *continuation*, a function that is waiting for a result of type $\mathtt{nat}$, but is not expected to return.

Because everything is written in continuation-passing style, ECPS computations do not usually return. Therefore, the operational semantics maps an ECPS computation to a tree whose nodes are effect operations and leaves are either $\downarrow$, which signifies the termination effect, or $\bot$ for nontermination. For example, consider the tree of:

$$m = (\lambda(n,k){:}(\mathtt{nat}, \neg\mathtt{nat}).\ or(\overline{5},\ y.\mathtt{case}\ y\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow (k\ \mathtt{succ}(n)),\ \mathtt{succ}(y') \Rightarrow \mathtt{case}\ y'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow (k\ \overline{0}),\ \mathtt{succ}(y'') \Rightarrow loop\}\}))\ (\overline{3},\ \lambda x{:}\mathtt{nat}.\downarrow)$$



In ECPS all effect operations have arity $\mathtt{nat} \times \alpha^{\mathtt{nat}} \to \alpha$, where α stands for a computation. Therefore, each node in a tree has a child for each natural number. In the case of *or*, the tree carries redundant information: only the paths in bold can occur during computation, and the index 5 is irrelevant.

Finally, two proof methods were reviewed: coinduction (Section 3.3) and logical relations (Section 3.4), both of which will be used in the next chapter.

Chapter 4

Comparison: ECPS vs. EPCF

This chapter presents a continuation-passing translation from the language PCF with effects (EPCF), introduced in Chapter 2, to its continuation-passing variant (ECPS), defined in Chapter 3. This translation is proved correct using logical relations and coinduction. The last section is an informal argument for why translating ECPS into EPCF is not possible. The contents of this chapter justify the choice of the ECPS language, so the following chapters are only concerned with ECPS.

4.1 CPS Translation

Preliminary ideas for the translation from EPCF to ECPS appear in a technical report by Lafont, Reus and Streicher [LRS93], but for simpler languages. In order to simplify the translation, we replace all the EPCF effect operations by operations with arity $\mathbb{N}\times\alpha^{\mathbb{N}} \to \alpha$, as follows:

$$\sigma(M_0,\ldots,M_{n-1}) \text{ becomes } \sigma(Z;W)$$

$$\sigma(V;M_0,\ldots,M_{n-1}) \text{ becomes } \sigma(V;W)$$

$$\sigma(V) \text{ becomes } \sigma(Z;V)$$

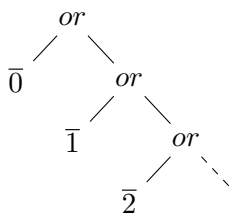
where

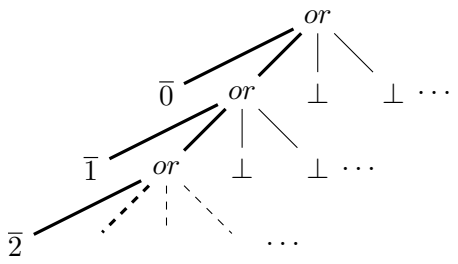
$$\begin{aligned} W = \lambda x{:}\mathbb{N}.\mathbf{case}\ x\ \mathbf{in}\ \{Z \Rightarrow M_0,\ S(x_1) \Rightarrow \\ \mathbf{case}\ x_1\ \mathbf{in}\ \{Z \Rightarrow M_1,\ S(x_2) \Rightarrow \ldots \\ \mathbf{case}\ x_{n-1}\ \mathbf{in}\ \{Z \Rightarrow M_{n-1},\ S(x_n) \Rightarrow \mathit{loop}\}\ldots\}\} \end{aligned}$$

$$\mathit{loop} = \mathbf{let}\ (\mathbf{fix}\ \lambda f{:}\mathbb{N}\to\tau.\mathbf{return}\ f) \Rightarrow x\ \mathbf{in}\ x\ Z. \qquad (\text{where } \tau \text{ is the type of } M_i)$$

The function W chooses between the arguments $M_0,\ldots,M_{n-1}$, just as σ used to do. This simplification means that EPCF computation trees carry more redundant information. For example:

Example 4.1.1 (Nondeterminism)**.** The computation tree of $?nat$

$|?nat| =$ 

becomes 

where only the paths in bold can occur. The definitions of modalities in $\mathcal{O}$ need to be adjusted to take this redundancy into account. For the effects considered in the dissertation, Scott-openness, decomposability and program equivalence are preserved.

The CPS translation for types is defined as follows:

$$\mathbb{N}^* = \mathtt{nat}$$

$$\mathbb{1}^* = \mathtt{unit}$$

$$(\rho \to \tau)^* = \neg(\rho^*, \neg\tau^*).$$

To translate contexts, assume that all variable names from EPCF appear in ECPS:

$$\emptyset^* = \emptyset$$

$$(\Gamma, x : \tau)^* = \Gamma^*, x : \tau^*.$$

The CPS translation for values and computations is given in Figures 4.1 and 4.2. It relates values in context $\Gamma \vdash V : \tau$ to ECPS values $\Gamma^* \vdash V^* : \tau^*$. A function $F = \lambda x{:}\rho.M$ of type $\rho \to \tau$ is translated to $F^* = \lambda(x, k){:}(\rho^*, \neg\tau^*).M^*\ k$ of type $\neg(\rho^*, \neg\tau^*)$. The function F^* is waiting for an argument of type ρ^* and a continuation of type $\neg\tau^*$. The continuation is itself waiting for the result of computation M.

A computation $\Gamma \vdash M : \tau$ is translated to a function $\Gamma^* \vdash M^* : \neg\neg\tau^*$. This is because M^* is waiting for a continuation of type $\neg\tau^*$ to which it can pass its result.

Computation $\mathbf{return}\ V : \tau$ becomes $\lambda k{:}\neg\tau^*.k\ V^*$. So value V^* is passed to the current continuation. Application $V\ W : \tau$ becomes $\lambda k{:}\neg\tau^*.V^*\ (W^*, k)$, which means that V^* is given arguments W^* and a continuation that is waiting for the result of the application. Sequencing $\mathbf{let}\ M \Rightarrow x\ \mathbf{in}\ N : \tau$ is implemented as $\lambda k{:}\neg\tau^*.M^*\ \lambda x{:}\rho^*.(N^*\ k)$. First, M^* is evaluated and its result is passed to the continuation $\lambda x{:}\rho^*.(N^*\ k)$ which executes N^* and passes the result to k.

To translate effects, assume that the set of effect operations Σ is the same in EPCF and ECPS. The translation of $\sigma(V; W) : \tau$ is as expected, the function W^* is passed a continuation in addition to argument x.

The translation of $(\mathbf{fix}\ F) : \tau \to \rho$ is complicated by the reduction behaviour of $\mathbf{fix}\ F$. Recall that:

$$\mathbf{fix}\ F \rightsquigarrow \mathbf{return}\ \lambda x{:}\tau.\mathbf{let}\ F\ (\lambda y{:}\tau.\mathbf{let}\ \mathbf{fix}\ F \Rightarrow z\ \mathbf{in}\ z\ y) \Rightarrow w\ \mathbf{in}\ w\ x.$$

As a sanity check for the translation, it can be shown that

$$(\mathbf{fix}\ F)^*\ k \longrightarrow^* k\ (\lambda x{:}\tau.\mathbf{let}\ F\ (\lambda y{:}\tau.\mathbf{let}\ \mathbf{fix}\ F \Rightarrow z\ \mathbf{in}\ z\ y) \Rightarrow w\ \mathbf{in}\ w\ x)^*$$

$$(\Gamma \vdash Z : \mathbb{N})^* = \Gamma^* \vdash \texttt{zero} : \texttt{nat}$$

$$(\Gamma \vdash S(V) : \mathbb{N})^* = \Gamma^* \vdash \texttt{succ}(V^*) : \texttt{nat}$$

$$(\Gamma \vdash x : \tau)^* = \Gamma^* \vdash x : \tau^*$$

$$(\Gamma \vdash \star : \mathbb{1})^* = \Gamma^* \vdash \star : \texttt{unit}$$

$$(\Gamma \vdash \lambda x{:}\rho.M : \rho \to \tau)^* = \Gamma^* \vdash \lambda(x,k){:}(\rho^*, \neg\tau^*).(M^*\ k) : \neg(\rho^*, \neg\tau^*)$$

$$(\Gamma \vdash V\ W : \tau)^* = \Gamma^* \vdash \lambda k{:}\neg\tau^*.V^*\ (W^*, k) : \neg\neg\tau^*$$

$$(\Gamma \vdash \textbf{return}\ V : \tau)^* = \Gamma^* \vdash \lambda k{:}\neg\tau^*.(k\ V^*) : \neg\neg\tau^*$$

$$(\Gamma \vdash \textbf{let}\ M \Rightarrow x\ \textbf{in}\ N : \tau)^* = \Gamma^* \vdash \lambda k{:}\neg\tau^*.M^*\ \lambda x{:}\rho^*.(N^*\ k) : \neg\neg\tau^*$$

$$(\Gamma \vdash \textbf{case}\ V\ \textbf{in}\ \{Z \Rightarrow M,\ S(x) \Rightarrow N\} : \tau)^* =$$

$$\Gamma^* \vdash \lambda k{:}\neg\tau^*.\texttt{case}\ V^*\ \texttt{in}\ \{\texttt{zero} \Rightarrow M^*\ k,\ \texttt{succ}(x) \Rightarrow N^*\ k\} : \neg\neg\tau^*$$

$$(\Gamma \vdash \sigma(V;W) : \tau)^* = \Gamma^* \vdash \lambda k{:}\neg\tau^*.\sigma(V^*, x.W^*(x,k)) : \neg\neg\tau^*$$

Figure 4.1: CPS translation of EPCF into ECPS – first part.

$$\begin{array}{l}
(\Gamma \vdash \textbf{fix}\ F : \tau \to \rho)^* = \\
\Gamma^* \vdash \lambda k{:}\neg(\neg(\tau^*, \neg\rho^*)). \\
\quad (\mu x.\lambda c{:}\neg(\neg(\tau^*, \neg\rho^*)). \\
\qquad c\ \lambda(x', k'){:}(\tau^*, \neg\rho^*). \\
\qquad\quad \lambda k''{:}\neg\rho^*.\lambda l{:}\neg(\neg(\tau^*, \neg\rho^*)).(F^*\ (\lambda(y, l'){:}(\tau^*, \neg\rho^*). \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda p{:}\neg\rho^*.x\ (\lambda z{:}\neg(\tau^*, \neg\rho^*).(\lambda p'{:}\neg\rho^*.z\ (y, p'))\ p)\ l',\ l)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda w{:}\neg(\tau^*, \neg\rho^*).((\lambda l''{:}\neg\rho^*.w\ (x', l''))\ k'') \\
\qquad\quad k' \\
\quad)\ k \\
\quad : \neg\neg(\neg(\tau^*, \neg\rho^*))
\end{array}$$

Figure 4.2: CPS translation of EPCF into ECPS – continued.

which matches:

$$(\mathbf{return}\ \lambda x{:}\tau.\mathbf{let}\ F\ (\lambda y{:}\tau.\mathbf{let\ fix}\ F \Rightarrow z\ \mathbf{in}\ z\ y) \Rightarrow w\ \mathbf{in}\ w\ x)^*\ k \longrightarrow$$
$$k\ (\lambda x{:}\tau.\mathbf{let}\ F\ (\lambda y{:}\tau.\mathbf{let\ fix}\ F \Rightarrow z\ \mathbf{in}\ z\ y) \Rightarrow w\ \mathbf{in}\ w\ x)^*.$$

4.2 Typing and CPS Translation for Stacks

In the CPS translation $(\mathbf{return}\ V)^* = \lambda k{:}\neg\tau^*.k\ V^*$, the continuation k plays a similar role to the stack $S \circ \mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M$, in the reduction:

$$(S \circ \mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M,\ \mathbf{return}\ V) \rightarrowtail (S,\ M[V/x]).$$

This suggests that EPCF stacks can be translated to ECPS continuations. Since the CPS translation is typed, we allow stacks to have free variables and introduce typing judgements for stacks. The typing judgement $\Gamma \vdash S : \tau \Rightarrow \rho$ says that by substituting a *closed* computation M of type τ for the hole in S we obtain a computation $\Gamma \vdash S\{M\} : \rho$.

$$\frac{}{\Gamma \vdash id : \rho \Rightarrow \rho}(\textsc{sid}) \qquad \frac{\Gamma, x : \tau \vdash M : \tau' \qquad \Gamma \vdash S : \tau' \Rightarrow \rho}{\Gamma \vdash S \circ (\mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M) : \tau \Rightarrow \rho}(\textsc{slet})$$

The substitution of closed values for free variables can be extended to stacks:

$$id[V/y] = id$$

$$(S \circ \mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M) = S[V/y] \circ (\mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M[V/y]).$$

Now EPCF stacks $\Gamma \vdash S : \tau \Rightarrow \rho$ can be translated to ECPS function values $\Gamma^* \vdash S^* : \neg\tau^*$, which are in fact continuations:

$$(\Gamma \vdash id : \rho \Rightarrow \rho)^* = \Gamma^* \vdash \lambda x{:}\rho^*.\downarrow : \neg\rho^*$$

$$(\Gamma \vdash S \circ (\mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M) : \tau \Rightarrow \rho)^* = \Gamma^* \vdash (\lambda x{:}\tau^*.M^*\ S^*) : \neg\tau^*.$$

The empty stack id is equivalent to a continuation which given any value terminates. Here, we can see an important difference between EPCF and ECPS: the stack id returns the value given to it, while id^* discards the value so we cannot observe it. The translation of $S \circ (\mathbf{let}\ (-) \Rightarrow x\ \mathbf{in}\ M)$ is a continuation which, when given value x, executes M^* and passes the result to S^*.

4.3 Correctness of the CPS Translation

The goal of this section is to prove that EPCF computation trees preserve their shape when translated into ECPS. This is stated as:

Theorem 4.3.1 (The CPS translation is correct)**.** *For any closed computation $M : \tau$ in EPCF and any stack $S : \tau \Rightarrow \rho$ the following holds:*

$$[\![M^*\ S^*]\!] = |S, M|[\downarrow /l_1, \downarrow /l_2, \ldots].$$

That is, all the value leaves of $|S, M|$ are replaced by $\downarrow$, but the nodes and the $\bot$-leaves stay the same.

Example 4.3.2 (Probabilistic choice)**.** As explained in Section 4.1, EPCF operation *p-or* is replaced by its version with arity $\mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha$. So computation $p\text{-}or(loop, \mathbf{return}\ \overline{3})$ becomes:

$$p\text{-}or(Z;\ \lambda x{:}\mathbb{N}.\mathbf{case}\ x\ \mathbf{in}\ \{Z \Rightarrow loop,\ S(x') \Rightarrow \mathbf{case}\ x'\ \mathbf{in}\ \{Z \Rightarrow \mathbf{return}\ \overline{3},\ S(x'') \Rightarrow loop\}\}).$$

However, we will still use notation $p\text{-}or(loop, \mathbf{return}\ \overline{3})$ for convenience, even when the version with arity $\mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha$ is meant. Its computation tree is:

$$|id,\ p\text{-}or(loop, \mathbf{return}\ \overline{3})| = \quad p\text{-}or_0 \quad \text{with children } \bot,\ \overline{3},\ \bot,\ \bot\cdots$$

According to Theorem 4.3.1, when translated to ECPS, this computation tree becomes:

$$\begin{aligned}
&[\![(p\text{-}or(loop, \mathbf{return}\ \overline{3}))^*\ id^*]\!] = \\
&\quad [\![(\lambda k{:}\mathtt{nat}.\ p\text{-}or(\overline{0},\ y.(\lambda(x,k'){:}(\mathtt{nat}, \neg\mathtt{nat}).\\
&\qquad\qquad (\lambda k''{:}\mathtt{nat}.\ \mathtt{case}\ x\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow loop^*\ k'',\\
&\qquad\qquad\quad \mathtt{succ}(x') \Rightarrow (\lambda k'''{:}\mathtt{nat}.\ \mathtt{case}\ x'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow (\lambda l{:}\mathtt{nat}.l\ \overline{3})\ k''',\\
&\qquad\qquad\qquad\qquad \mathtt{succ}(x'') \Rightarrow loop^*\ k'''\}\\
&\qquad\qquad\qquad) \ k''\}\\
&\qquad\qquad) \ k'\\
&\qquad\quad) \ (y,k)\\
&\qquad)\\
&\quad) \ (\lambda x{:}\mathtt{nat}.\downarrow)]\!] = \quad p\text{-}or_0 \quad \text{with children } \bot,\ \downarrow,\ \bot,\ \bot\cdots
\end{aligned}$$

In order to prove Theorem 4.3.1, we introduce a step-indexed biorthognal logical relation, as discussed in Section 3.4. The purpose of the logical relation is to relate EPCF and ECPS computations that have similar reduction behaviour. Because EPCF computations reduce in a stack, it is useful to use biorthogonality, and have a relation between stacks and continuations as well. Step-indexing is needed to deal with the **fix** constructor.

The notion of observation $\mathfrak{O}((S, M),\ t)$ that the logical relation uses is a *step-indexed similarity* relation $\mathcal{S}$:

$$\mathcal{S}^n_{\tau,\rho} \subseteq (Stack(\tau,\rho) \times Comp(\tau)) \times (\vdash)$$

defined for each pair of EPCF types τ, ρ and each $n \in \mathbb{N}$. This is not related to the applicative similarity discussed before because it contains pairs of programs from two different languages. However, we will show that $\mathcal{S}$ is related to the abstract notion of bisimulation from Section 3.3. First, we recall the definition of a step-indexed relation from [Pit10] and then define step-indexed similarity:

Definition 4.3.3. *A step-indexed relation on the set X is an $\mathbb{N}$-indexed family of sets:*

$$R = (R_n \mid n \in \mathbb{N}) \text{ satisfying } X \supseteq R_0 \supseteq R_1 \supseteq R_2 \supseteq \ldots.$$

Definition 4.3.4. *The step-indexed similarity relation $\mathcal{S}$ is the family of greatest relations such that $(S, M)\ \mathcal{S}^n_{\tau,\rho}\ t$ implies:*

1. $\forall p < n.\ ((S, M) \rightarrowtail^{p} (S', \sigma(V; W)) \implies t \longrightarrow^{*} \sigma(v, x.t')$ *where* $V = \overline{m}$ *and* $v = \overline{m}$ *and* $\forall l \in \mathbb{N}.\ (S', W\ \bar{l})\ \mathcal{S}^{n-p}_{\tau',\rho}\ t'[\bar{l}/x])$.

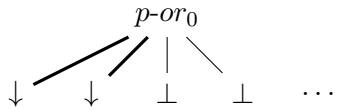
2. $\forall p \leq n.\ ((S, M) \rightarrowtail^{p} (id, \mathbf{return}\ V)$ *for some* $V : \rho \implies \tau \longrightarrow^{*}\downarrow)$.

The statement $(S, M)\ \mathcal{S}^{n}_{\tau,\rho}\ t$ means that the ECPS computation t simulates the reduction of the EPCF configuration (S, M) for n steps. The novel feature of this notion of observation is that it tracks effect operations, not only the termination behaviour of related programs, as compared to previous work on logical relations [Ahm06, BH09, Pit10, JT11].

Example 4.3.5 (Probabilistic choice)**.** For example, the ECPS computation:

$$t = (\lambda k{:}\neg\mathtt{nat}.\ \textit{p-or}(\overline{0},\ x.\mathtt{case}\ x\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow k\ \overline{2}\ ,\ \mathtt{succ}(x') \Rightarrow \mathtt{case}\ x'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow k\ \overline{3},\ \mathtt{succ}(x'') \Rightarrow \mathit{loop}\}\}))\ (\lambda x{:}\mathtt{nat}.\downarrow)$$

with computation tree



and EPCF computation $M = \textit{p-or}(\mathit{loop}, \mathbf{return}\ \overline{3})$ satisfy $(id, M)\ \mathcal{S}^{n}_{\mathbb{N},\mathbb{N}}\ t$ for any $n > 0$.

Intuitively, as the index n increases, step-indexed similarity checks for more and more matching reduction steps. Therefore, $\mathcal{S}$ respects the definition of a step-indexed relation. Below is a useful fact about step-indexed similarity which will be used later. The proof appears in Appendix A.

Lemma 4.3.6. *If* $(S, M) \rightarrowtail^{k} (S', M')$ *and* $t \longrightarrow^{*} t'$ *where* $k < n$ *then:*

$$(S, M)\ \mathcal{S}^{n}_{\tau,\rho}\ t \iff (S', M')\ \mathcal{S}^{n-k}_{\tau,\rho}\ t'.$$

We can define a well-typed relation $\mathcal{S}$ which encodes the fact that an ECPS computation t simulates configuration (S, M) for *any* number of reduction steps. Step-indexed similarity implies this new notion of similarity.

Definition 4.3.7. *Define similarity as the family* $\mathcal{S}$ *of greatest relations indexed by EPCF types* τ *and* ρ *such that* $(S, M)\ \mathcal{S}_{\tau,\rho}\ t$ *implies:*

1. $(S, M) \rightarrowtail^{*} (S', \sigma(V; W)) \implies t \longrightarrow^{*} \sigma(v, x.t')$ *where* $V = \overline{m}$ *and* $v = \overline{m}$ *and* $\forall l \in \mathbb{N}.\ (S', W\ \bar{l})\ \mathcal{S}_{\tau',\rho}\ t'[\bar{l}/x]$.

2. $(S, M) \rightarrowtail^{*} (id, \mathbf{return}\ V)$ *for some* $V : \rho \implies \tau \longrightarrow^{*}\downarrow$.

Lemma 4.3.8. *For any configuration* (S, M) *and closed computation* t*:*

$$(\forall n \in \mathbb{N}.\ (S, M)\ \mathcal{S}^{n}_{\tau,\rho}\ t) \implies (S, M)\ \mathcal{S}_{\tau,\rho}\ t.$$

Proof. The strategy is to show $\bigcap_{n\in\mathbb{N}} \mathcal{S}^{n}$ is a simulation relating (S, M) and t, and thus it is included in $\mathcal{S}$. The rest of the proof appears in Appendix A. □

Now we are ready to define the logical relation. It is in fact a collection of relations $(\mathcal{R}_\tau^{\mathfrak{v},n}, \mathcal{R}_\tau^{\mathfrak{c},n}, \mathcal{R}_{\tau,\rho}^{\mathfrak{s},n})$ for all EPCF types τ, ρ and $n \in \mathbb{N}$.

$$\mathcal{R}_\tau^{\mathfrak{v},n} \subseteq Val(\tau) \times (\vdash \tau^*)$$

$$\mathcal{R}_\tau^{\mathfrak{c},n} \subseteq Comp(\tau) \times (\vdash \neg\neg\tau^*)$$

$$\mathcal{R}_{\tau,\rho}^{\mathfrak{s},n} \subseteq Stack(\tau,\rho) \times (\vdash \neg\tau^*).$$

It is defined by well-founded induction on the natural numbers and on EPCF types, following a tutorial paper by Pitts [Pit10] which deals with the untyped λ-calculus.

Definition 4.3.9 (Logical Relation)**.**

$$\mathcal{R}_{\mathbb{N}}^{\mathfrak{v},n} = \{(V,v) \mid V = \overline{m} \text{ and } v = \overline{m} \text{ for some } m \in \mathbb{N}\}$$

$$\mathcal{R}_{\mathbb{1}}^{\mathfrak{v},n} = \{(V,v) \mid V = \star \text{ and } v = \star\}$$

$$\mathcal{R}_{\tau\to\rho}^{\mathfrak{v},n} = \{(V,v) \mid \forall p < n.\ \forall (V_1,v_1) \in \mathcal{R}_\tau^{\mathfrak{v},p}.\ (V\ V_1,\ \lambda k{:}\neg\rho^*.v\ (v_1,k)) \in \mathcal{R}_\rho^{\mathfrak{c},p}\}$$

$$\mathcal{R}_{\tau,\rho}^{\mathfrak{s},n} = \{(S,k) \mid p \le n.\ \forall (V,v) \in \mathcal{R}_\tau^{\mathfrak{v},p}.\ (S, \mathbf{return}\ V)\ \mathcal{S}_{\tau,\rho}^p\ (k\ v)\}$$

$$\mathcal{R}_\tau^{\mathfrak{c},n} = \{(M,v) \mid \forall p \le n.\ \forall (S,k) \in \mathcal{R}_{\tau,\rho}^{\mathfrak{s},p}.\ (S,M)\ \mathcal{S}_{\tau,\rho}^p\ (v\ k)\}.$$

These relations satisfy the definition of a step-indexed relation at each type, $\mathcal{R}_\tau^{n+1} \subseteq \mathcal{R}_\tau^n$, because, as the index of $X\ \mathcal{R}_\tau^n\ x$ increases, the pair (X,x) needs to satisfy more conditions.

The logical relation is defined for closed values, stacks and computations and makes use of the step-indexed similarity relation $\mathcal{S}^n$. As expected, related functions map related values to related computations. A stack and a continuation are related if, given any related values, they have a similar reduction behaviour. Finally, computations are related if they have similar reduction behaviour in all related stack-continuation pairs.

We can extend the logical relation to open terms and stacks by substituting in related values. The definition below generalises over all indices:

$$\overrightarrow{x_i:\tau_i} \vdash V\ \mathcal{R}_\rho^{\mathfrak{v}}\ v \iff \overrightarrow{x_i:\tau_i} \vdash V : \rho \text{ and } (\overrightarrow{x_i:\tau_i})^* \vdash v : \rho^* \text{ and}$$
$$(\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i,v_i):\tau_i}.\ (\overrightarrow{(V_i,v_i) \in \mathcal{R}_{\tau_i}^{\mathfrak{v},n}} \implies (V[\overrightarrow{V_i/x_i}], v[\overrightarrow{v_i/x_i}]) \in \mathcal{R}_\rho^{\mathfrak{v},n}))$$

$$\overrightarrow{x_i:\tau_i} \vdash M\ \mathcal{R}_\rho^{\mathfrak{c}}\ v \iff \overrightarrow{x_i:\tau_i} \vdash M : \rho \text{ and } (\overrightarrow{x_i:\tau_i})^* \vdash v : \neg\neg\rho^* \text{ and}$$
$$(\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i,v_i):\tau_i}.\ (\overrightarrow{(V_i,v_i) \in \mathcal{R}_{\tau_i}^{\mathfrak{v},n}} \implies (M[\overrightarrow{V_i/x_i}], v[\overrightarrow{v_i/x_i}]) \in \mathcal{R}_\rho^{\mathfrak{c},n}))$$

$$\overrightarrow{x_i:\tau_i} \vdash S\ \mathcal{R}_{\rho,\rho'}^{\mathfrak{s}}\ k \iff \overrightarrow{x_i:\tau_i} \vdash S : \rho \Rightarrow \rho' \text{ and } (\overrightarrow{x_i:\tau_i})^* \vdash k : \neg\rho^* \text{ and}$$
$$(\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i,v_i):\tau_i}.\ (\overrightarrow{(V_i,v_i) \in \mathcal{R}_{\tau_i}^{\mathfrak{v},n}} \implies (S[\overrightarrow{V_i/x_i}], k[\overrightarrow{v_i/x_i}]) \in \mathcal{R}_{\rho,\rho'}^{\mathfrak{s},n})).$$

An important property of the logical relation is reflexivity. Since $\mathcal{R}$ relates EPCF terms to ECPS terms the property is formulated as below:

Lemma 4.3.10 (Fundamental property of the logical relation)**.** *For any value* $\overrightarrow{x_i : \tau_i} \vdash V : \rho$*, any computation* $\overrightarrow{x_i : \tau_i} \vdash M : \rho$ *and any stack* $\overrightarrow{x_i : \tau_i} \vdash S : \rho \Rightarrow \rho'$ *in EPCF:*

1. $\overrightarrow{x_i : \tau_i} \vdash V \; \mathcal{R}^{\mathfrak{v}}_{\rho} \; V^*$.
2. $\overrightarrow{x_i : \tau_i} \vdash M \; \mathcal{R}^{\mathfrak{c}}_{\rho} \; M^*$.
3. $\overrightarrow{x_i : \tau_i} \vdash S \; \mathcal{R}^{\mathfrak{s}}_{\rho,\rho'} \; S^*$.

Proof. The proof is by induction on the typing derivations of V, M and S. The most interesting case is $M = \mathbf{fix}\, W$. It uses the indices of the logical relation to reason about the number of times the fixed point of W is unfolded. All the cases appear in Appendix A. □

The similarity relation $\mathcal{S}^n$ expresses the fact that computation t simulates configuration (S, M) for n steps. Analogously, we can define a notion of similarity $\mathcal{S}'^n$ which says that (S, M) simulates t for n steps. This will be a family of relations parametrised by $n \in \mathbb{N}$, and EPCF types τ and ρ:

$$\mathcal{S}'^n_{\tau,\rho} \subseteq (Stack(\tau, \rho) \times Comp(\tau)) \times (\vdash).$$

Definition 4.3.11. *The step-indexed similarity relation* $\mathcal{S}'$ *is the family of greatest relations such that* $(S, M) \; \mathcal{S}'^n_{\tau,\rho} \; t$ *implies:*

1. $\forall p < n.\ (t \longrightarrow^p \sigma(v, x.t') \implies (S, M) \rightarrowtail^* (S', \sigma(V; W))$ *where* $V = \overline{m}$ *and* $v = \overline{m}$ *and* $\forall l \in \mathbb{N}.\ (S', W\ \bar{l}) \; \mathcal{S}'^{n-p}_{\tau',\rho} \; t'[\bar{l}/x])$.
2. $\forall p \leq n.\ (t \longrightarrow^p \downarrow \implies (S, M) \rightarrowtail^* (id, \mathbf{return}\ V)$ *for some* $V : \rho)$.

Using $\mathcal{S}'^n$, we can define a family of relations $\mathcal{S}'$ indexed by types τ and ρ, such that $(S, M) \; \mathcal{S}' \; t$ says that (S, M) simulates t for any number of steps. The definition is similar to that of $\mathcal{S}$ (Definition 4.3.7). The analogous of Lemma 4.3.8 can be proved for $\mathcal{S}'^n$ and $\mathcal{S}'$.

Finally, we can define a logical relation $\mathcal{R}'$ exactly as $\mathcal{R}$ was defined but using $\mathcal{S}'$ instead of $\mathcal{S}$. The logical relation $\mathcal{R}'$ also has the fundamental property.

The next step is proving that the trees of a configuration and a computation which simulate each other are closely related. The proof is done by coinduction. We first discuss some properties of the sets of closed ECPS computations and EPCF stack-computation pairs.

Consider the functor:

$$T : \mathbf{Set} \Longrightarrow \mathbf{Set} \text{ where } T(X) = \{\downarrow, \bot\} + (\Sigma \times \mathbb{N} \times X^{\mathbb{N}}).$$

Consider the following coalgebra for this functor: $(\mathit{Trees}_\Sigma, c : \mathit{Trees}_\Sigma \longrightarrow T(\mathit{Trees}_\Sigma))$ where

$$c(tr) = \begin{cases} \downarrow & \text{if } tr = \downarrow \\ \bot & \text{if } tr = \bot \\ (\sigma, n, \overrightarrow{tr'}) & \text{if } tr = \sigma_n(\overrightarrow{tr'}). \end{cases}$$

It is a standard result that the category of coalgebras and coalgebra morphisms for the functor T has a terminal object. This is shown for example by Jacobs [Jac16, Theorem 2.3.9]. Moreover, the terminal object is $(\mathit{Trees}_\Sigma, c)$. The proof of this is very similar to the proof of Proposition 2.3.5 from [Jac16].

Lemma 4.3.12. *The function $[\![-]\!] : (\vdash) \longrightarrow \mathit{Trees}_\Sigma$ is a coalgebra morphism in the category of coalgebras for the functor T.*

Proof. First, note that $(\vdash)$ is indeed a coalgebra by considering the following function $a : (\vdash) \longrightarrow T(\vdash)$ on closed ECPS computations:

$$a(t) = \begin{cases} \downarrow & \text{if } t \longrightarrow^* \downarrow \\ \bot & \text{if } t \longrightarrow^\infty \\ (\sigma, k, \overrightarrow{t'[\overline{n}/x]}) & \text{if } t \longrightarrow^* \sigma(\overline{k}, x.t'). \end{cases}$$

By definition of $\longrightarrow$ exactly one of the cases above will occur, so a is a well-defined function.

To prove that $[\![-]\!]$ is a coalgebra morphism, we use standard domain theoretic techniques. All the necessary results can be found for example in [Fio17]. The proof involves making the order $\leq$ on Trees_Σ more precise, and defining an order $\leq_T$ on $T(\mathit{Trees}_\Sigma)$ which makes it an ω-CPO. The full development can be found in Appendix A. □

Define the following function indexed by EPCF types τ and ρ:

$$|-,-|^*_{(-)} : \mathit{Stack}(\tau,\rho) \times \mathit{Comp}(\tau) \times \mathbb{N} \longrightarrow \mathit{Trees}_\Sigma.$$

similarly to how $|-,-|_{(-)}$ was defined. Assuming all EPCF effect operations have been replaced with operations of arity $\mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha$, the definition is:

$$|S, M|^*_0 = \bot$$

$$|S, M|^*_{n+1} = \begin{cases} \downarrow & \text{if } S = id \text{ and } M = \mathbf{return}\ V \\ |S', M'|^*_n & \text{if } (S, M) \rightarrowtail (S', M') \\ \sigma_k(|S, V\ \overline{0}|^*_n, |S, V\ \overline{1}|^*_n, \ldots) & \text{if } \sigma : \mathbb{N} \times \alpha^{\mathbb{N}} \to \alpha \text{ and } M = \sigma(\overline{k}; V) \\ \bot & \text{otherwise.} \end{cases}$$

The tree $|S, M|^*_n$ is different from $|S, M|_n$ because all value leaves are replaced by $\downarrow$. We can see that $|S, M|^*_n \leq |S, M|^*_{n+1}$ in Trees_Σ so we can define:

$$|-,-|^* : \mathit{Stack}(\tau,\rho) \times \mathit{Comp}(\tau) \longrightarrow \mathit{Trees}_\Sigma$$

$$|S, M|^* = \bigsqcup_{n \in \mathbb{N}} |S, M|^*_n.$$

Define the set of all well-formed stack-computation pairs as:

$$\mathit{Stack} \times \mathit{Comp} = \{(S, M) \mid S \in \mathit{Stack}(\tau,\rho),\ M \in \mathit{Comp}(\tau) \text{ for some EPCF types } \tau, \rho\}$$

and extend the function $|-,-|^*$ to this set:

$$\beta : \mathit{Stack} \times \mathit{Comp} \longrightarrow \mathit{Trees}_\Sigma$$

$$\beta(S, M) = |S, M|^*.$$

Lemma 4.3.13. *The function $\beta : \mathit{Stack} \times \mathit{Comp} \longrightarrow \mathit{Trees}_\Sigma$ is a coalgebra morphism in the category of coalgebras for the functor T.*

Proof. To see that $Stack \times Comp$ is indeed a coalgebra, consider the following function $(Stack \times Comp,\ b : Stack \times Comp \longrightarrow T(Stack \times Comp))$:

$$b(S,M) = \begin{cases} \downarrow & \text{if } (S,M) \rightarrowtail^* (id, \mathbf{return}\ V) \\ \bot & \text{if } (S,M) \rightarrowtail^\infty \\ (\sigma, k, ((S', V\overline{0}), (S', V\overline{1}), \ldots)) & \text{if } (S,M) \rightarrowtail^* (S', \sigma(\overline{k}; V)). \end{cases}$$

By definition of $\rightarrowtail$ the cases above are exhaustive, and by determinacy only one of them can occur, so b is a well-defined function.

The proof that β is a coalgebra morphism is very similar to the proof for $[\![-]\!]$ so we omit it. It uses the ω-CPO structure of $Trees_\Sigma$ and $T(Trees_\Sigma)$ to show that the coalgebra morphism diagram commutes. □

Proposition 4.3.14. *For any well-typed EPCF configuration (S, M), where $S : \tau \Rightarrow \rho$, and any ECPS computation t:*

$$((S,M),t) \in \mathcal{S}_{\tau,\rho} \cap \mathcal{S'}_{\tau,\rho} \implies [\![t]\!] = |S,M|[\downarrow/l_1, \downarrow/l_2, \ldots].$$

Proof. Apply the coinduction proof principle from Proposition 3.3.2, use Lemmas 4.3.12 and 4.3.13 and the fact that $(Trees_\Sigma, c : Trees_\Sigma \longrightarrow T(Trees_\Sigma))$ is a final coalgebra. The full proof appears in Appendix A. □

Finally, we can prove the correctness of the CPS translation:

Theorem 4.3.1. *For any closed computation $M : \tau$ in EPCF and any stack $S : \tau \Rightarrow \rho$ the following holds:*

$$[\![M^*\ S^*]\!] = |S,M|[\downarrow/l_1, \downarrow/l_2, \ldots].$$

That is, all the value leaves of $|S, M|$ are replaced by $\downarrow$, but the nodes and the $\bot$-leaves stay the same.

Proof. From the fundamental property of the logical relation (Lemma 4.3.10 and its analogue for $\mathcal{R}'$) we know that for any closed configuration (S, M), where $S : \tau \Rightarrow \rho$:

$$\forall n \in \mathbb{N}.\ ((S,M),(M^*\ S^*)) \in \mathcal{S}^n_{\tau,\rho} \cap \mathcal{S'}^n_{\tau,\rho}.$$

Then using Lemma 4.3.8 and its analogue for $\mathcal{S}'$ we can deduce that:

$$((S,M),(M^*\ S^*)) \in \mathcal{S}_{\tau,\rho} \cap \mathcal{S'}_{\tau,\rho}.$$

Applying Proposition 4.3.14 we know that:

$$[\![M^*\ S^*]\!] = |S,M|[\downarrow/l_1, \downarrow/l_2, \ldots].$$

□

4.4 ECPS Is More Expressive than EPCF

From the point of view of the propositions-as-types correspondence, or the Curry-Howard isomorphism, the λ-calculus corresponds to intuitionistic logic [SU06, Chapter 6]. Griffin showed that a language with *control operators* can extend this correspondence to classical logic [Gri90].

One such control operator is `call-cc` (call-with-current-continuation) from the programming language Scheme. Griffin showed that `call-cc` can be assigned the type of Peirce's law, a proposition which is only provable classically:

$$((A \to B) \to A) \to A.$$

It is known that PCF and EPCF do not contain any control operators but we will show that ECPS does. In ECPS we can write a term which has the type of Peirce's law and the reduction behaviour of `call-cc`. In this sense, ECPS is more expressive than EPCF. Below is an informal explanation of `call-cc` and how it arises in ECPS.

One way of adding `call-cc` to PCF [RT99] is to add a new type $Cont(A)$ which stands for a continuation waiting for a term of type A. Such a continuation can be written as $\gamma x{:}A.E[x]$ where E is a PCF evaluation context and $E[x]$ is a PCF term. Then add the following term constructors to PCF:

$$\frac{}{\Gamma \vdash \texttt{callcc} : (Cont(A) \to A) \to A} \qquad \frac{}{\Gamma \vdash \texttt{throw} : Cont(A) \to A \to B}$$

The reduction rules for these new constructors are:

$$E[\texttt{callcc}\ (\lambda x{:}Cont(A).M)] \longrightarrow E[M[x := (\gamma y{:}A.E[y])]] \tag{4.4.1}$$

$$E'[\texttt{throw}\ (\gamma y{:}A.E[y])\ V] \longrightarrow E[V]. \tag{4.4.2}$$

The first rule says that when `callcc` is encountered the current evaluation context E is bound to the continuation $x = \gamma y{:}A.E[y]$. Then evaluation of M proceeds normally. If M never throws, the control flow is not changed, the return value of `callcc` was M. If at some point M invokes `throw` $(\gamma y{:}A.E[y])\ V$, the current evaluation context E' is abandoned. Instead, the context E is restored with value V. This looks as if the return value of `callcc` was V.

The type $Cont(A)$ can be interpreted as $A \to B$ for some type B. Thus `callcc` has type $((A \to B) \to A) \to A$. And `throw` has type $(A \to B) \to A \to B$. When the continuation $(\gamma y{:}A.E[y]) : Cont(A)$ is thrown, the current context E' that is waiting for type B is discarded. Therefore, B can be anything.

In ECPS, the PCF type $A \to B$ is interpreted as $\neg(A, \neg B)$. In other words, the implication $A \to B$ is expressed as $\neg(A \wedge \neg B)$. Peirce's law then becomes:

$$\neg(\ \neg(\neg(\mathbf{A} \wedge \neg\mathbf{B}) \wedge \neg\mathbf{A})\ \wedge\ \neg A).$$

In ECPS, a term of this type is:

$$\texttt{callcc}^* = \lambda(f,k){:}(\neg(\neg(\mathbf{A}, \neg\mathbf{B}),\ \neg\mathbf{A}),\ \neg A).\ f\ (\lambda(y,k'){:}(A, \neg B).k\ y,\ k).$$

A term with the type of `throw`:

$$(A \to B) \to A \to B = \neg(\neg(A \wedge \neg B) \wedge \ \neg\neg(A \wedge \neg B))$$

is

$$\texttt{throw}^* = \lambda(k, l){:}(\neg(A, \neg B),\ \neg\neg(A, \neg B)).(l\ k).$$

To illustrate the reduction behaviour of $\texttt{callcc}^*$ and $\texttt{throw}^*$ consider the following examples where $A = \mathbb{N}$:

$$f_1 = \lambda(x, k){:}(\neg(\mathbb{N}, \neg B),\ \neg\mathbb{N}).\ \texttt{throw}^*\ (x,\ \lambda g{:}\neg(\mathbb{N}, \neg B).g\ (\overline{3},\ \lambda w{:}B.\mathit{loop})).$$

Here x is the analogous of x from $\lambda x{:}\mathit{Cont}(A).M$, equation 4.4.1. The function f_1 throws continuation x with value $V = \overline{3}$, in the context $\lambda w{:}B.\mathit{loop}$ analogous to E', from equation 4.4.2.

Now consider the following computation, where $\lambda z{:}\mathbb{N}.\downarrow$ stands for the context E from equation 4.4.1:

$$\begin{aligned}
&\texttt{callcc}^*\ (f_1,\ \lambda z{:}\mathbb{N}.\downarrow)\\
&\qquad\longrightarrow f_1\ (\lambda(y, k'){:}(\mathbb{N}, \neg B).(\lambda z{:}\mathbb{N}.\downarrow)\ y,\ \lambda z{:}\mathbb{N}.\downarrow)\\
&\qquad\longrightarrow \texttt{throw}^*\ (\lambda(y, k'){:}(\mathbb{N}, \neg B).(\lambda z{:}\mathbb{N}.\downarrow)\ y,\ \lambda g{:}\neg(\mathbb{N}, \neg B).g\ (\overline{3},\ \lambda w{:}B.\mathit{loop}))\\
&\qquad\longrightarrow^2 (\lambda(y, k'){:}(\mathbb{N}, \neg B).(\lambda z{:}\mathbb{N}.\downarrow)\ y)\ (\overline{3},\ \lambda w{:}B.\mathit{loop})\\
&\qquad\longrightarrow (\lambda z{:}\mathbb{N}.\downarrow)\ \overline{3}.
\end{aligned}$$

When $\texttt{callcc}^*$ is called, the current continuation $\lambda z{:}\mathbb{N}.\downarrow$ is saved inside

$$\lambda(y, k'){:}(\mathbb{N}, \neg B).(\lambda z{:}\mathbb{N}.\downarrow)\ y.$$

Then when $\texttt{throw}^*$ occurs, the now current continuation $\lambda w{:}B.\mathit{loop}$ is abandoned and the continuation $\lambda z{:}\mathbb{N}.\downarrow$ is restored with value $\overline{3}$.

As another example, consider a function f_2 which does not throw. It just invokes the continuation k that was passed to it:

$$f_2 = \lambda(x, k){:}(\neg(\mathbb{N}, \neg B),\ \neg\mathbb{N}).\ k\ \overline{3}$$

$$\begin{aligned}
&\texttt{callcc}^*\ (f_2,\ \lambda z{:}\mathbb{N}.\downarrow)\\
&\qquad\longrightarrow f_2\ (\lambda(y, k'){:}(\mathbb{N}, \neg B).(\lambda z{:}\mathbb{N}.\downarrow)\ y,\ \lambda z{:}\mathbb{N}.\downarrow)\\
&\qquad\longrightarrow (\lambda z{:}\mathbb{N}.\downarrow)\ \overline{3}.
\end{aligned}$$

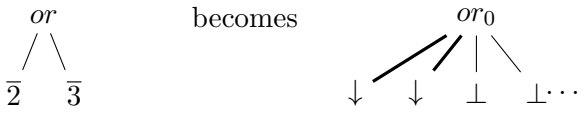
Here, the control flow is not changed by the use of $\texttt{callcc}^*$.

This section is not a full proof that ECPS is more expressive than EPCF. In particualar, we have only shown examples that $\texttt{callcc}^*$ has the desired behaviour. However, this is strong evidence to suggest that an embedding of ECPS in EPCF is not possible. This is why we only studied a translation of EPCF into ECPS in this chapter.

4.5 Chapter Summary

This chapter started the exposition of the novel technical content of the dissertation. In Section 4.1, we presented a continuation-passing translation from EPCF to ECPS. For example, an EPCF function V of type $\tau \to \mathbb{N}$ is translated to an ECPS function $V^* : \neg(\tau^*, \neg\mathtt{nat})$, where $\neg\mathtt{nat}$ is the type of a continuation waiting for the result of V. An EPCF computation $M : \tau$ is translated to a function $M^* = (\lambda k{:}\neg\tau^*. \ldots) : \neg\neg\tau^*$, where k is the continuation waiting for the result of M.

The correctness theorem of this translation (Theorem 4.3.1) implies that: the tree of M^* $(\lambda x{:}\neg\tau^*.\downarrow)$ is obtained from the tree of EPCF computation M by replacing all value leaves with $\downarrow$. For example:



This means that the reduction behaviour of terms is preserved by the translation.

We could not prove Theorem 4.3.1 directly by induction on EPCF terms. To obtain a stronger induction hypothesis, we defined a *logical relation* (Definition 4.3.9) between EPCF terms and ECPS terms. Most importantly, two functions are related if and only if they send related arguments to related computations. Two computations are related when they simulate each other's behaviour, including effect operations. This is a custom notion of similarity introduced in Definition 4.3.4.

We then proved the fundamental property of the logical relation (Lemma 4.3.10): for any EPCF term T, the pair (T, T^*) is in the relation. Using this, and the coinduction proof principle from the previous chapter (Proposition 3.3.2), we proved the correctness of the translation.

Finally, we argued informally that ECPS is strictly more expressive than EPCF because it contains the control operator `call-cc`. Overall, this chapter showed that ECPS is a reasonable choice of language for studying program equivalence. Moreover, because ECPS contains more program contexts than EPCF, it becomes more likely that contextual equivalence equals applicative bisimilarity, which we will prove in Chapter 7, even though this is false for EPCF.

Chapter 5

Applicative Bisimilarity for ECPS

This chapter starts by defining observations for ECPS for all the running examples of effects. Using them, applicative $\mathfrak{P}$-bisimilarity is defined. Two sufficient conditions for bisimilarity to be compatible are identified: Scott-openness and a novel notion of decomposability. All the example observations are proved decomposable. The final section uses Howe's method to prove bisimilarity is indeed compatible. In the following chapters, applicative $\mathfrak{P}$-bisimilarity is compared with other notions of program equivalence.

5.1 Observations for ECPS

To define applicative bisimulation for ECPS we first fix a set of observations $\mathfrak{P}$, which contains subsets of Trees_Σ. The set $\mathfrak{P}$ depends on the effects that are present in the language. It can be used to define various forms of program equivalence which check whether computation trees are in $P \in \mathfrak{P}$.

Observations $P \in \mathfrak{P}$ play a similar role to modalities from $\mathcal{O}$ and to the observations defined by Johann, Simpson and Voigtländer [JSV10]. For the example effects considered so far, observations are defined as follows:

Example 5.1.1 (Pure functional computation)**.** Define $\mathfrak{P} = \{\Downarrow\}$ where $\Downarrow = \{\downarrow\}$. There are no effect operations so the $\Downarrow$ observation only checks for termination.

Example 5.1.2 (Nondeterminism)**.** Define $\mathfrak{P} = \{\mathit{Trees}_\Sigma, \Diamond, \Box\}$ where:

$$\Diamond = \{tr \in \mathit{Trees}_\Sigma \mid \text{at least one of the paths in } tr \text{ that can occur has a } \downarrow \text{ leaf}\}$$

$$\Box = \{tr \in \mathit{Trees}_\Sigma \mid \text{the paths in } tr \text{ that can occur are all finite and finish with a } \downarrow\}.$$

The intuition is that, if $[\![t]\!] \in \Diamond$, then computation t *may* terminate. Whereas, if $[\![t]\!] \in \Box$, t *must* terminate. Notice that there is no condition on the value with which t terminates because ECPS computations do not have a return value.

As discussed in Section 3.2, every node in a computation tree has infinitely many children so some paths in the tree can never be executed. For example, the *or* operation always chooses between its first two children. The definitions of $\Diamond$ and $\Box$ take this into account.

The set of all trees Trees_Σ is chosen to be an observation for technical reasons that will become clear in the next section. However, the fact that Trees_Σ is an observation will not affect any notion of program equivalence because for all computations t, $[\![t]\!] \in \mathit{Trees}_\Sigma$.

Example 5.1.3 (Probabilistic choice)**.** Define the set of observations as:

$$\mathfrak{P} = \{\mathbf{P}_{>q} \mid q \in \mathbb{Q},\ 0 \leq q < 1\} \cup \{\mathit{Trees}_\Sigma\}.$$

Define $\mathbb{P} : \mathit{Trees}_\Sigma \longrightarrow [0,1]$ to be the least function, by the pointwise order, such that:

$$\mathbb{P}(tr) = \begin{cases} 1 & \text{if } tr = \downarrow \\ \frac{1}{2}\mathbb{P}(tr_0) + \frac{1}{2}\mathbb{P}(tr_1) & \text{if } tr = \mathit{p\text{-}or}(tr_0, tr_1). \end{cases}$$

Given functions $f_1, f_2 : \mathit{Trees}_\Sigma \longrightarrow [0,1]$ the pointwise order is defined as:

$$f_1 \leq f_2 \iff \forall tr \in \mathit{Trees}_\Sigma.\ f_1(tr) \leq f_2(tr).$$

Observations are defined as below:

$$\mathbf{P}_{>q} = \{tr \in \mathit{Trees}_\Sigma \mid \mathbb{P}(tr) > q\}.$$

This means that $tr \in \mathbf{P}_{>q}$ if the probability that tr terminates is greater than q. A *p-or* node chooses between its first two children with probability 0.5, so the probability that tree tr terminates is calculated over these choices. Notice that $\mathbb{P}(\bot) = 0$.

Example 5.1.4 (Global store)**.** Define the set of states as the set of functions from storage locations to natural numbers: $\mathit{State} = \mathbb{L} \longrightarrow \mathbb{N}$. The set $\mathfrak{P}$ is defined as:

$$\mathfrak{P} = \{(s \rightarrowtail r) \mid s, r \in \mathit{State}\} \cup \{\mathit{Trees}_\Sigma\}.$$

Define the execution of a tree starting in a state as the *least* partial function:

$$\mathit{exec} : \mathit{Trees}_\Sigma \times \mathit{State} \longrightarrow \{\downarrow\} \times \mathit{State}$$

which satisfies

$$\mathit{exec}(tr, s) = \begin{cases} (\downarrow, s) & \text{if } tr = \downarrow \\ \mathit{exec}(tr_{s(l)}, s) & \text{if } tr = \mathit{lookup}_{l,n}(tr_0, tr_1, \ldots) \\ & \text{and } \mathit{exec}(tr_{s(l)}, s) \text{ is defined} \\ \mathit{exec}(tr_0, s[l := n]) & \text{if } tr = \mathit{update}_{l,n}(tr_0, tr_1, \ldots) \\ & \text{and } \mathit{exec}(tr_0, s[l := n]) \text{ is defined.} \end{cases}$$

Now define observations as:

$$(s \rightarrowtail r) = \{tr \in \mathit{Trees}_\Sigma \mid \mathit{exec}(tr, s) \text{ is defined and } \mathit{exec}(tr, s) = (\downarrow, r)\}.$$

Notice that $\mathit{exec}(tr, s)$ is defined only when the execution of tr terminates. So $tr \in (s \rightarrowtail r)$ only if the execution of tr started in state s terminates in state r.

Example 5.1.5 (Input/output)**.** An I/O-trace is a finite word w over the alphabet

$$\{?n \mid n \in \mathbb{N}\} \cup \{!n \mid n \in \mathbb{N}\}.$$

Thus, a trace is a sequence of input and output operations, where $?n$ means that the number n was given as input to a *read* operation, and $!n$ means that the number n was output by a *write* operation. We can use them to define the set of observation:

$$\mathfrak{P} = \{\langle w \rangle_{\ldots} \mid w \text{ an I/O-trace}\}$$

where

$$\langle w \rangle_{\ldots} = \{tr \in \mathit{Trees}_\Sigma \mid \text{the execution of } tr \text{ produces I/O-trace } w\}.$$

To specify rigorously when "the execution of a tree produces an I/O trace", we can define a relation between trees and I/O traces, by induction on traces. Denote this relation by $\models$.

$$\begin{aligned} tr \models \langle \epsilon \rangle_{\ldots} &\iff \text{true} \\ tr \models \langle (?n)w \rangle_{\ldots} &\iff tr = \mathit{read}_k(tr_0, tr_1, \ldots) \text{ and } tr_n \models \langle w \rangle_{\ldots} \\ tr \models \langle (!n)w \rangle_{\ldots} &\iff tr = \mathit{write}_n(tr_0, tr_1, \ldots) \text{ and } tr_0 \models \langle w \rangle_{\ldots}. \end{aligned}$$

5.2 Applicative $\mathfrak{P}$-Bisimilarity

Definition 5.2.1 (Applicative $\mathfrak{P}$-simulation)**.** *A collection of relations $\mathcal{R}^{\mathfrak{v}}_A \subseteq (\vdash A) \times (\vdash A)$ for each type A and $\mathcal{R}^{\mathfrak{c}} \subseteq (\vdash) \times (\vdash)$ is an applicative* $\mathfrak{P}$-simulation *if:*

1. $v \; \mathcal{R}^{\mathfrak{v}}_{\mathtt{unit}} \; w \implies v = w = \star$.
2. $v \; \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}} \; w \implies v = w$.
3. $s \; \mathcal{R}^{\mathfrak{c}} \; t \implies \forall P \in \mathfrak{P}.\ (\llbracket s \rrbracket \in P \implies \llbracket t \rrbracket \in P)$.
4. $v \; \mathcal{R}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)} \; u \implies \forall \vdash w_1 : A_1, \ldots, \vdash w_n : A_n.\ v(w_1, \ldots, w_n) \; \mathcal{R}^{\mathfrak{c}} \; u(w_1, \ldots, w_n)$.

Applicative $\mathfrak{P}$-similarity $\precsim$ *is the union of all applicative* $\mathfrak{P}$*-simulations. Therefore, it is the greatest applicative* $\mathfrak{P}$*-simulation.*

According to the definition above, unit values and natural number values are similar if and only if they are equal. The third clause says that t simulates s if the computation tree of t has all the properties of the computation tree of s. The properties are specified using the set of observations $\mathfrak{P}$. Notice that simulation for computations is not defined using simulations for values, since computations do not have a return value. The last clause compares the behaviour of functions for all possible arguments.

Definition 5.2.2 (Applicative $\mathfrak{P}$-bisimulation)**.** *An applicative* $\mathfrak{P}$-bisimulation *is a symmetric* $\mathfrak{P}$*-simulation. Applicative* $\mathfrak{P}$-bisimilarity $\sim$ *is the union of all applicative* $\mathfrak{P}$*-bisimulations. Therefore it is the greatest applicative* $\mathfrak{P}$*-bisimulation.*

Below are two properties of bisimilarity and similarity which will be used later. They are followed by two examples of how bisimilarity can be established.

Proposition 5.2.3. *Applicative* $\mathfrak{P}$*-bisimilarity coincides with the intersection between applicative* $\mathfrak{P}$*-similarity and its converse:*

$$(\sim) = (\precsim) \cap (\precsim)^{op}.$$

Proof. The proof is done using the definitions of similarity and bisimilarity. The $\supseteq$ inclusion is shown by proving $(\precsim) \cap (\precsim)^{op}$ is a symmetric simulation. The full proof appears in Appendix B. □

Lemma 5.2.4. *Similarity is preserved by the reduction relation, that is:*

$$\forall s, t.\ \vdash s \precsim^{\mathfrak{c}} t \text{ and } s \longrightarrow^* s' \text{ and } t \longrightarrow^* t' \implies \vdash s' \precsim^{\mathfrak{c}} t'.$$

Proof. By the definition of $[\![-]\!]$ we know that $[\![s]\!] = [\![s']\!]$ and $[\![t]\!] = [\![t']\!]$. So because $\vdash s \precsim^{\mathfrak{c}} t$ we know that:

$$\forall P \in \mathfrak{P}.\ ([\![s]\!] = [\![s']\!] \in P \implies [\![t]\!] = [\![t']\!] \in P).$$

This is enough to establish that $\vdash s' \precsim^{\mathfrak{c}} t'$. □

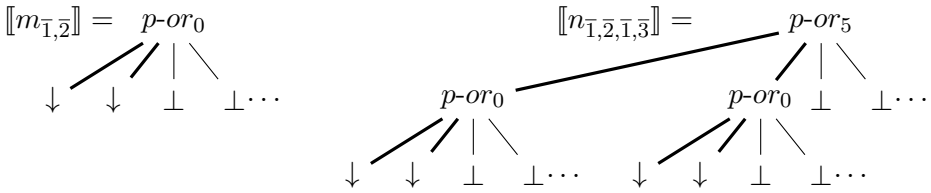
Example 5.2.5 (Probabilistic choice)**.** Consider the following computations:

$$m_{\overline{1},\overline{2}} = \textit{p-or}(\overline{0},\ y.\texttt{case}\ y\ \texttt{in}\ \{\texttt{zero} \Rightarrow (\lambda x{:}\texttt{nat}.\downarrow)\ \overline{1},\ \texttt{succ}(y') \Rightarrow \texttt{case}\ y'\ \texttt{in}\ \{\texttt{zero} \Rightarrow (\lambda x{:}\texttt{nat}.\downarrow)\ \overline{2},\ \texttt{succ}(y'') \Rightarrow \textit{loop}\}\}).$$

Computation $m_{\overline{1},\overline{3}}$ is defined analogously to $m_{\overline{1},\overline{2}}$ where the value $\overline{2}$ inside the computation is replaced by the value $\overline{3}$.

$$n_{\overline{1},\overline{2},\overline{1},\overline{3}} = \textit{p-or}(\overline{5},\ y.\texttt{case}\ y\ \texttt{in}\ \{\texttt{zero} \Rightarrow m_{\overline{1},\overline{2}},\ \texttt{succ}(y') \Rightarrow \texttt{case}\ y'\ \texttt{in}\ \{\texttt{zero} \Rightarrow m_{\overline{1},\overline{3}},\ \texttt{succ}(y'') \Rightarrow \textit{loop}\}\}).$$

Their computation trees are:

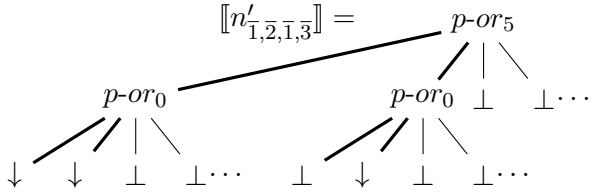


Both computations terminate with a $\downarrow$ with probability 1 so

$$\forall P \in \mathfrak{P}.\ [\![m_{\overline{1},\overline{2}}]\!] \in P \iff [\![n_{\overline{1},\overline{2},\overline{1},\overline{3}}]\!] \in P$$

is true. Therefore $m_{\overline{1},\overline{2}}$ and $n_{\overline{1},\overline{2},\overline{1},\overline{3}}$ are bisimilar. Notice that the subscripts 0 and 5 on the *p-or* nodes do not play any role in establishing bisimilarity.

However, if we consider computation $n'_{\overline{1},\overline{2},\overline{1},\overline{3}}$ with tree:



it has a probability of 3/4 of terminating. Therefore, $[\![m_{\overline{1},\overline{2}}]\!] \in \mathbf{P}_{>0.9}$ but $[\![n'_{\overline{1},\overline{2},\overline{1},\overline{3}}]\!] \notin \mathbf{P}_{>0.9}$ so the two computations are not bisimilar.

Example 5.2.6 (Global store)**.** Plotkin and Power [PP02] axiomatise the behaviour of the global store operations *lookup* and *update* using a set of program equations. We can show that these equations are in fact induced by applicative $\mathfrak{P}$-bisimilarity. For example, consider the following equation:

$$\forall loc, loc' \in \mathbb{L} \text{ where } loc \neq loc'.\ \forall n, n' \in \mathbb{N}.\ \forall (y : \mathtt{nat}, y' : \mathtt{nat} \vdash t).$$
$$l = update_{loc}(\overline{n},\ y.update_{loc'}(\overline{n'},\ y'.t)) \sim^{\mathfrak{c}} update_{loc'}(\overline{n'},\ y'.update_{loc}(\overline{n},\ y.t)) = r.$$

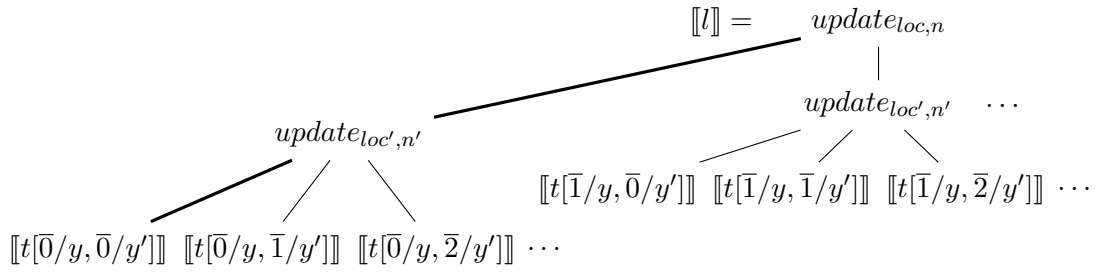
It says that writes to two different locations can be interchanged. Because we are dealing with computations, to prove bisimilarity it suffices to show:

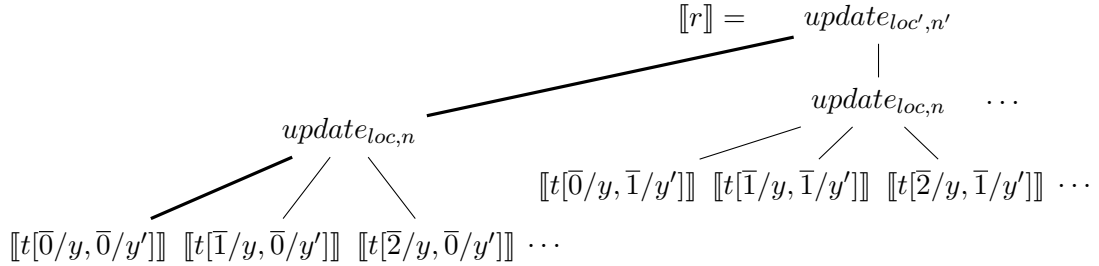
$$\forall s_1, s_2 \in State.\ [\![l]\!] \in (s_1 \rightarrowtail s_2) \iff [\![r]\!] \in (s_1 \rightarrowtail s_2).$$

The observation $(s_1 \rightarrowtail s_2)$ was defined using the partial function *exec* which formalises the execution of a computation tree. So in fact, we need to show that:

$$\forall s_1, s_2 \in State.\ exec([\![l]\!], s_1) \text{ is defined and } exec([\![l]\!], s_1) = (\downarrow, s_2)$$
$$\iff exec([\![r]\!], s_1) \text{ is defined and } exec([\![r]\!], s_1) = (\downarrow, s_2). \quad (5.2.1)$$

The computation trees of l and r respectively are:





Using the trees and the definition of *exec*, we can deduce the following chain of equations,

which is enough to prove equation 5.2.1:

$$\begin{aligned}
exec([\![l]\!],\ s_1) &= exec(update_{loc',n'}(\overrightarrow{[\![t[\overline{0}/y, \overline{i}/y]]\!]}),\ s_1[loc := n]) \\
&= exec([\![t[\overline{0}/y, \overline{0}/y']]\!],\ (s_1[loc = n])[loc' = n']) \\
&= exec([\![t[\overline{0}/y, \overline{0}/y']]\!],\ s_1[loc = n, loc' = n']) && \text{(because } loc \neq loc') \\
&= exec([\![t[\overline{0}/y, \overline{0}/y']]\!],\ (s_1[loc' = n'])[loc = n]) \\
&= exec(update_{loc,n}(\overrightarrow{[\![t[\overline{i}/y, \overline{0}/y]]\!]}),\ s_1[loc' = n']) = exec([\![r]\!],\ s_1) = (\downarrow, s_2).
\end{aligned}$$

The other six equations for global store from [PP02] can be proved analogously. Moreover, Plotkin and Power observe that adding more equations to this set of seven leads to inconsistency. Therefore, these are all the equations between computations for the global store effect.

5.3 Applicative $\mathfrak{P}$-Bisimilarity is a Congruence

This section discusses the two main properties required for $\mathfrak{P}$-bisimilarity to be a well-behaved program equivalence: being an equivalence relation and compatibility. The proof of the next lemma appears in Appendix B.

Lemma 5.3.1. *Applicative $\mathfrak{P}$-similarity is a preorder. Applicative $\mathfrak{P}$-bisimilarity is an equivalence relation.*

Applicative similarity is a *well-typed relation* on *closed* ECPS terms. Compatibility says we can substitute related programs for a variable inside related programs. Therefore, we need to talk about bisimilarity of programs with free variables. Bisimilarity can be extended to open terms in a standard way [LGL17a].

Definition 5.3.2 (Open extension)**.** *Given a well-typed relation on closed terms, $\mathcal{R} = (\mathcal{R}^{\mathfrak{v}}_A, \mathcal{R}^{\mathfrak{c}})$, the open extension of $\mathcal{R}$ is $\mathcal{R}^{\circ} = (\mathcal{R}^{\circ,\mathfrak{v}}_A, \mathcal{R}^{\circ,\mathfrak{c}})$ where:*

$$\begin{aligned}
\overrightarrow{x_i : A_i} \vdash v\ \mathcal{R}^{\circ,\mathfrak{v}}_B\ w \quad &\iff \quad \forall \overrightarrow{u_i : A_i}.\ v[\overrightarrow{u_i/x_i}]\ \mathcal{R}^{\mathfrak{v}}_B\ w[\overrightarrow{u_i/x_i}] \\
\overrightarrow{x_i : A_i} \vdash s\ \mathcal{R}^{\circ,\mathfrak{c}}\ t \quad &\iff \quad \forall \overrightarrow{u_i : A_i}.\ s[\overrightarrow{u_i/x_i}]\ \mathcal{R}^{\mathfrak{c}}\ t[\overrightarrow{u_i/x_i}].
\end{aligned}$$

Definition 5.3.3 (Compatibility [LGL17a])**.** *A well-typed open relation $\mathcal{R} = (\mathcal{R}^{\mathfrak{v}}_A, \mathcal{R}^{\mathfrak{c}})$ is compatible if it is closed under the rules in Figure 5.1. We define $\mathcal{R}$ to be a precongruence if it is a compatible preorder, and a congruence if it is a compatible equivalence relation.*

The following lemma identifies some alternative compatibility rules which will be used in later proofs. Its proof can be found in Appendix B.

Lemma 5.3.4. *Consider a well-typed relation $\mathcal{R}$ that is a preorder. The compatibility rules* (COMP6)*,* (COMP7)*,* (COMP8) *and* (COMP10) *from Figure 5.1 are equivalent to the conjunction of their single-premise versions. More explicitly:*

$$\frac{}{\Gamma \vdash x\ \mathcal{R}^{\mathfrak{v}}_{A}\ x}(\textsc{Comp}1) \qquad \frac{}{\Gamma \vdash \star\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{unit}}\ \star}(\textsc{Comp}2)$$

$$\frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash s\ \mathcal{R}^{\mathfrak{c}}\ t}{\Gamma \vdash \lambda\overrightarrow{x}{:}\overrightarrow{A}.s\ \mathcal{R}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ \lambda\overrightarrow{x}{:}\overrightarrow{A}.t}(\textsc{Comp}3)$$

$$\frac{}{\Gamma \vdash \mathtt{zero}\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ \mathtt{zero}}(\textsc{Comp}4) \qquad \frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v'}{\Gamma \vdash \mathtt{succ}(v)\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ \mathtt{succ}(v')}(\textsc{Comp}5)$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ v' \qquad \Gamma \vdash w_1\ \mathcal{R}^{\mathfrak{v}}_{A_1}\ w'_1, \ldots, \Gamma \vdash w_n\ \mathcal{R}^{\mathfrak{v}}_{A_n}\ w'_n}{\Gamma \vdash v(w_1, \ldots, w_n)\ \mathcal{R}^{\mathfrak{c}}\ v'(w'_1, \ldots, w'_n)}(\textsc{Comp}6)$$

$$\frac{\Gamma, x : \neg(\overrightarrow{A}) \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A})}\ v' \qquad \Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i \text{ for each i}}{\Gamma \vdash (\mu x.v)(\overrightarrow{w})\ \mathcal{R}^{\mathfrak{c}}\ (\mu x.v')(\overrightarrow{w'})}(\textsc{Comp}7)$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \sigma(v, x.t)\ \mathcal{R}^{\mathfrak{c}}\ \sigma(v', x.t')}\sigma \in \Sigma\ (\textsc{Comp}8) \qquad \frac{}{\Gamma \vdash\downarrow\ \mathcal{R}^{\mathfrak{c}}\ \downarrow}(\textsc{Comp}9)$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma \vdash s\ \mathcal{R}^{\mathfrak{c}}\ s' \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}\ \mathcal{R}^{\mathfrak{c}}\ \mathtt{case}\ v'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s',\ \mathtt{succ}(x) \Rightarrow t'\}}$$

(Comp10)

Figure 5.1: Compatibility rules.

- *Rule* (COMP6) *is equivalent to the conjunction of the rules:*

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ v' \qquad \Gamma \vdash \overrightarrow{w_j : A_j}}{\Gamma \vdash v(w_1,\ldots,w_n)\ \mathcal{R}^{\mathfrak{c}}\ v'(w_1,\ldots,w_n)}(\textsc{Comp6L})$$

$$\frac{\Gamma \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (\overrightarrow{w_{1,i-1} : A_{1,i-1}}) \qquad \Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i \qquad \Gamma \vdash (\overrightarrow{w_{i+1,n} : A_{i+1,n}})}{\Gamma \vdash v(w_1,\ldots,w_i,\ldots,w_n)\ \mathcal{R}^{\mathfrak{c}}\ v(w_1,\ldots,w'_i,\ldots,w_n)}$$

($\textsc{Comp6R}_i$) *for each* $i = \overline{1,n}$

- *Rule* (COMP7) *is equivalent to the conjunction of the rules:*

$$\frac{\Gamma, x : \neg(\overrightarrow{A_j}) \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A_j})}\ v' \qquad \Gamma \vdash \overrightarrow{w_j : A_j}}{\Gamma \vdash (\mu x.v)(\overrightarrow{w_j})\ \mathcal{R}^{\mathfrak{c}}\ (\mu x.v')(\overrightarrow{w_j})}(\textsc{Comp7L})$$

$$\frac{\Gamma, x : \neg(\overrightarrow{A_j}) \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (\overrightarrow{w_{1,i-1} : A_{1,i-1}}) \qquad \Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i \qquad \Gamma \vdash (\overrightarrow{w_{i+1,n} : A_{i+1,n}})}{\Gamma \vdash (\mu x.v)(w_1,\ldots,w_i,\ldots,w_n)\ \mathcal{R}^{\mathfrak{c}}\ (\mu x.v)(w_1,\ldots,w'_i,\ldots,w_n)}$$

($\textsc{Comp7R}_i$) *for each* $i = \overline{1,n}$

- *Rule* (COMP8) *is equivalent to the conjunction of the rules:*

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma, x : \mathtt{nat} \vdash t}{\Gamma \vdash \sigma(v, x.t)\ \mathcal{R}^{\mathfrak{c}}\ \sigma(v', x.t)}\sigma \in \Sigma\ (\textsc{Comp8L})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \sigma(v, x.t)\ \mathcal{R}^{\mathfrak{c}}\ \sigma(v, x.t')}\sigma \in \Sigma\ (\textsc{Comp8R})$$

- *Rule* (COMP10) *is equivalent to the conjunction of the rules:*

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma \vdash s \qquad \Gamma, x : \mathtt{nat} \vdash t}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}\ \mathcal{R}^{\mathfrak{c}}\ \mathtt{case}\ v'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}}$$

(COMP10V)

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma \vdash s\ \mathcal{R}^{\mathfrak{c}}\ s' \qquad \Gamma, x : \mathtt{nat} \vdash t}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}\ \mathcal{R}^{\mathfrak{c}}\ \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s',\ \mathtt{succ}(x) \Rightarrow t\}}$$

(COMP10L)

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma \vdash s \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}\ \mathcal{R}^{\mathfrak{c}}\ \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t'\}}$$

(COMP10R)

To prove $\mathfrak{P}$-bisimilarity is a congruence, we have identified two sufficient conditions that the set of observations $\mathfrak{P}$ should satisfy. One of them is that every observation needs

to be Scott-open, as in the work of Simpson and Voorneveld [SV18]. The second condition is that $\mathfrak{P}$ needs to be decomposable for a novel definition of decomposability.

Definition 5.3.5 (Scott-openness)**.** *A set of trees X is Scott-open if:*

1. *It is upwards closed, that is: $tr \in X$ and $tr \leq tr'$ imply $tr' \in X$.*
2. *Whenever $tr_1 \leq tr_2 \leq \ldots$ is an ascending chain with least upper bound $\bigsqcup tr_i \in X$, then $tr_j \in X$ for some j.*

Definition 5.3.6 (Decomposability)**.** *The set of observations $\mathfrak{P}$ is decomposable if for any $P \in \mathfrak{P}$ and for any $tr \in P$:*

$$tr = \sigma_n(\overrightarrow{tr'}) \implies \exists \overrightarrow{P'} \in \mathfrak{P}.\ \overrightarrow{tr'} \in \overrightarrow{P'} \text{ and } \forall \overrightarrow{p'} \in \overrightarrow{P'}.\ \sigma_n(\overrightarrow{p'}) \in P.$$

Decomposability says that, whenever a tree tr is part of an observation P, the children of tr's root should themselves be part of some observations which fully capture the restrictions that P places on them. This is true for all examples of effects considered so far. Using the last two definitions we can state the main theorem of this chapter:

Theorem 5.3.7. *Given a decomposable set of Scott-open observations $\mathfrak{P}$:*

1. *The open extension of applicative $\mathfrak{P}$-similarity, $\precsim^\circ$, is compatible, and hence it is a precongruence.*
2. *The open extension of applicative $\mathfrak{P}$-bisimilarity, $\sim^\circ$, is compatible, and hence it is a congruence.*

It is easy to check that all the running examples of observations are upwards closed. The proof that they satisfy the second condition in the definition of Scott-openness is the same as in [SV18]. It remains to check that $\mathfrak{P}$ is decomposable:

Example 5.3.8 (Pure functional computation)**.** The only observation is $\Downarrow = \{\downarrow\}$. There are no trees in $\Downarrow$ whose root has children, so decomposability is satisfied.

Example 5.3.9 (Nondeterminism)**.** Recall that $\mathfrak{P} = \{\mathit{Trees}_\Sigma, \Diamond, \Box\}$. For any $tr \in \mathit{Trees}_\Sigma$ we can choose each $P'_n = \mathit{Trees}_\Sigma$ to fulfil decomposability.

Consider $tr \in \Diamond$. Either $tr = \downarrow$, in which case we are done, or $tr = or_n(tr'_0, tr'_1, tr'_2 \ldots)$. It must be the case that either tr'_0 or tr'_1 have a reachable $\downarrow$-leaf. Without loss of generality, assume tr'_0 has a reachable $\downarrow$-leaf. Then we know $tr'_0 \in \Diamond$ so we can choose $P'_0 = \Diamond, P'_1 = \mathit{Trees}_\Sigma, P'_2 = \mathit{Trees}_\Sigma, \ldots$. For any $\overrightarrow{p'} \in \overrightarrow{P'}$ we know $or_n(\overrightarrow{p'}) \in \Diamond$ because p'_0 has a reachable $\downarrow$-leaf.

The argument for $tr \in \Box$ is analogous, if we choose $P'_0 = \Box, P'_1 = \Box, P'_2 = \mathit{Trees}_\Sigma, P'_3 = \mathit{Trees}_\Sigma, \ldots$. This proof relies on the fact that Trees_Σ is an observation, which is way we decided to include it in $\mathfrak{P}$.

Example 5.3.10 (Probabilistic choice)**.** Consider $tr = p\text{-}or_n(tr'_0, tr'_1, tr'_2, \ldots) \in \mathbf{P}_{>q}$ for some $q \in \mathbb{Q}$, $0 \leq q < 1$. Recall the definition of the partial function $\mathbb{P}$ from Example 5.1.3. We know that:

$$\mathbb{P}(tr) = \frac{1}{2}\mathbb{P}(tr'_0) + \frac{1}{2}\mathbb{P}(tr'_1) > q. \tag{5.3.1}$$

Define:

$$q_0 = \frac{\mathbb{P}(tr'_0)}{\mathbb{P}(tr'_0) + \mathbb{P}(tr'_1)} \cdot 2q$$

$$q_1 = \frac{\mathbb{P}(tr'_1)}{\mathbb{P}(tr'_0) + \mathbb{P}(tr'_1)} \cdot 2q.$$

The two probabilities $\mathbb{P}(tr'_0)$ and $\mathbb{P}(tr'_1)$ are rational numbers because they are defined as a sum of rational numbers. The threshold q is rational by assumption, so q_1 and q_2 are rational.

From equation 5.3.1 we can deduce that:

$$1 \geq \mathbb{P}(tr'_0) > q_0$$

$$1 \geq \mathbb{P}(tr'_1) > q_1.$$

So we can choose $P'_0 = \mathbf{P}_{>q_0}, P'_1 = \mathbf{P}_{>q_1}, P'_2 = \mathit{Trees}_\Sigma, P'_3 = \mathit{Trees}_\Sigma, \ldots$. Therefore, $\overrightarrow{tr'} \in \overrightarrow{P'}$ as required.

Consider some other subtrees $\overrightarrow{p'} \in \overrightarrow{P'}$. By the way we defined $\overrightarrow{P'}$ it follows that:

$$\frac{1}{2}\mathbb{P}(p'_0) + \frac{1}{2}\mathbb{P}(p'_1) > \frac{1}{2}(q_0 + q_1) = q$$

so $p\text{-}or(\overrightarrow{p'}) \in \mathbf{P}_{>q}$ as required.

Example 5.3.11 (Global store)**.** Consider a tree $tr = \sigma_n(tr'_0, tr'_1, tr'_2, \ldots) \in (s \rightarrowtail r)$. It must be the case that $\mathit{exec}(tr, s) = (\downarrow, r)$.

If $\sigma_n = \mathit{lookup}_{l,n}$: because $\mathit{exec}(tr, s)$ is defined it must be the case that $\mathit{exec}(tr'_{s(l)}, s)$ is also defined and $\mathit{exec}(tr'_{s(l)}, s) = \mathit{exec}(tr, s) = (\downarrow, r)$ so we know that $tr'_{s(l)} \in (s \rightarrowtail r)$. In the definition of decomposability, choose $P'_{s(l)} = (s \rightarrowtail r)$ and $P'_{k \neq s(l)} = \mathit{Trees}_\Sigma$ and we are done.

If $\sigma_n = \mathit{update}_{l,n}$: because $\mathit{exec}(tr, s)$ is defined it must be the case that $\mathit{exec}(tr'_0, s[l := n])$ is also defined and $\mathit{exec}(tr'_0, s[l := n]) = \mathit{exec}(tr, s) = (\downarrow, r)$. Therefore $tr'_0 \in (s[l := n] \rightarrowtail r)$. We can choose $P'_0 = (s[l := n] \rightarrowtail r)$ and $P'_{k \neq 0} = \mathit{Trees}_\Sigma$ and we are done.

Example 5.3.12 (Input/output)**.** Consider a tree $tr = \sigma_n(tr'_0, tr'_1, tr'_2, \ldots) \in \langle w \rangle_{\ldots}$. If $w = \epsilon$, then decomposability is immediately satisfied by choosing $P'_k = \langle \epsilon \rangle_{\ldots}$. Assume $w \neq \epsilon$.

If $\sigma_n = \mathit{read}_n$, it must be the case that $w = (?k)w'$ and $tr'_k \models \langle w' \rangle_{\ldots}$. We can choose $P'_k = \langle w' \rangle_{\ldots}$ and $P'_{m \neq k} = \langle \epsilon \rangle_{\ldots}$ and we are done.

If $\sigma_n = \mathit{write}_n$, then $w = (!n)w'$ and $tr'_0 \models \langle w' \rangle_{\ldots}$. Choose $P'_0 = \langle w' \rangle_{\ldots}$ and $P'_{k \neq 0} = \langle \epsilon \rangle_{\ldots}$ and we are done.

5.4 Howe's Method

To prove Theorem 5.3.7 we will use a method originally due to Howe [How96]. The strategy is to define a relation $\precsim^{\mathcal{H}}$ named the Howe extension of $\precsim$, prove that it is compatible, and then prove that it coincides with $\precsim^{\circ}$.

$$\frac{}{\Gamma \vdash x\ \widehat{\mathcal{R}}^{\mathfrak{v}}_{A}\ x}(\text{C1}) \quad \frac{}{\Gamma \vdash \star\ \widehat{\mathcal{R}}^{\mathfrak{v}}_{\mathtt{unit}}\ \star}(\text{C2}) \quad \frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash s\ \mathcal{R}^{\mathfrak{c}}\ t}{\Gamma \vdash \lambda\overrightarrow{x}{:}\overrightarrow{A}.s\ \widehat{\mathcal{R}}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ \lambda\overrightarrow{x}{:}\overrightarrow{A}.t}(\text{C3})$$

$$\frac{}{\Gamma \vdash \mathtt{zero}\ \widehat{\mathcal{R}}^{\mathfrak{v}}_{\mathtt{nat}}\ \mathtt{zero}}(\text{C4}) \quad \frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v'}{\Gamma \vdash \mathtt{succ}(v)\ \widehat{\mathcal{R}}^{\mathfrak{v}}_{\mathtt{nat}}\ \mathtt{succ}(v')}(\text{C5})$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ v' \qquad \Gamma \vdash w_1\ \mathcal{R}^{\mathfrak{v}}_{A_1}\ w'_1, \ldots, \Gamma \vdash w_n\ \mathcal{R}^{\mathfrak{v}}_{A_n}\ w'_n}{\Gamma \vdash v(w_1, \ldots, w_n)\ \widehat{\mathcal{R}}^{\mathfrak{c}}\ v'(w'_1, \ldots, w'_n)}(\text{C6})$$

$$\frac{\Gamma, x : \neg(\overrightarrow{A}) \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A})}\ v' \qquad \Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i \text{ for each i}}{\Gamma \vdash (\mu x.v)(\overrightarrow{w})\ \widehat{\mathcal{R}}^{\mathfrak{c}}\ (\mu x.v')(\overrightarrow{w'})}(\text{C7})$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \sigma(v, x.t)\ \widehat{\mathcal{R}}^{\mathfrak{c}}\ \sigma(v', x.t')}\sigma \in \Sigma\ (\text{C8}) \quad \frac{}{\Gamma \vdash\downarrow\ \widehat{\mathcal{R}}^{\mathfrak{c}}\ \downarrow}(\text{C9})$$

$$\frac{\Gamma \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v' \qquad \Gamma \vdash s\ \mathcal{R}^{\mathfrak{c}}\ s' \qquad \Gamma, x : \mathtt{nat} \vdash t\ \mathcal{R}^{\mathfrak{c}}\ t'}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\}\ \widehat{\mathcal{R}}^{\mathfrak{c}}\ \mathtt{case}\ v'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s',\ \mathtt{succ}(x) \Rightarrow t'\}}$$

(C10)

Figure 5.2: Compatible refinement rules.

The justification that Pitts [Pit11] gives for the use of Howe's method in the case of the untyped λ-calculus applies to ECPS as well. A direct proof that $\precsim^{\circ}$ is compatible is problematic because it requires proving a substitutivity property of $\precsim^{\circ}$ which is very close to compatibility. Howe's method avoids this problem because $\precsim^{\mathcal{H}}$ is compatible by construction, and can be proved substitutive.

The proof of compatibility of applicative $\mathfrak{P}$-bisimilarity follows a similar structure to that for applicative bisimilarity for EPCF, found in [SV17, Appendix] and [SV18, Section 6]. We present all the proofs in detail, filling in gaps in Simpson's and Voorneveld's presentation, and adapting to the setting of ECPS.

Definition 5.4.1 (Compatibe refinement)**.** *Given a well-typed open relation $\mathcal{R}$ its compatible refinement $\widehat{\mathcal{R}}$ is inductively defined by the rules in Figure 5.2.*

Definition 5.4.2 (Howe extension)**.** *Given a well-typed closed relation $\mathcal{R}$, we define its Howe extension $\mathcal{R}^{\mathcal{H}}$ to be the least relation $\mathcal{S}$ such that $\mathcal{S} = \mathcal{R}^{\circ} \circ \widehat{\mathcal{S}}$.*

It has been observed by Levy ([Lev06, Proposition 5.4]) that the equation above determines a unique relation. The Howe extension can equivalently be defined inductively as *the least relation closed under the rules*:

$$\frac{\Gamma \vdash s\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{c}}\ t \qquad \Gamma \vdash t\ \mathcal{R}^{\circ,\mathfrak{c}}\ r}{\Gamma \vdash s\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ r}(\text{HC}) \quad \frac{\Gamma \vdash v\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{v}}_{A}\ w \qquad \Gamma \vdash w\ \mathcal{R}^{\circ,\mathfrak{v}}_{A}\ u}{\Gamma \vdash v\ \mathcal{R}^{\mathcal{H},\mathfrak{v}}_{A}\ u}(\text{HV})$$

This is shown in [LGL17b].

Below are two lemmas about the open extension and the Howe extension of a relation. Their proofs appear in Appendix B.1.

Lemma 5.4.3 (From [SV17, Appendix])**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is reflexive:*

1. *The Howe extension of $\mathcal{R}$, $\mathcal{R}^{\mathcal{H}}$, is compatible and hence reflexive.*
2. $\mathcal{R}^{\circ} \subseteq \mathcal{R}^{\mathcal{H}}$.

Lemma 5.4.4 (From [SV17, Appendix])**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is transitive:*

$$\mathcal{R}^{\circ} \circ \mathcal{R}^{\mathcal{H}} \subseteq \mathcal{R}^{\mathcal{H}}.$$

Lemma 5.4.5 (Substitutivity)**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is transitive, its Howe extension satisfies the following two value-substitutivity properties:*

1. $\overrightarrow{x_i : A_i}, y : B \vdash s \; \mathcal{R}^{\mathcal{H},\mathfrak{c}} \; t$ *and* $\overrightarrow{x_i : A_i} \vdash v \; \mathcal{R}_B^{\mathcal{H},\mathfrak{v}} \; w \implies \overrightarrow{x_i : A_i} \vdash s[v/y] \; \mathcal{R}^{\mathcal{H},\mathfrak{c}} \; t[w/y]$.
2. $\overrightarrow{x_i : A_i}, y : B \vdash u \; \mathcal{R}_C^{\mathcal{H},\mathfrak{v}} \; u'$ *and* $\overrightarrow{x_i : A_i} \vdash v \; \mathcal{R}_B^{\mathcal{H},\mathfrak{v}} \; w \implies \overrightarrow{x_i : A_i} \vdash u[v/y] \; \mathcal{R}_C^{\mathcal{H},\mathfrak{v}} \; u'[w/y]$.

Proof. The proof is done by induction on the structure of s and u. It can be found in Appendix B.1. □

The following lemma will help prove that $\precsim^{\mathcal{H}}$ restricted to closed terms is a simulation. Its proof appears in Appendix B.1.

Lemma 5.4.6. *Consider a well-typed closed relation $\leq$ that is a $\mathfrak{P}$-simulation. For any closed values v and w:*

$$\vdash v \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w \implies v = w.$$

Using the domain theoretic definition of ECPS computation trees (Definition 3.2.2) we can state the Key Lemma which will help us prove $\precsim^{\mathcal{H}}$ is a simulation. Recall that $[\![s]\!]_n$ is the tree resulting from n steps of evaluation of s.

Lemma 5.4.7 (Key Lemma)**.** *Consider a decomposable set of Scott-open observations $\mathfrak{P}$. Consider a well-typed closed relation $\leq$ that is a preorder and a $\mathfrak{P}$-simulation. For any closed computations s and t, $\vdash s \leq^{\mathcal{H},\mathfrak{c}} t$ implies:*

$$\forall n \in \mathbb{N}.\ \forall P \in \mathfrak{P}.\ [\![s]\!]_n \in P \implies [\![t]\!] \in P.$$

Proof. The proof is done by induction on $n \in \mathbb{N}$. It uses the fact that observations are upwards closed and $\mathfrak{P}$ is decomposable. The full proof appears in Appendix B.1. □

Proposition 5.4.8. *Consider a decomposable set of Scott-open observations $\mathfrak{P}$. Consider a well-typed closed relation $\leq$ that is a preorder and a $\mathfrak{P}$-simulation. The Howe extension of $\leq$, $\leq^{\mathcal{H}}$, restricted to closed terms is an applicative simulation.*

Proof. We need to verify that all four conditions in the definition of applicative simulation are satisfied by $\leq^{\mathcal{H}}$ for closed terms.

1. $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{unit}} w \implies v = w = \star$. Assume $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{unit}} w$. The only closed value of type `unit` is $\star$ so $v = w = \star$.

2. $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{nat}} w \implies v = w$. This is Lemma 5.4.6.

3. $\vdash s \leq^{\mathcal{H},\mathfrak{c}} t \implies \forall P \in \mathfrak{P}.\ (\llbracket s \rrbracket \in P \implies \llbracket t \rrbracket \in P)$. Assume that $\vdash s \leq^{\mathcal{H},\mathfrak{c}} t$ and $\llbracket s \rrbracket \in P$ for some $P \in \mathfrak{P}$. From the Key Lemma (Lemma 5.4.7) we know that:

$$\forall n \in \mathbb{N}.\ \forall P \in \mathfrak{P}.\ \llbracket s \rrbracket_n \in P \implies \llbracket t \rrbracket \in P. \tag{5.4.1}$$

   We know $\llbracket s \rrbracket = \bigsqcup_{m \in \mathbb{N}} \llbracket s \rrbracket_m \in P$ and that $\{\llbracket s \rrbracket_m\}_{m \in \mathbb{N}}$ is an ascending chain. By Scott-openness of P, there exists $j \in \mathbb{N}$ such that $\llbracket s \rrbracket_j \in P$.

   Therefore, by equation 5.4.1 we have the desired result: $\llbracket t \rrbracket \in P$.

4. $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\neg(A_1,\ldots,A_n)} w \implies \forall \vdash u_1 : A_1, \ldots, \vdash u_n : A_n.\ v(u_1, \ldots, u_n) \leq^{\mathcal{H},\mathfrak{c}} w(u_1, \ldots, u_n)$. Assume $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\neg(A_1,\ldots,A_n)} w$. By Lemma 5.4.3 we know $\leq^{\mathcal{H}}$ is compatible and reflexive. Therefore, for all i, $u_i \leq^{\mathcal{H},\mathfrak{v}}_{A_i} u_i$ so by compatibility $v(u_1, \ldots, u_n) \leq^{\mathcal{H},\mathfrak{c}} w(u_1, \ldots, u_n)$.

□

The following lemmas will be used to prove that similarity and bisimilarity are compatible. Their proofs can be found in Appendix B.1.

Lemma 5.4.9. *Given a well-typed open relation $\mathcal{R}$ that is reflexive and has the two substitutivity properties from Lemma 5.4.5, and a well-typed closed relation $\mathcal{S}$ then:*

if $\mathcal{R}$ restricted to closed terms is included in $\mathcal{S}$ then $\mathcal{R} \subseteq \mathcal{S}^\circ$.

Lemma 5.4.10. *Given a $\mathfrak{P}$-simulation $\mathcal{R}$, its reflexive-transitive closure, $\mathcal{R}^*$ is also a $\mathfrak{P}$-simulation.*

Lemma 5.4.11. *Given a well-typed compatible relation $\mathcal{R}$, its reflexive-transitive closure $\mathcal{R}^*$ is also compatible.*

Lemma 5.4.12 (From [Las98])**.** *Given a well-typed closed relation $\mathcal{R}$ the following holds:*

if $\mathcal{R}^\circ$ is reflexive and symmetric, then $\mathcal{R}^{\mathcal{H}*}$ is symmetric.

Where S^* denotes the reflexive-transitive closure of a relation $\mathcal{S}$.

Finally, we can prove similarity and bisimilarity are compatible. We first recall the formal statement of this:

Theorem 5.3.7. *Given a decomposable set of Scott-open observations $\mathfrak{P}$:*

1. *The open extension of applicative $\mathfrak{P}$-similarity, $\precsim^\circ$, is compatible, and hence it is a precongruence.*

2. *The open extension of applicative $\mathfrak{P}$-bisimilarity, $\sim^\circ$, is compatible, and hence it is a congruence.*

Proof. This proof has the same structure as the proof of Theorem 3 from [SV18]. Here, we present significant details that were missing.

1. We need to prove that the open extension of applicative $\mathfrak{P}$-similarity, $\precsim^\circ$, is compatible. We know that $\precsim$ is a preorder (Lemma 5.3.1) and a simulation. Therefore, we can apply Proposition 5.4.8 to deduce that the restriction of $\precsim^{\mathcal{H}}$ to closed terms is a simulation, so it is included in the greatest simulation, $\precsim$.

   Since $\precsim$ is transitive we know from Lemma 5.4.5 that $\precsim^{\mathcal{H}}$ has the two substitution properties. Because $\precsim$ is reflexive, we know from Lemma 5.4.3 that $\precsim^{\mathcal{H}}$ is reflexive.

   We can use all these to apply Lemma 5.4.9 for $\precsim^{\mathcal{H}}$ and $\precsim$ to deduce $\precsim^{\mathcal{H}} \subseteq \precsim^\circ$.

   From Lemma 5.4.3 we already know that $\precsim^\circ \subseteq \precsim^{\mathcal{H}}$ and that $\precsim^{\mathcal{H}}$ is compatible. Therefore, the open extension of applicative $\mathfrak{P}$-similarity equals the Howe extension, $\precsim^\circ = \precsim^{\mathcal{H}}$, and is compatible.

   Since $\precsim$ is a preorder, $\precsim^\circ$ is also a preorder, so it is a precongruence.

2. We need to prove that $\sim^\circ$ is compatible. The relation $\sim$ is also a $\mathfrak{P}$-simulation. From Lemma 5.3.1 we know $\sim$ is an equivalence relation, hence a preorder. Therefore we can use Proposition 5.4.8 to deduce that $\sim^{\mathcal{H}}$ restricted to closed terms is a simulation.

   From Lemma 5.4.10, we obtain that $\sim^{\mathcal{H}*}$ restricted to closed terms is a simulation, because restricting to closed terms and taking the reflexive-transitive closure are commutative operations.

   Because $\sim$ is reflexive and symmetric it follows that $\sim^\circ$ is also reflexive and symmetric. We can then apply Lemma 5.4.12 for $\sim$ to deduce that $\sim^{\mathcal{H}*}$ is symmetric. Therefore, $\sim^{\mathcal{H}*}$ restricted to closed terms is also symmetric.

   As a result, we know that $\sim^{\mathcal{H}*}$ restricted to closed terms is a bisimulation, so it is included in the greatest bisimulation, $\sim$.

   Now we would like to use Lemma 5.4.9 to deduce $\sim^{\mathcal{H}*} \subseteq \sim^\circ$. We know $\sim$ is transitive so we can apply Lemma 5.4.5 to deduce that $\sim^{\mathcal{H}}$ has the required substitutivity properties. Then it is easy to prove using transitivity and reflexivity that $\sim^{\mathcal{H}*}$ also has the required substitutivity properties. Moreover, $\sim^{\mathcal{H}*}$ is reflexive by definition. So we can apply Lemma 5.4.9.

   Because $\sim$ is reflexive we know from Lemma 5.4.3 that $\sim^\circ \subseteq \sim^{\mathcal{H}}$ and that $\sim^{\mathcal{H}}$ is compatible. By definition of the reflexive transitive closure we know $\sim^{\mathcal{H}} \subseteq \sim^{\mathcal{H}*}$.

   Therefore we know $\sim^{\mathcal{H}*} \subseteq \sim^\circ$ and $\sim^\circ \subseteq \sim^{\mathcal{H}} \subseteq \sim^{\mathcal{H}*}$. So $\sim^\circ = \sim^{\mathcal{H}*}$.

   Because $\sim^{\mathcal{H}}$ is compatible, from Lemma 5.4.11, $\sim^{\mathcal{H}*}$ is also compatible. Therefore, $\sim^\circ$ is compatible. Since $\sim$ is an equivalence relation, $\sim^\circ$ is also an equivalence relation, so it is a congruence.

□

5.5 Chapter Summary

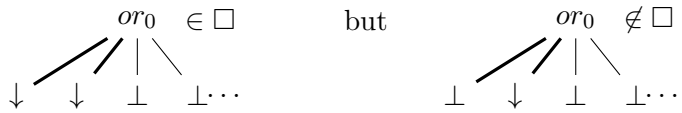
This chapter started by defining a set $\mathfrak{P}$ of *observations*, for each of the following effects: nondeterminisim, probabilistic choice, global store and I/O. In ECPS, we can observe termination and some effect operations, both of which are encoded by computation trees.

Therefore, each observation is a set of trees. For example, for nondeterminism the observations are:

$$\Diamond = \{\text{trees in which at least one execution path that can occur ends in } \downarrow\}$$

$$\Box = \{\text{trees in which all paths that can occur have finite height and end in } \downarrow\}.$$

We used observations to define applicative bisimilarity for ECPS (Definition 5.2.2). Two values of ground type are bisimilar when they are equal. Two functions are bisimilar if, for all arguments, they yield bisimilar computations. Finally, two computations are bisimilar if they belong to exactly the same elements of $\mathfrak{P}$, in other words, they have the same observable behaviour. For example, two computations with the trees below are not bisimilar:



Next, we defined compatibility formally (Definition 5.3.3). The definition says that two related programs can be substituted in related contexts to yield another pair of related programs. We identified two sufficient conditions that the set of observations $\mathfrak{P}$ should satisfy in order for bisimilarity to be compatible. These are Scott-openness, which has to do with the topology of the elements of $\mathfrak{P}$, and *decomposability* (Definition 5.3.6). This definition of decomposability is novel. It states that, for each tree tr in an observation $P \in \mathfrak{P}$, there exist observations that characterise the restrictions that P places on the children of tr.

The main result of this chapter is Theorem 5.3.7, which states that, given a decomposable set $\mathfrak{P}$ of Scott-open observations, bisimilarity is compatible. The proof of this uses Howe's method (Section 5.4) and required significant effort. The idea is to define a new relation named the Howe extension of bisimilarity, which is compatible by construction, and prove it equal to bisimilarity.

Chapter 6

Logical Equivalence for ECPS

This chapter introduces the logic $\mathcal{F}$ whose formulas express properties of ECPS terms. The logic is defined using the set of observations $\mathfrak{P}$. We prove that program equivalence induced by $\mathcal{F}$ is an equivalence relation and coincides with applicative $\mathfrak{P}$-bisimilarity. Therefore, using the main theorem of Chapter 5, we can deduce that $\mathcal{F}$-logical equivalence is compatible. The results in this chapter will be used in Chapter 7 to show that the logic $\mathcal{F}$ characterises contextual equivalence, which is our main goal.

6.1 Two Logics for ECPS

Recall the set of observations $\mathfrak{P}$, defined in Section 5.1, which contains sets of ECPS effect trees. Each $P \in \mathfrak{P}$ specifies the shape of computation trees for a particular effect. Using $\mathfrak{P}$, we define two slightly different logics for ECPS named $\mathcal{V}$ and $\mathcal{F}$ respectively.

In the logic $\mathcal{V}$ values appear inside logical formulas, whereas this is not the case in $\mathcal{F}$. Both logics make a distinction between value formulas and computation formulas. Value formulas are always associated an ECPS type.

Definition 6.1.1 (Logic $\mathcal{F}$)**.** *The value formulas of the logic $\mathcal{F}$ are constructed from basic formulas $\phi = \{n\}$ and $(\phi_1, \ldots, \phi_n) \mapsto P$, where P is an observation from $\mathfrak{P}$, according to the rules in Figure 6.1. In these rules, A stands for an ECPS type. The computation formulas are the elements of $\mathfrak{P}$.*

The satisfaction relation $\models$ relates a closed value $\vdash v : A$ to a value formula $\phi : A$ of the same type, or a closed computation t to an observation P. The definition of $\models$ appears in Figure 6.2. Intuitively, $v \models \phi$ means that the program v has property ϕ.

Let $\mathcal{F}^+$ be the fragment of $\mathcal{F}$ without negation.

Definition 6.1.2 (Logic $\mathcal{V}$)**.** *The logic $\mathcal{V}$ is the same as $\mathcal{F}$ except that the* (VAL) *rule is replaced by:*

$$\frac{\vdash w_1 : A_1 \ldots \vdash w_n : A_n}{(w_1, \ldots, w_n) \mapsto P : \neg(A_1, \ldots, A_n)} P \in \mathfrak{P} \ \text{(VAL')}$$

$$v \models (w_1, \ldots, w_n) \mapsto P \quad \Longleftrightarrow \quad [\![v(w_1 \ldots w_n)]\!] \in P.$$

That is, formulas of function type are now constructed using ECPS values.

Let $\mathcal{V}^+$ be the fragment of $\mathcal{V}$ without negation.

$$\frac{n \in \mathbb{N}}{\{n\} : \mathtt{nat}}(\text{NAT}) \quad \frac{\phi_1 : A_1 \ldots \phi_n : A_n}{(\phi_1, \ldots, \phi_n) \mapsto P : \neg(A_1, \ldots, A_n)} P \in \mathfrak{P}\ (\text{VAL})$$

$$\frac{(\phi_i : A)_{i \in I}}{\vee_{i \in I} \phi_i : A}(\text{DISJ}) \quad \frac{(\phi_i : A)_{i \in I}}{\wedge_{i \in I} \phi_i : A}(\text{CONJ}) \quad \frac{\phi : A}{\neg\phi : A}(\text{NEG})$$

Figure 6.1: Value formulas in the logic $\mathcal{F}$.

$$\begin{array}{rcl}
v \models \{n\} & \iff & v = \overline{n} \\
v \models (\phi_1, \ldots, \phi_n) \mapsto P & \iff & \text{for all closed values } w_1, \ldots, w_n \text{ such that } w_i \models \phi_i \\
 & & \text{then } [\![v(w_1 \ldots w_n)]\!] \in P \\
v \models \vee_{i \in I} \phi_i & \iff & \text{there exists } j \in I \text{ such that } v \models \phi_j \\
v \models \wedge_{i \in I} \phi_i & \iff & \text{for all } j \in I,\ v \models \phi_j \\
v \models \neg\phi & \iff & \text{it is false that } v \models \phi \\
t \models P & \iff & [\![t]\!] \in P
\end{array}$$

Figure 6.2: Satisfaction relation $\models$ for the logic $\mathcal{F}$.

Notice that a computation formula is just one of the observations $P \in \mathfrak{P}$. For example, it can be $\Box$, $\mathbf{P}_{>0.5}$, $(s \rightarrowtail r)$ depending on the effects present in the language. Therefore, $t \models P$ tests the shape of the computation tree of t without looking at its possible return values. This is consistent with the fact that ECPS computations are not expected to return.

However, this is unlike computation formulas in EPCF logic, $o\phi$, which test whether return values satisfy ϕ (Section 2.6). Therefore, P is no longer a modality in the same sense as o because it does not lift formulas. This is why we called P an *observation*. In the logic $\mathcal{F}$, it can still be argued that P is a modality in the traditional sense because it takes the value formula ϕ to another value formula $\phi \mapsto P$.

In both $\mathcal{F}$ and $\mathcal{V}$ the value formulas of type `nat` are obtained from the natural numbers, arbitrary conjunctions and disjunctions, and negation. However, the basic formulas of function type are different. In $\mathcal{V}$:

$$v \models (\lambda x{:}\mathtt{nat}.\downarrow) \mapsto \Diamond$$

means that $v : \neg(\neg(\mathtt{nat}))$ *may* terminate when given argument $\lambda x{:}\mathtt{nat}.\downarrow$. In $\mathcal{F}$, an analogous statement is:

$$v \models ((\vee_{n \in \mathbb{N}} \{n\}) \mapsto \Box) \mapsto \Diamond.$$

This says that v may terminate when given as argument a function that satisfies $\psi =$

$(\vee_{n\in\mathbb{N}}\{n\}) \mapsto \Box$. Notice that $(\lambda x{:}\mathtt{nat}.\downarrow)$ indeed satisfies ψ.

There is no need to include logical connectives at the level on computation formulas because they can be encoded in value formulas. For example $\phi \mapsto \wedge_{i\in I} P_i$ can be expressed as:

$$\wedge_{i\in I}(\phi \mapsto P_i).$$

The statement $t \models \wedge_{i\in I} P_i$ can instead be expressed as:

$$\lambda x{:}\mathtt{unit}.t \models \wedge_{i\in I}(true \mapsto P_i).$$

The indexing set I in $\wedge_{i\in I}$ may be uncountable. However, the sets of values and computations are countable. Since logical formulas are interpreted over values and computations, all conjunctions and disjunctions are semantically equivalent to countable ones.

The following example compares logical formulas for EPCF with $\mathcal{F}$-formulas:

Example 6.1.3 (Nondeterminism)**.** Recall the EPCF logic formula $\phi_2 = \{3\} \mapsto \Diamond\{2\}$ from Examples 2.2.1 and 2.6.1 and the function:

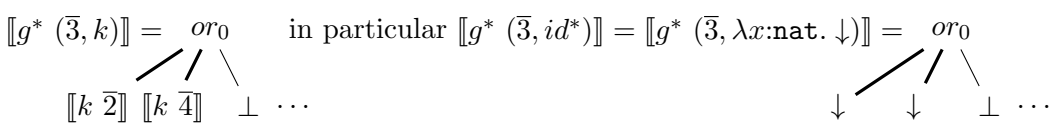
$$g = \lambda n{:}\mathbb{N}.or(\mathbf{pred}\ n,\ \mathbf{succ}\ n).$$

We have previously established that $g \models_{EPCF} \phi_2$ because, when given argument $\overline{3}$ in the empty stack id, $(id, g\ \overline{3})$ *may* return $\overline{2}$.

Now consider the CPS translation (Section 4.1) of the function g into ECPS, named g^*. Apart from the natural number argument that g receives, g^* also receives a continuation k to which it passes its result.

$$\begin{array}{l}
g^* = \lambda(n,k){:}(\mathtt{nat}, \neg\mathtt{nat}).\\
\quad (\lambda k'{:}\neg\mathtt{nat}.or(\overline{0},\ x.(\lambda(y,k''){:}(\mathtt{nat},\neg\mathtt{nat}).\\
\qquad\qquad (\lambda k''' : \neg\mathtt{nat.case}\ y\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow (\mathbf{pred}\ n)^*\ k''',\\
\qquad\qquad\qquad \mathtt{succ}(y') \Rightarrow \mathtt{case}\ y'\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow (\mathbf{succ}\ n)^*\ k''',\\
\qquad\qquad\qquad\qquad \mathtt{succ}(y'') \Rightarrow loop^*\ k'''\}\}\\
\qquad\qquad)\ k''\\
\qquad)\ (x,k')\\
\quad)\\
)\ k\,.
\end{array}$$

The tree of $(g^*\ (\overline{3},k))$ is:



The formula ϕ_2 could be translated to the $\mathcal{F}$ formula:

$$\phi_2^* = ((\{3\},\{2\} \mapsto \Diamond) \mapsto \Diamond) \wedge ((\{3\},\{2\} \mapsto \Box) \mapsto \Diamond).$$

Intuitively, $(\{3\},\{2\} \mapsto \Diamond) \mapsto \Diamond$ says that, when g^* is given as arguments $\overline{3}$ and a continuation k that satisfies $\{2\} \mapsto \Diamond$, the computation *may* eventually terminate. So if g indeed returns $\overline{2}$ this formula will be satisfied. Similarly for $(\{3\},\{2\} \mapsto \Box) \mapsto \Diamond$. Therefore, we can see that $g \models_{EPCF} \phi_2$ implies $g^* \models_{\mathcal{F}} \phi_2^*$.

The conjunction in ϕ_2^* is over all nontrivial $P \in \mathfrak{P}$ that appear in $\{2\} \mapsto P$, namely $\Diamond$ and $\Box$.

6.2 Logical Equivalence

The logics $\mathcal{F}$ and $\mathcal{V}$ induce a notion of program equivalence defined below:

Definition 6.2.1 (Logical preorder and equivalence)**.** *Consider a fragment $\mathcal{L}$ of one of the logics $\mathcal{F}$ or $\mathcal{V}$. For any closed values $\vdash v_1 : A$ and $\vdash v_2 : A$:*

$$v_1 \sqsubseteq_{\mathcal{L}} v_2 \iff \forall \phi : A \text{ in } \mathcal{L}.\ (v_1 \models_{\mathcal{L}} \phi \implies v_2 \models_{\mathcal{L}} \phi).$$

And for any closed computations $\vdash s_1$ and $\vdash s_2$:

$$s_1 \sqsubseteq_{\mathcal{L}} s_2 \iff \forall P \text{ in } \mathcal{L}.\ (s_1 \models_{\mathcal{L}} P \implies s_2 \models_{\mathcal{L}} P).$$

Two terms (values or computations) are logically equivalent, $t_1 \equiv_{\mathcal{L}} t_2$, if $t_1 \sqsubseteq_{\mathcal{L}} t_2$ and $t_2 \sqsubseteq_{\mathcal{L}} t_1$.

The definition of logical equivalence provides a convenient way of proving that two programs are *not* equivalent: we just need to find a formula that one of them does not satisfy but the other does. For example:

Example 6.2.2 (Probabilistic choice)**.** Consider the following ECPS functions, where $m_{\overline{1},\overline{2}}$, $n_{\overline{1},\overline{2},\overline{1},\overline{3}}$ and $n'_{\overline{1},\overline{2},\overline{1},\overline{3}}$ are defined as in Example 5.2.5:

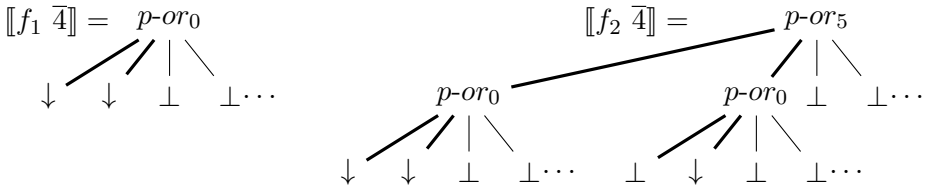
$$f_1 = \lambda x{:}\mathtt{nat}.m_{\overline{1},\overline{2}}$$

$$f_2 = \lambda x{:}\mathtt{nat}.\mathtt{case}\ x\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow n_{\overline{1},\overline{2},\overline{1},\overline{3}},\ \mathtt{succ}(y) \Rightarrow n'_{\overline{1},\overline{2},\overline{1},\overline{3}}\}.$$

Consider the $\mathcal{F}$-formula:

$$\phi = \{4\} \mapsto \mathbf{P}_{>0.9}.$$

Formula ϕ distinguishes between these two functions because: $f_1 \models_{\mathcal{F}} \phi$ but $f_2 \not\models_{\mathcal{F}} \phi$. This can be seen by looking at their computation trees:



Therefore, f_1 and f_2 are not $\mathcal{F}$-logically equivalent. However, we can see that $f_1 \models_{\mathcal{F}} \{0\} \mapsto \mathbf{P}_{>0.9}$ and $f_2 \models_{\mathcal{F}} \{0\} \mapsto \mathbf{P}_{>0.9}$.

It is the program equivalence induced by $\mathcal{F}$, rather than $\mathcal{V}$, that we are mostly interested in. This is because $\mathcal{F}$ enforces a natural separation between ECPS terms and program properties. Using $\mathcal{F}$, we can specify program properties without knowing the syntax of the programming language.

It can be easily seen that $\sqsubseteq_{\mathcal{F}^+}$ is a preorder and $\equiv_{\mathcal{F}}$ is an equivalence relation. Compatibility, the property that related programs can be substituted for variables in related contexts, is established in the main theorem of this chapter. The proof of this theorem appears at the end of the next section.

Theorem 6.2.3. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations:*

1. *Applicative $\mathfrak{P}$-similarity, $\precsim$, coincides with the logical preorder induced by the logic $\mathcal{F}^+$, $\sqsubseteq_{\mathcal{F}^+}$. Hence, the open extension of the $\mathcal{F}^+$-logical preorder $\sqsubseteq^{\circ}_{\mathcal{F}^+}$ is compatible.*

2. *Applicative $\mathfrak{P}$-bisimilarity, $\sim$, coincides with the logical equivalence induced by the logic $\mathcal{F}$, $\equiv_{\mathcal{F}}$. Hence, the open extension of $\mathcal{F}$-logical equivalence $\equiv^{\circ}_{\mathcal{F}}$ is compatible.*

This theorem is important because, when combined with the result of the next chapter, it shows that the logic $\mathcal{F}$ characterises contextual equivalence. This answers the main question asked in the introduction.

6.3 Logical Equivalence Coincides with Bisimilarity

The aim of this section is to show that program equivalence induced by the logic $\mathcal{F}$ coincides with applicative $\mathfrak{P}$-bisimilarity, defined in the previous chapter. First, we show that this is the case for the logic $\mathcal{V}$. The proof appears in Appendix C.

Proposition 6.3.1. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations:*

1. *Applicative $\mathfrak{P}$-similarity, $\precsim$, coincides with the logical preorder induced by the logic $\mathcal{V}^+$, $\sqsubseteq_{\mathcal{V}^+}$. Therefore, the open extension of $\sqsubseteq_{\mathcal{V}^+}$ is compatible.*

2. *Applicative $\mathfrak{P}$-bisimilarity, $\sim$, coincides with the logical equivalence induced by the logic $\mathcal{V}$, $\equiv_{\mathcal{V}}$. Therefore, the open extension of $\equiv_{\mathcal{V}}$ is compatible.*

Next, we show that the logics $\mathcal{F}$ and $\mathcal{V}$ are in fact equivalent. This is done by translating $\mathcal{F}$-formulas into $\mathcal{V}$, and vice-versa, and proving that the satisfaction relation is preserved. Define a translation from $\mathcal{F}$ to $\mathcal{V}$, $(-)^\flat$, and a translation from $\mathcal{V}$ to $\mathcal{F}$, $(-)^\sharp$. The definition appears in Figure 6.3. It makes use of the following lemma, which is similar to a lemma for EPCF proved in [SV18]:

Lemma 6.3.2 (Characteristic formula)**.** *For any fragment $\mathcal{L}$ of $\mathcal{F}$ or $\mathcal{V}$ closed under countable conjunction it is true that*

for any closed value v there exists a formula $\chi_v \in \mathcal{L}$ such that:

$$u \models_{\mathcal{L}} \chi_v \iff v \sqsubseteq_{\mathcal{L}} u.$$

Proof. For each closed value u such that $v \not\sqsubseteq_{\mathcal{L}} u$ we can choose a formula ϕ_u such that $v \models_{\mathcal{L}} \phi_u$ but $u \not\models_{\mathcal{L}} \phi_u$. Define χ_v to be:

$$\chi_v = \wedge_{\{u \mid v \not\sqsubseteq_{\mathcal{L}} u\}} \phi_u.$$

We can see that $u \not\models_{\mathcal{L}} \chi_v \iff v \not\sqsubseteq_{\mathcal{L}} u$, which is what we need. □

Theorem 6.3.3. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations, the logics $\mathcal{F}^+$ and $\mathcal{V}^+$ are equi-expressive.*

$$((\phi_1, \ldots, \phi_n) \mapsto P)^\flat = \bigwedge\{(w_1, \ldots, w_n) \mapsto P \mid w_1 \models_{\mathcal{V}} \phi_1^\flat, \ldots, w_n \models_{\mathcal{V}} \phi_n^\flat\}$$

$$((w_1, \ldots, w_n) \mapsto P)^\sharp = (\chi_{w_1}, \ldots, \chi_{w_n}) \mapsto P$$

$$\{n\}^\flat = \{n\} \qquad \{n\}^\sharp = \{n\}$$

$$P^\flat = P \qquad P^\sharp = P$$

$$(\vee_{i\in I}\phi_i)^\flat = \vee_{i\in I}\phi_i^\flat \qquad (\vee_{i\in I}\phi_i)^\sharp = \vee_{i\in I}\phi_i^\sharp$$

$$(\wedge_{i\in I}\phi_i)^\flat = \wedge_{i\in I}\phi_i^\flat \qquad (\wedge_{i\in I}\phi_i)^\sharp = \wedge_{i\in I}\phi_i^\sharp$$

$$(\neg\phi)^\flat = \neg\phi^\flat \qquad (\neg\phi)^\sharp = \neg\phi^\sharp$$

The formula χ_{w_i} is the characteristic formula of w_i in the logic $\mathcal{F}$, from Lemma 6.3.2.

Figure 6.3: Translation from $\mathcal{F}$ to $\mathcal{V}$ and vice-versa.

1. *For any type A, for any formula ϕ in $\mathcal{F}^+$, $\phi : A$ implies that for any value $\vdash v : A$:*

$$v \models_{\mathcal{F}^+} \phi \iff v \models_{\mathcal{V}^+} \phi^\flat.$$

For any $P \in \mathfrak{P}$ and any computation $\vdash t$:

$$t \models_{\mathcal{F}^+} P \iff t \models_{\mathcal{V}^+} P^\flat.$$

2. *For any type A, for any formula ϕ in $\mathcal{V}^+$, $\phi : A$ implies that for any value $\vdash v : A$:*

$$v \models_{\mathcal{V}^+} \phi \iff v \models_{\mathcal{F}^+} \phi^\sharp.$$

For any $P \in \mathfrak{P}$ and any computation $\vdash t$:

$$t \models_{\mathcal{V}^+} P \iff t \models_{\mathcal{F}^+} P^\sharp.$$

Proof. **Statement 1.** For computation formulas the result is immediate because they do not change when translated.

For value formulas we prove the following property:

$$\Phi(\phi, A) = (\phi : A \implies (\forall \vdash v : A.\ v \models_{\mathcal{F}^+} \phi \iff v \models_{\mathcal{V}^+} \phi^\flat))$$

by induction on the rules in Figure 6.1, which specify when $\phi : A$ is well-formed.

In the case (NAT), $\phi = \{n\}$. The equivalence holds because $\{n\}^\flat = \{n\}$ and the satisfaction relation does not change with the translation.

The cases for the logical connectives follow from the induction hypothesis.

In the case (VAL), $\phi = (\phi_1, \ldots, \phi_n) \mapsto P$. Let $v \models_{\mathcal{F}^+} \phi$ and consider some arbitrary $w_1 \models_{\mathcal{V}^+} \phi_1^\flat, \ldots, w_n \models_{\mathcal{V}^+} \phi_n^\flat$. By the induction hypothesis we know $w_1 \models_{\mathcal{F}^+} \phi_1, \ldots,$

$w_n \models_{\mathcal{F}^+} \phi_n$. So by assumption $[\![v(w_1, \ldots, w_n)]\!] \in P$. Therefore it is true that, in $\mathcal{V}^+$, v satisfies $(w_1, \ldots, w_n) \mapsto P$ so in general $v \models_{\mathcal{V}^+} \phi^\flat$.

For the reverse implication let $v \models_{\mathcal{V}^+} \phi^\flat$ and consider some arbitrary $w_1 \models_{\mathcal{F}^+} \phi_1, \ldots,$ $w_n \models_{\mathcal{F}^+} \phi_n$. By the induction hypothesis $w_1 \models_{\mathcal{V}^+} \phi_1^\flat, \ldots, w_n \models_{\mathcal{V}^+} \phi_n^\flat$, so by assumption $[\![v(w_1, \ldots, w_n)]\!] \in P$. Therefore $v \models_{\mathcal{F}^+} \phi$ as required.

Statement 2. For computation formulas $P^\sharp = P$ so the equivalence holds.

For value formulas proceed by induction on the type A. If $A = \texttt{nat}$, then the formulas ϕ and $\phi^\sharp$ represent the same set of natural numbers. Therefore, $v \models_{\mathcal{V}^+} \phi$ is equivalent to $v \models_{\mathcal{F}^+} \phi^\sharp$. For $A = \texttt{unit}$ the only formulas are $true$ and $false$ so the equivalence holds trivially. For $A = \neg(B_1, \ldots, B_n)$ the induction hypothesis is, for each B_i:

$$\text{For any formula } \phi' \text{ in } \mathcal{V}^+,\ \phi' : B_i \text{ implies that for any value } \vdash v : B_i\text{:}$$

$$v \models_{\mathcal{V}^+} \phi' \iff v \models_{\mathcal{F}^+} \phi'^\sharp.$$

We do an additional induction on ϕ:

Case $\phi = (w_1, \ldots, w_n) \mapsto P : \neg(B_1, \ldots, B_n)$**.** Assume $v \models_{\mathcal{V}^+} \phi$, that is $[\![v(w_1, \ldots, w_n)]\!] \in P$. We need to prove that for any $u_1 : B_1, \ldots, u_n : B_n$ such that $u_i \models_{\mathcal{F}^+} \chi_{w_i}$ for all i, we have $[\![v(u_1, \ldots, u_n)]\!] \in P$.

By the definition of χ_{w_i} we know that $w_i \sqsubseteq_{\mathcal{F}^+} u_i$. We can show $w_i \sqsubseteq_{\mathcal{V}^+} u_i$ as follows: consider an arbitrary $\psi : B_i$ such that $w_i \models_{\mathcal{V}^+} \psi$. Then by the induction hypothesis for the type B_i we know $w_i \models_{\mathcal{F}^+} \psi^\sharp$. Hence deduce $u_i \models_{\mathcal{F}^+} \psi^\sharp$ from $w_i \sqsubseteq_{\mathcal{F}^+} u_i$. Again from the induction hypothesis for B_i, we have $u_i \models_{\mathcal{V}^+} \psi$, as required.

Now that we have $w_i \sqsubseteq_{\mathcal{V}^+} u_i$ we can use compatibility of $\sqsubseteq_{\mathcal{V}^+}$, Proposition 6.3.1, and reflexivity to deduce:

$$v(w_1, \ldots, w_n) \sqsubseteq_{\mathcal{V}^+} v(u_1, \ldots, u_n).$$

So from $[\![v(w_1, \ldots, w_n)]\!] \in P$ we get the desired result $[\![v(u_1, \ldots, u_n)]\!] \in P$.

For the reverse implication assume $v \models_{\mathcal{F}^+} (\chi_{w_1}, \ldots, \chi_{w_n}) \mapsto P$. We need to prove that $[\![v(w_1, \ldots, w_n)]\!] \in P$. This follows from the fact that $w_i \models_{\mathcal{F}^+} \chi_{w_i}$ because $\sqsubseteq_{\mathcal{F}^+}$ is reflexive.

Case $\phi = \vee_{i \in I} \varphi_i : \neg(B_1, \ldots, B_n)$**.** From the type of ϕ we know that for all i, $\varphi_i : \neg(B_1, \ldots, B_n)$. This means that the induction hypothesis for φ_i gives us:

$$\text{For any value } \vdash v : \neg(B_1, \ldots, B_n)\text{:}$$

$$v \models_{\mathcal{V}^+} \varphi_i \iff v \models_{\mathcal{F}^+} \varphi_i^\sharp.$$

Assume $v \models_{\mathcal{V}^+} \vee_{i \in I} \varphi_i$. There exists $j \in I$ such that $v \models_{\mathcal{V}^+} \varphi_j$. By the induction hypothesis for φ_j we have that $v \models_{\mathcal{F}^+} \varphi_j^\sharp$. So $v \models_{\mathcal{F}^+} \phi^\sharp$.

The reverse implication is analogous.

Case $\phi = \wedge_{i\in I}\varphi_i : \neg(B_1, \ldots, B_n)$**.** Analogous to the previous case. □

Theorem 6.3.4. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations, the logics $\mathcal{F}$ and $\mathcal{V}$ are equi-expressive.*

Proof. We need to prove the same statements as in Theorem 6.3.3, where $\mathcal{V}^+$ is replaced by $\mathcal{V}$ and $\mathcal{F}^+$ is replaced by $\mathcal{F}$. The proof is very similar and the differences are pointed out in Appendix C. □

Using all the results in this section we can finally prove Theorem 6.2.3, which says that $\mathcal{F}$-logical equivalence coincides with bisimilarity.

Proof of Theorem 6.2.3. From Theorems 6.3.3 and 6.3.4 we can deduce that:

$$v \sqsubseteq_{\mathcal{V}^+} u \iff v \sqsubseteq_{\mathcal{F}^+} u$$

$$v \equiv_{\mathcal{V}} u \iff v \equiv_{\mathcal{F}} u$$

and similarly for computations.

Then by Proposition 6.3.1 we have $(\precsim) = (\sqsubseteq_{\mathcal{F}^+})$ and $(\sim) = (\equiv_{\mathcal{F}})$. From Theorem 5.3.7 we know $\precsim^\circ$ and $\sim^\circ$ are compatible, so this is also the case for $\sqsubseteq^\circ_{\mathcal{F}^+}$ and $\equiv^\circ_{\mathcal{F}}$. □

Notice that the proofs of Theorems 6.3.3 and 6.3.4 make use of compatibility of $\sqsubseteq_{\mathcal{V}^+}$ and $\equiv_{\mathcal{V}}$, which was established via Howe's method. So a direct proof of Theorem 6.2.3 would require us to prove a compatibility property of $\sqsubseteq_{\mathcal{F}^+}$ first. As we have seen in the previous chapter, proofs of compatibility are laborious. Therefore, the method of going through the logic $\mathcal{V}$ to prove Theorem 6.2.3 is justified.

6.4 Chapter Summary

This chapter defined a logic $\mathcal{F}$ in which each formula expresses a property of an ECPS program (Definition 6.1.1). Formulas which concern computations are elements of the set of observations $\mathfrak{P}$. Formulas for function values have the form $(\phi_1, \phi_2, \ldots, \phi_n) \mapsto P$. They assert that, if the arguments $x_1, \ldots, x_n$ of a function satisfy $\phi_1, \ldots, \phi_n$ respectively, the resulting computation is in $P \in \mathfrak{P}$.

Recall the successor function from equation 3.5.1:

$$f = \lambda(n, k){:}(\texttt{nat}, \neg\texttt{nat}).(k\ \texttt{succ}(n)) : \neg(\texttt{nat}, \neg\texttt{nat}).$$

We can see that it satisfies the following formula:

$$\phi = (\{2\}, \{3\} \mapsto \Diamond) \mapsto \Diamond.$$

This says that, given argument $\overline{2}$ and a continuation k which may terminate for input $\overline{3}$, the body of f may terminate. Consider the formula $\phi' = \{2\} \mapsto \Diamond\{3\}$, which describes a program in direct style that may return $\overline{3}$. This is similar to the formulas discussed in Example 2.2.1, but it is not a valid $\mathcal{F}$-formula. Formula ϕ can be viewed as a translation of ϕ' describing a program in continuation-passing style instead.

The goal of this chapter was to prove that program equivalence induced by the logic $\mathcal{F}$ coincides with applicative bisimilarity (Theorem 6.2.3). Hence, according to Theorem 5.3.7

from the previous chapter, for a decomposable set $\mathfrak{P}$ of Scott-open observations, $\mathcal{F}$-logical equivalence is compatible.

To prove this, we defined a logic $\mathcal{V}$ which is similar to $\mathcal{F}$, but in which ECPS values can appear inside formulas. From this point of view $\mathcal{V}$ is not satisfactory as a specification language because we need knowledge of the syntax of ECPS to express properties of programs.

Nevertheless, it is relatively straightforward to prove that $\mathcal{V}$-logical equivalence coincides with applicative bisimilarity (Proposition 6.3.1), but we do not know any proof of a similar result for $\mathcal{F}$-logical equivalence. Instead, we proved that $\mathcal{F}$ and $\mathcal{V}$ are equi-expressive, using translations between the two logics (Theorem 6.3.4). Thus, we obtained a proof of Theorem 6.2.3. This theorem will be used in the next chapter to prove the main result of the dissertation: that $\mathcal{F}$-logical equivalence coincides with contextual equivalence.

Chapter 7

Contextual Equivalence for ECPS

This chapter defines contextual equivalence for ECPS coinductively and proves that it is compatible and an equivalence relation. Then, contextual equivalence is proved to coincide with applicative bisimilarity for ECPS (Theorem 7.2.2). As a result, they both coincide with logical equivalence induced by the logic $\mathcal{F}$ (Corollary 7.2.3). Therefore, the logic $\mathcal{F}$ characterises contextual equivalence. Establishing this was the main goal of the dissertation. Finally, we present an alternative definition of contextual equivalence using program contexts, and prove it equivalent with the coinducitve definition (Theorem 7.3.7).

7.1 Contextual Equivalence Coinductively

This section presents a coinductive definition of contextual preorder and equivalence, initially proposed by Lassen [Las98] and Gordon [Gor98]. Contextual preorder is defined as the greatest compatible and adequate relation, for a suitable definition of adequacy.

The advantage of this definition is that we do not need to deal with contexts explicitly. This is important in the case of ECPS where contexts are duplicated because of the distinction between values and computations, as Section 7.3 will show.

Definition 7.1.1 (Adequacy)**.** *Consider a set of observations $\mathfrak{P}$ and a well-typed relation on possibly open terms $\mathcal{R} = (\mathcal{R}^{\mathfrak{v}}_A, \mathcal{R}^{\mathfrak{c}})$, where $\mathcal{R}^{\mathfrak{c}}$ relates computations. The relation $\mathcal{R}$ is $\mathfrak{P}$-adequate if:*

$$\forall s, t.\ \vdash s\ \mathcal{R}^{\mathfrak{c}}\ t \implies \forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P.$$

The relation $\mathcal{R}$ is $\mathfrak{P}$-biadequate if:

$$\forall s, t.\ \vdash s\ \mathcal{R}^{\mathfrak{c}}\ t \implies \forall P \in \mathfrak{P}.\ [\![s]\!] \in P \iff [\![t]\!] \in P.$$

The definition of adequacy is motivated by the fact that, in ECPS, the observable behaviour of a program s, in the sense of Section 2.1, is whether $s \in P$, where $P \in \mathfrak{P}$. So adequacy checks that t simulates the observable behaviour of s.

Definition 7.1.2 (Contextual preorder)**.** *Let $\mathbb{CA}$ be the set of well-typed relations on possibly open terms that are both compatible and $\mathfrak{P}$-adequate. Define the contextual preorder $\sqsubseteq_{ctx}$ to be $\bigcup \mathbb{CA}$.*

The next proposition establishes that contextual preorder is a precongruence. This will help prove that contextual equivalence is a congruence, hence a well-behaved notion of program equivalence.

Proposition 7.1.3. *The contextual preorder $\sqsubseteq_{ctx}$ is a preorder, and is moreover compatible and $\mathfrak{P}$-adequate. Thus, it is the greatest compatible and $\mathfrak{P}$-adequate preorder.*

Proof. The proof follows the structure of the proof of Proposition 4 from [LGL17b]. To prove reflexivity, we show that the open identity relation $\mathcal{I}$ is in $\mathbb{CA}$. To show transitivity it suffices to show that the composition of relations in $\mathbb{CA}$ is itself in $\mathbb{CA}$. The relation $\sqsubseteq_{ctx}$ is shown compatible using the definition of $\mathbb{CA}$. The complete proof of this proposition can be found in Appendix D. □

Definition 7.1.4 (Contextual equivalence)**.** *Let $\mathbb{CAS}$ be the set of well-typed relations on possibly open terms that are both compatible and $\mathfrak{P}$-biadequate. Define contextual equivalence $\equiv_{ctx}$ to be $\bigcup \mathbb{CAS}$.*

Proposition 7.1.5. *Contextual equivalence is the intersection of the contextual preorder with its converse:*

$$(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}.$$

The proof of the above proposition appears in Appendix D. This relationship between contextual equivalence and preorder in ECPS is expected. It also holds in the case of the untyped λ-calculus with generic effects, as shown by [LGL17a]. Finally, we can prove contextual equivalence is a congruence.

Proposition 7.1.6. *Contextual equivalence $\equiv_{ctx}$ is an equivalence relation, and is moreover compatible and $\mathfrak{P}$-biadequate. Thus, it is the greatest compatible and $\mathfrak{P}$-biadequate equivalence relation.*

Proof. From Proposition 7.1.5 we know $(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$. We have proved $\sqsubseteq_{ctx}$ is a preorder, so $(\sqsubseteq_{ctx})^{op}$ is also a preorder. The intersection of two preorders is another preorder so $\equiv_{ctx}$ is a preorder. Moreover, it is symmetric because it is the intersection of a relation and its converse. Thus $\equiv_{ctx}$ is an equivalence relation. In the proof of Proposition 7.1.5 we have shown $(\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$ is compatible and biadequate, so $\equiv_{ctx}$ is as well. □

7.2 Contextual Equivalence Coincides with Bisimilarity

This section shows that the coinductive notion of contextual equivalence (Definition 7.1.4) coincides with applicative bisimilarity. Using the results from the previous chapter about logical equivalence, we can in fact deduce that all notions of program equivalence for ECPS considered so far are the same. Thus, we have defined a logic $\mathcal{F}$ whose induced program equivalence characterises contextual equivalence. This is the main contribution of this dissertation.

To obtain this result, the set of observations $\mathfrak{P}$ needs to satisfy one more condition apart from decomposability and Scott-openness, named *consistency*.

Definition 7.2.1 (Consistency)**.** *A set of observations $\mathfrak{P}$ is consistent if there exists at least one observation $P_0 \in \mathfrak{P}$ such that:*

1. *$P_0 \neq Trees_\Sigma$ and*
2. *there exists at least one computation t_0 such that $[\![t_0]\!] \in P_0$.*

This definition says that $\mathfrak{P}$ contains a non-trivial observation P_0 which contains at least one computation tree $[\![t_0]\!]$. If this were not the case, then contextual equivalence would equate all terms, including natural numbers.

On the other hand, applicative bisimilarity would equate all computations, but not all natural numbers because $v \sim^{\mathfrak{v}}_{\mathtt{nat}} w \iff v = w$. Hence, it would not be the case that applicative bisimilarity coincides with contextual equivalence.

However, program equivalence induced by a set of observations $\mathfrak{P}$ which does not satisfy consistency is not meaningful because not enough programs are distinguished. In particular, all computations would be equivalent to *loop*, the program that runs forever. Indeed, for all example of effects considered so far $\mathfrak{P}$ is consistent. Therefore consistency is a reasonable assumption.

Theorem 7.2.2. *Consider a decomposable set of Scott-open observations $\mathfrak{P}$ which is consistent. Then:*

1. *The open extension of applicative $\mathfrak{P}$-similarity, $\precsim^\circ$, coincides with the contextual preorder, $\sqsubseteq_{ctx}$.*
2. *The open extension of applicative $\mathfrak{P}$-bisimilarity, $\sim^\circ$, coincides with contextual equivalence, $\equiv_{ctx}$.*

Proof. **We first show** $(\precsim^\circ) = (\sqsubseteq_{ctx})$. We have shown in Theorem 5.3.7 that $\precsim^\circ$ is compatible under the current assumptions. Consider $\vdash s \precsim^\circ t$. Then $\vdash s \precsim t$ so by the definition of simulation we know that:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P.$$

Therefore, $\precsim^\circ$ is adequate. Being both compatible and adequate, $\precsim^\circ$ is included in $\sqsubseteq_{ctx}$.

Now we need to show $(\sqsubseteq_{ctx}) \subseteq (\precsim^\circ)$. We first show that $\sqsubseteq_{ctx}$ restricted to closed terms is included in $\precsim$, then extend this to open terms. To do this, we show $\sqsubseteq_{ctx}$ restricted to closed terms is a simulation by checking it satisfies the four conditions in the definition of simulation (Definition 5.2.1).

We will concentrate on the case for natural numbers because it is the most interesting. It makes use of the existence of computation t_0 and observation P_0 from the definition of consistency. The complete proof can be found in Appendix D.

2. Assume $\vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\mathtt{nat}}\ u$. Consider the computation:

$$loop = (\mu f.\lambda x{:}\mathtt{nat}.f\ x)(\overline{0}).$$

This computation leads to an infinite chain of reductions:

$$(\mu f.\lambda x{:}\mathtt{nat}.f\ x)\ \overline{0} \longrightarrow (\lambda x{:}\mathtt{nat}.(\lambda y{:}\mathtt{nat}.(\mu f.\lambda x{:}\mathtt{nat}.f\ x)\ y)\ x)\ \overline{0} \longrightarrow^2$$

$$(\mu f.\lambda x{:}\mathtt{nat}.f\ x)\ \overline{0} \longrightarrow^*$$

so $[\![loop]\!] = \bot$.

Since v and u are closed values there exist m and n in $\mathbb{N}$ such that $u = \overline{n}$ and $v = \overline{m}$. Consider the following computation:

$$\texttt{if } x = \overline{n} \texttt{ then } t_0 \texttt{ else } loop = \texttt{case } x \texttt{ in } \{\texttt{zero} \Rightarrow loop,\ \texttt{succ}(x_1) \Rightarrow \texttt{case} \ldots x_n \texttt{ in } \{\texttt{zero} \Rightarrow t_0,\ \texttt{succ}(x_{n+1}) \Rightarrow loop\} \ldots\}$$

which evaluates to t_0 if $x = \overline{n}$ or loops otherwise. We know by consistency that $[\![t_0]\!] \in P_0 \neq \mathit{Trees}_\Sigma$. It must be the case that $\bot \notin P_0$ because otherwise by upwards-closure of P_0 we would obtain $P_0 = \mathit{Trees}_\Sigma$.

By compatibility and reflexivity of $\sqsubseteq_{ctx}$ we know that:

$$\vdash (\texttt{if } \overline{n} = \overline{n} \texttt{ then } t_0 \texttt{ else } loop)\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ (\texttt{if } \overline{m} = \overline{n} \texttt{ then } t_0 \texttt{ else } loop). \tag{7.2.1}$$

Suppose by contradiction that $n \neq m$. Then:

$$[\![\texttt{if } \overline{n} = \overline{n} \texttt{ then } t_0 \texttt{ else } loop]\!] = [\![t_0]\!] \in P_0$$

$$[\![\texttt{if } \overline{m} = \overline{n} \texttt{ then } t_0 \texttt{ else } loop]\!] = [\![loop]\!] = \bot \notin P_0.$$

But this contradicts equation 7.2.1 because $\sqsubseteq_{ctx}$ is adequate. Therefore, $n = m$ and $v = u$ as required.

3. By adequacy of $\sqsubseteq_{ctx}$.
4. By compatibility of $\sqsubseteq_{ctx}$.

So we have established $(\sqsubseteq_{ctx}) \subseteq (\precsim)$ for closed terms. Now we need to prove $(\sqsubseteq_{ctx}) \subseteq (\precsim^\circ)$ in general. To do this, we consider computations and each type of value separately. Again, we concentrate on the case for natural numbers. The cases for computations and function values are proved using compatibility of $\sqsubseteq_{ctx}$. We give the proof of the computation case as an example. The proof for function values can be found in Appendix D.

If $\overrightarrow{x_j : A_j} \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\texttt{nat}}\ w$ then for any $\vdash \overrightarrow{u_j : A_j}$ there exist natural numbers m and k such that $v[\overrightarrow{u_j/x_j}] = \overline{m}$ and $w[\overrightarrow{u_j/x_j}] = \overline{k}$. We want to show $m = k$ and therefore:

$$\forall(\vdash \overrightarrow{u_j : A_j}).\ v[\overrightarrow{u_j/x_j}] = w[\overrightarrow{u_j/x_j}].$$

This would imply by the definition of simulation and of open extension that $\overrightarrow{x_i : A_i} \vdash v \precsim^\circ w$, as required. To do this consider the computation:

$$eq(y, v) = (\mu f.\lambda(y_0, x_0){:}(\texttt{nat}, \texttt{nat}).\ \texttt{case } y_0 \texttt{ in } \{\texttt{zero} \Rightarrow \texttt{case } x_0 \texttt{ in } \{\texttt{zero} \Rightarrow t_0,\ \texttt{succ}(x_1) \Rightarrow loop\}\ ,\ \texttt{succ}(y_1) \Rightarrow \texttt{case } x_0 \texttt{ in } \{\texttt{zero} \Rightarrow loop,\ \texttt{succ}(x_1) \Rightarrow f\ (y_1, x_1)\}\})\ (y, v).$$

If $y = v$ then $[\![eq(y,v)]\!] = [\![t_0]\!]$, otherwise $[\![eq(y,v)]\!] = \bot$. Using context weakening and compatibility and reflexivity of $\sqsubseteq_{ctx}$ deduce that:

$$\overrightarrow{x_j : A_j}, y : \mathtt{nat} \vdash eq(y,v)\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ eq(y,w)$$

and then again by compatibility

$$\vdash \lambda(\overrightarrow{x_j}, y){:}(\overrightarrow{A_j}, \mathtt{nat}).eq(y,v)\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_j},\mathtt{nat})}\ \lambda(\overrightarrow{x_j}, y){:}(\overrightarrow{A_j}, \mathtt{nat}).eq(y,w).$$

From $(\sqsubseteq_{ctx}) \subseteq (\precsim)$ for closed terms we can now establish that:

$$\vdash \lambda(\overrightarrow{x_j}, y){:}(\overrightarrow{A_j}, \mathtt{nat}).eq(y,v) \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{A_j},\mathtt{nat})} \lambda(\overrightarrow{x_j}, y){:}(\overrightarrow{A_j}, \mathtt{nat}).eq(y,w)$$

so from the definition of $\precsim$ and the fact that reduction preserves similarity (Lemma 5.2.4) we know that:

$$\forall(\vdash \overrightarrow{u_j : A_j}).\ \forall(\overline{n} : \mathtt{nat}).\ \vdash eq(\overline{n}, v[\overrightarrow{u_j/x_j}]) \precsim^{\mathfrak{c}} eq(\overline{n}, w[\overrightarrow{u_j/x_j}]). \tag{7.2.2}$$

Assume by contradiction that there exists $\overrightarrow{u_j : A_j}$ such that $v[\overrightarrow{u_j/x_j}] \neq w[\overrightarrow{u_j/x_j}]$. Choose $\overline{n} = v[\overrightarrow{u_j/x_j}]$. Then:

$$[\![eq(\overline{n}, v[\overrightarrow{u_j/x_j}])]\!] = [\![t_0]\!] \in P_0$$

$$[\![eq(\overline{n}, w[\overrightarrow{u_j/x_j}])]\!] = \bot \notin P_0$$

but this contradicts equation 7.2.2 because of the definition of $\precsim$. Therefore, it must be the case that $\forall \vdash \overrightarrow{u_j : A_j}.\ v[\overrightarrow{u_j/x_j}] = w[\overrightarrow{u_j/x_j}]$, which is what he had to prove.

If $\overrightarrow{x_i : A_i} \vdash s\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ t$ then by compatibility of $\sqsubseteq_{ctx}$ we know that:

$$\vdash \lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.s\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.t.$$

So using $(\sqsubseteq_{ctx}) \subseteq (\precsim)$ for closed terms we have:

$$\vdash \lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.s \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.t.$$

By the definition of $\precsim$ for function values we can deduce:

$$\forall(\vdash \overrightarrow{v_i : A_i}).\ \vdash (\lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.s)\ (\overrightarrow{v_i}) \precsim^{\mathfrak{c}} (\lambda(\overrightarrow{x_i}){:}\overrightarrow{A_i}.t)\ (\overrightarrow{v_i}).$$

From Lemma 5.2.4 we know that reduction preserves similarity so:

$$\forall(\vdash \overrightarrow{v_i : A_i}).\ \vdash s[\overrightarrow{v_i/x_i}] \precsim^{\mathfrak{c}} t[\overrightarrow{v_i/x_i}]$$

which by the definition of open extensions means that:

$$\overrightarrow{x_i : A_i} \vdash s \precsim^{\circ,\mathfrak{c}} t.$$

Now show that $(\sim^\circ) = (\equiv_{ctx})$**.** This is done using $(\precsim^\circ) = (\sqsubseteq_{ctx})$ and the facts that $(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$ (Proposition 7.1.5) and $(\sim) = (\precsim) \cap (\precsim^{op})$ (Proposition 5.2.3). The full proof can be found in Appendix D. □

Corollary 7.2.3. *For a decomposable set of Scott-open observations $\mathfrak{P}$ that is consistent, the logic $\mathcal{F}$ characterises contextual equivalence for ECPS. That is:*

1. *The open extension of $\mathcal{F}$-logical preorder coincides with the contextual preorder: $(\sqsubseteq^\circ_{\mathcal{F}^+}) = (\sqsubseteq_{ctx})$.*
2. *The open extension of $\mathcal{F}$-logical equivalence coincides with contextual equivalence: $(\equiv^\circ_{\mathcal{F}}) = (\equiv_{ctx})$.*

Hence, applicative bisimilarity, logical equivalence and contextual equivalence all coincide.

Proof. Recall Theorem 6.2.3 which says that, for a decomposable set of Scott-open observations $\mathfrak{P}$, $(\precsim) = (\sqsubseteq_{\mathcal{F}^+})$ and $(\sim) = (\equiv_{\mathcal{F}})$. From Theorem 7.2.2 we obtain the desired result:

$$(\sqsubseteq^\circ_{\mathcal{F}^+}) = (\sqsubseteq_{ctx}) = (\precsim^\circ)$$

$$(\equiv^\circ_{\mathcal{F}}) = (\equiv_{ctx}) = (\sim^\circ).$$

□

7.3 Contextual Equivalence via Contexts

For completeness, we give a more familiar definition of contextual equivalence using program contexts, following Crary and Harper [CH07] and Pitts [Pit11]. This way of defining contextual equivalence implements the intuition that equal programs behave the same in all contexts. This definition is proved to be the same as the coinductive one (Definition 7.1.4), a fact that is not immediately apparent and which justifies the use of the coinductive definition.

Informally, a context is an ECPS program with a hole. Because ECPS makes a distinction between values and computations, contexts are divided according to whether they accept and produce computations or values. This leads to a duplication of contexts which makes contextual equivalence tedious to work with. This is why we preferred working with the coinductive definition of contextual equivalence to establish the main results of this chapter.

Definition 7.3.1. *Program contexts for ECPS are defined by the following grammar:*

$$C^{\mathfrak{v}}_{\mathfrak{v}} \coloneqq [-]^{\mathfrak{v}} \mid \mathtt{succ}(C^{\mathfrak{v}}_{\mathfrak{v}}) \mid \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.C^{\mathfrak{v}}_{\mathfrak{c}}$$

$$C^{\mathfrak{v}}_{\mathfrak{c}} \coloneqq C^{\mathfrak{v}}_{\mathfrak{v}}(\overrightarrow{w_i}) \mid v(w_1, \ldots, C^{\mathfrak{v}}_{\mathfrak{v}}, \ldots, w_n) \mid (\mu x.C^{\mathfrak{v}}_{\mathfrak{v}})(\overrightarrow{w_i}) \mid (\mu x.v)(w_1, \ldots, C^{\mathfrak{v}}_{\mathfrak{v}}, \ldots, w_n) \mid$$

$$\sigma(C^{\mathfrak{v}}_{\mathfrak{v}}, x.t) \mid \sigma(v, x.C^{\mathfrak{v}}_{\mathfrak{c}}) \mid \mathtt{case}\ C^{\mathfrak{v}}_{\mathfrak{v}}\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \mid$$

$$\mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow C^{\mathfrak{v}}_{\mathfrak{c}},\ \mathtt{succ}(x) \Rightarrow t\} \mid \mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow C^{\mathfrak{v}}_{\mathfrak{c}}\}$$

$$C^{\mathfrak{c}}_{\mathfrak{c}} \coloneqq [-]^{\mathfrak{c}} \mid C^{\mathfrak{c}}_{\mathfrak{v}}(\overrightarrow{w_i}) \mid v(w_1, \ldots, C^{\mathfrak{c}}_{\mathfrak{v}}, \ldots, w_n) \mid (\mu x.C^{\mathfrak{c}}_{\mathfrak{v}})(\overrightarrow{w_i}) \mid (\mu x.v)(w_1, \ldots, C^{\mathfrak{c}}_{\mathfrak{v}}, \ldots, w_n) \mid$$
$$\sigma(v, x.C^{\mathfrak{c}}_{\mathfrak{c}}) \mid \texttt{case}\ v\ \texttt{in}\ \{\texttt{zero} \Rightarrow C^{\mathfrak{c}}_{\mathfrak{c}},\ \texttt{succ}(x) \Rightarrow t\} \mid$$
$$\texttt{case}\ v\ \texttt{in}\ \{\texttt{zero} \Rightarrow s,\ \texttt{succ}(x) \Rightarrow C^{\mathfrak{c}}_{\mathfrak{c}}\}$$
$$C^{\mathfrak{c}}_{\mathfrak{v}} \coloneqq \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.C^{\mathfrak{c}}_{\mathfrak{c}}.$$

In the context $v(w_1, \ldots, C^{\mathfrak{v}}_{\mathfrak{v}}, \ldots, w_n)$, $C^{\mathfrak{v}}_{\mathfrak{v}}$ can appear in any of the positions 1 to n, for any $n \in \mathbb{N}$. Similarly for the other expressions which take multiple arguments.

The notation $C^{\mathfrak{v}}_{\mathfrak{c}}$ stands for a context whose hole can only be filled with a value, and the resulting term is a computation; $C^{\mathfrak{v}}_{\mathfrak{v}}$, $C^{\mathfrak{c}}_{\mathfrak{v}}$ and $C^{\mathfrak{c}}_{\mathfrak{c}}$ should be read analogously.

Filling the whole of a context with a value or computation, as appropriate, is defined by recursion on the structure of contexts. The definition is standard so we present only a few cases:

$$[-]^{\mathfrak{v}}[u] = u$$
$$(\lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).C^{\mathfrak{v}}_{\mathfrak{c}})[u] = \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).C^{\mathfrak{v}}_{\mathfrak{c}}[u]$$
$$((\mu x.C^{\mathfrak{v}}_{\mathfrak{v}})(\overrightarrow{w_i}))[u] = (\mu x.C^{\mathfrak{v}}_{\mathfrak{v}}[u])(\overrightarrow{w_i})$$
$$\ldots$$

Note that contexts can bind free variables in the term that fills their hole.

There are no contexts of the form $\sigma(C^{\mathfrak{c}}_{\mathfrak{v}}, x.t)$, $\texttt{case}\ C^{\mathfrak{c}}_{\mathfrak{v}} \ldots$ or $\texttt{succ}(C^{\mathfrak{c}}_{\mathfrak{v}})$. In these cases $C^{\mathfrak{c}}_{\mathfrak{v}}[t]$ needs to be a value of type $\texttt{nat}$, for some computation t. But the values of type $\texttt{nat}$ are either variables x or natural numbers $\overline{n}$, which do not contain any computation. Hence, they cannot be obtained from $C^{\mathfrak{c}}_{\mathfrak{v}}$.

We write $C[C']$ for composition of contexts. This means replacing the hole, $[-]$ in C with the context C'. We can now define typing judgements for contexts:

Definition 7.3.2. *The typing relation $C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)$ asserts that, given a value $\Gamma' \vdash u : B$, $C^{\mathfrak{v}}_{\mathfrak{c}}[u]$ is a well-formed computation in the environment Γ.*

Similarly, define typing relations:

$$C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash A)$$
$$C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)$$
$$C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash A).$$

These relations are the least relations closed under the rules in Figures 7.1 and 7.2.

Lemma 7.3.3. *Context instantiation and context composition yield well-typed contexts:*

1a. If $C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash A)$ and $\Gamma' \vdash u : B$ then $\Gamma \vdash C^{\mathfrak{v}}_{\mathfrak{v}}[u] : A$.

And the analogous statements for contexts $C^{\mathfrak{v}}_{\mathfrak{c}}$, $C^{\mathfrak{c}}_{\mathfrak{c}}$, $C^{\mathfrak{c}}_{\mathfrak{v}}$.

$$\frac{}{[-]^{\mathfrak{v}} : (\Gamma \vdash A) \Rightarrow (\Gamma \vdash A)}(\text{VV-ID}) \qquad \frac{C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \mathtt{nat})}{\mathtt{succ}(C^{\mathfrak{v}}_{\mathfrak{v}}) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \mathtt{nat})}(\text{VV-NAT})$$

$$\frac{C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma, \overrightarrow{x_i : A_i} \vdash)}{\lambda \overrightarrow{x_i} : \overrightarrow{A_i}.C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \neg(\overrightarrow{A_i}))}(\text{VV-LBD}) \qquad \frac{C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \neg(\overrightarrow{A_i})) \qquad \Gamma \vdash \overrightarrow{w_i} : \overrightarrow{A_i}}{C^{\mathfrak{v}}_{\mathfrak{v}}(\overrightarrow{w_i}) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-APPL})$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (w_1, \ldots, w_{i-1}) : (A_1, \ldots, A_{i-1}) \\ C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash A_i) \qquad \Gamma \vdash (w_{i+1}, \ldots, w_n) : (A_{i+1}, \ldots, A_n)\end{array}}{v(w_1, \ldots, w_{i-1}, C^{\mathfrak{v}}_{\mathfrak{v}}, w_{i+1}, \ldots, w_n) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-APPR}_i)$$

$$\frac{C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma, x : \neg(\overrightarrow{A_i}) \vdash) \qquad \Gamma \vdash \overrightarrow{w_i} : \overrightarrow{A_i}}{(\mu x.C^{\mathfrak{v}}_{\mathfrak{v}})(\overrightarrow{w_i}) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-MUL})$$

$$\frac{\begin{array}{c}\Gamma, x : \neg(\overrightarrow{A_j}) \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (w_1, \ldots, w_{i-1}) : (A_1, \ldots, A_{i-1}) \\ C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash A_i) \qquad \Gamma \vdash (w_{i+1}, \ldots, w_n) : (A_{i+1}, \ldots, A_n)\end{array}}{(\mu x.v)(w_1, \ldots, w_{i-1}, C^{\mathfrak{v}}_{\mathfrak{v}}, w_{i+1}, \ldots, w_n) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-MUR}_i)$$

$$\frac{C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \mathtt{nat}) \qquad \Gamma, x : \mathtt{nat} \vdash t}{\sigma(C^{\mathfrak{v}}_{\mathfrak{v}}, x.t) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-OPL})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma, x : \mathtt{nat} \vdash)}{\sigma(v, x.C^{\mathfrak{v}}_{\mathfrak{c}}) : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-OPR})$$

$$\frac{C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash \mathtt{nat}) \qquad \Gamma \vdash s \qquad \Gamma, x : \mathtt{nat} \vdash t}{\mathtt{case}\ C^{\mathfrak{v}}_{\mathfrak{v}}\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-CASEV})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash) \qquad \Gamma, x : \mathtt{nat} \vdash t}{\mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow C^{\mathfrak{v}}_{\mathfrak{c}},\ \mathtt{succ}(x) \Rightarrow t\} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-CASEL})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma \vdash s \qquad C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash B) \Rightarrow (\Gamma, x : \mathtt{nat} \vdash)}{\mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow C^{\mathfrak{v}}_{\mathfrak{c}}\} : (\Gamma' \vdash B) \Rightarrow (\Gamma \vdash)}(\text{VC-CASER})$$

Figure 7.1: Typing rules for contexts that accept a value.

$$\frac{}{[-]^{\mathfrak{c}} : (\Gamma \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-ID}) \qquad \frac{C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash \neg(\overrightarrow{A_i})) \qquad \Gamma \vdash \overrightarrow{w_i} : \overrightarrow{A_i}}{C^{\mathfrak{c}}_{\mathfrak{v}}(\overrightarrow{w_i}) : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-APPL})$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (w_1, \ldots, w_{i-1}) : (A_1, \ldots, A_{i-1}) \\ C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash A_i) \qquad \Gamma \vdash (w_{i+1}, \ldots, w_n) : (A_{i+1}, \ldots, A_n)\end{array}}{v(w_1, \ldots, w_{i-1}, C^{\mathfrak{c}}_{\mathfrak{v}}, w_{i+1}, \ldots, w_n) : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-APPR}_i)$$

$$\frac{C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma, x : \neg(\overrightarrow{A_i}) \vdash) \qquad \Gamma \vdash \overrightarrow{w_i} : \overrightarrow{A_i}}{(\mu x.C^{\mathfrak{c}}_{\mathfrak{v}})(\overrightarrow{w_i}) : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-MUL})$$

$$\frac{\begin{array}{c}\Gamma, x : \neg(\overrightarrow{A_j}) \vdash v : \neg(\overrightarrow{A_j}) \qquad \Gamma \vdash (w_1, \ldots, w_{i-1}) : (A_1, \ldots, A_{i-1}) \\ C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash A_i) \qquad \Gamma \vdash (w_{i+1}, \ldots, w_n) : (A_{i+1}, \ldots, A_n)\end{array}}{(\mu x.v)(w_1, \ldots, w_{i-1}, C^{\mathfrak{c}}_{\mathfrak{v}}, w_{i+1}, \ldots, w_n) : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-MUR}_i)$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma, x : \mathtt{nat} \vdash)}{\sigma(v, x.C^{\mathfrak{c}}_{\mathfrak{c}}) : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-OP})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash) \qquad \Gamma, x : \mathtt{nat} \vdash t}{\mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow C^{\mathfrak{c}}_{\mathfrak{c}},\ \mathtt{succ}(x) \Rightarrow t\} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-CASEL})$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma \vdash s \qquad C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma, x : \mathtt{nat} \vdash)}{\mathtt{case}\ v\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow C^{\mathfrak{c}}_{\mathfrak{c}}\} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)}(\text{CC-CASER})$$

$$\frac{C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma, \overrightarrow{x_i : A_i} \vdash)}{\lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash \neg(\overrightarrow{A_i}))}(\text{CV-LBD})$$

Figure 7.2: Typing rules for contexts that accept a computation.

2a. If $C^{\mathfrak{c}}_{\mathfrak{v}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash A)$ *and* $C'^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma'' \vdash B) \Rightarrow (\Gamma' \vdash)$ *then*

$$C^{\mathfrak{c}}_{\mathfrak{v}}[C'^{\mathfrak{v}}_{\mathfrak{c}}] : (\Gamma'' \vdash B) \Rightarrow (\Gamma \vdash A).$$

And the analogous statements for all valid combinations of contexts C *and* C'*.*

Proof. In both cases proceed by induction on the typing derivation of the context C.

1. The two base cases (VV-ID) and (CC-ID) follow by assumption. All the other cases are solved by applying the induction hypothesis then the appropriate typing rule for terms.

2. Again, the base cases follow by assumption. In the other cases apply the induction hypothesis then apply the context typing rule that matches C.

□

Contextual equivalence checks whether two terms have the same behaviour in all program contexts that yield closed computations. Only these contexts are used because we can only observe the behaviour of closed computations. The set of observations $\mathfrak{P}$ encodes the observable behaviour of programs, so contextual equivalence makes use of it.

Definition 7.3.4. *Contextual preorder and equivalence are well-typed relations on possibly open terms defined as follows:*

1. *Given values* $\Gamma \vdash v, u : A$*, they are in the contextual preorder,* $\Gamma \vdash v \ (\leq_{ctx})^{\mathfrak{v}}_{A} \ u$*, if and only if:*

$$\forall C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma \vdash A) \Rightarrow (\emptyset \vdash).\ \forall P \in \mathfrak{P}.\ [\![C^{\mathfrak{v}}_{\mathfrak{c}}[v]]\!] \in P \implies [\![C^{\mathfrak{v}}_{\mathfrak{c}}[u]]\!] \in P.$$

2. *Given two computations* $\Gamma \vdash s, t$*, they are in the contextual preorder,* $\Gamma \vdash s \ (\leq_{ctx})^{\mathfrak{c}} \ t$*, if and only if:*

$$\forall C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma \vdash) \Rightarrow (\emptyset \vdash).\ \forall P \in \mathfrak{P}.\ [\![C^{\mathfrak{c}}_{\mathfrak{c}}[s]]\!] \in P \implies [\![C^{\mathfrak{c}}_{\mathfrak{c}}[t]]\!] \in P.$$

Two values v *and* u *are contextually equivalent,* $\Gamma \vdash v \ (=_{ctx})^{\mathfrak{v}}_{A} \ u$*, if* $\Gamma \vdash v \ (\leq_{ctx})^{\mathfrak{v}}_{A} \ u$ *and* $\Gamma \vdash u \ (\leq_{ctx})^{\mathfrak{v}}_{A} \ v$*. And similarly for computations. Therefore:*

$$(=_{ctx}) = (\leq_{ctx}) \cap (\leq_{ctx})^{op}.$$

Next, we prove that the definition above coincides with the coinductive definition of contextual equivalence from the previous section (Definition 7.1.4). The proofs of the following two lemmas can be found in Appendix D.

Lemma 7.3.5. *Contextual preorder defined with contexts,* $\leq_{ctx}$*, is a compatible and adequate preorder. Hence, it is included in contextual preorder defined coinductively,* $\sqsubseteq_{ctx}$*.*

Lemma 7.3.6. *Contextual preorder defined coinductively,* $\sqsubseteq_{ctx}$*, is closed under program contexts, that is:*

1. *If* $\Gamma' \vdash v \ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A} \ u$ *and* $C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash A) \Rightarrow (\Gamma \vdash B)$ *then* $\Gamma \vdash C^{\mathfrak{v}}_{\mathfrak{v}}[v] \ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{B} \ C^{\mathfrak{v}}_{\mathfrak{v}}[u]$*.*

   And the analogous statement for $C^{\mathfrak{v}}_{\mathfrak{c}}$*.*

2. *If* $\Gamma' \vdash s\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ t$ *and* $C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)$ *then* $\Gamma \vdash C^{\mathfrak{c}}_{\mathfrak{c}}[s]\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ C^{\mathfrak{c}}_{\mathfrak{c}}[t]$.

   And the analogous statement for $C^{\mathfrak{c}}_{\mathfrak{v}}$.

Theorem 7.3.7. *Contextual preorder defined using program contexts,* $\leq_{ctx}$*, coincides with contextual preorder defined coinductively,* $\sqsubseteq_{ctx}$*. Moreover,* $(=_{ctx}) = (\equiv_{ctx})$.

Proof. We have already shown in Lemma 7.3.5 that $(\leq_{ctx}) \subseteq (\sqsubseteq_{ctx})$. So it only remains to show the inverse inclusion.

Consider $\Gamma \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A}\ u$. Then by Lemma 7.3.6 we know that:

$$\forall C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma \vdash A) \Rightarrow (\emptyset \vdash).\ \emptyset \vdash C^{\mathfrak{v}}_{\mathfrak{c}}[v]\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ C^{\mathfrak{v}}_{\mathfrak{c}}[u].$$

By adequacy of $\sqsubseteq_{ctx}$ we can deduce that:

$$\forall C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma \vdash A) \Rightarrow (\emptyset \vdash).\ \forall P \in \mathfrak{P}.\ [\![C^{\mathfrak{v}}_{\mathfrak{c}}[v]]\!] \in P \implies [\![C^{\mathfrak{v}}_{\mathfrak{c}}[u]]\!] \in P$$

which is equivalent to:

$$\Gamma \vdash v\ (\leq_{ctx})^{\mathfrak{v}}_{A}\ u.$$

A similar reasoning can be applied for computations. So $(\sqsubseteq_{ctx}) \subseteq (\leq_{ctx})$ and hence $(\sqsubseteq_{ctx}) = (\leq_{ctx})$.

We know from Proposition 7.1.5 that $(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$. By definition of $=_{ctx}$ we know $(=_{ctx}) = (\leq_{ctx}) \cap (\leq_{ctx})^{op}$. So since $(\sqsubseteq_{ctx})^{op} = (\leq_{ctx})^{op}$ as well, we know that:

$$(=_{ctx}) = (\equiv_{ctx}).$$

□

7.4 Chapter Summary

Section 7.1 defined contextual equivalence for ECPS coinductively, as the greatest compatible relation that is adequate. Adequacy means that related computations are part of exactly the same observations from $\mathfrak{P}$. Thus, contextual equivalence is compatible by construction.

We then identified a condition on $\mathfrak{P}$ named *consistency* (Definition 7.2.1), which requires contextual equivalence to distinguish between at least two computations. This is a reasonable condition; otherwise, all computations would be identified with the divergent one.

Given a decomposable set $\mathfrak{P}$ of Scott-open observations which is consistent, we proved that contextual equivalence coincides with applicative bisimilarity (Theorem 7.2.2). The proof proceeds by showing the equality of the two relations for closed terms, and then extends this to open terms. Consistency is used in the proof to construct ECPS computations which distinguish between two different natural numbers. The construction of these terms required some ingenuity.

Theorems 7.2.2 and 6.2.3 allowed us to deduce the main result of the dissertation: that under the assumptions of Scott-openness, decomposability and consistency of $\mathfrak{P}$, $\mathcal{F}$-logical equivalence characterises contextual equivalence (Corollary 7.2.3).

The coinductive definition of contextual equivalence (Definition 7.1.4) might seem unintuitive, so we also presented the standard definition (Definition 7.3.4). This says that programs are related if they have the same *observable* behaviour in all program contexts. In ECPS the observable behaviour is encoded by the set $\mathfrak{P}$. So two programs are contextually equivalent if, when substituted in an arbitrary context yielding closed computations, the resulting computations are part of the same observations $P \in \mathfrak{P}$.

This definition of contextual equivalence is difficult to work with because of the universal quantification over a large number of contexts. Therefore, we preferred working with the coinductive definition in order to establish the most important results in this chapter.

Chapter 8

Conclusion

This chapter summarises the motivation and outcomes of the dissertation. It provides a comparison with previous work and outlines two directions for further research. The chapter concludes with an assessment of my personal development throughout the project.

8.1 Summary

Program equivalence, establishing when two programs are interchangeable, is one of the most important problems in the theory of programming languages. As explained in the introduction, the case of higher-order functions is especially difficult. Many definitions of program equivalence for increasingly complex higher-order languages have been proposed over the years. This variety of approaches has led researchers to investigate the relationships between different definitions of program equivalence.

This dissertation studied program equivalence for a higher-order language with algebraic effects. Algebraic effects are an approach to giving uniform formal semantics to impure operations such as input and output, probabilistic choice, nondeterminism etc. not usually found in purely functional languages. This method was discovered relatively recently [PP01, PP02, PP03, HPP06] so work still needs to be done to understand program equivalence in the presence of generic algebraic effects.

One open question we identified was: can we formulate a logic of program properties that characterises contextual equivalence for a higher-order language with generic algebraic effects? The language we chose to study is named ECPS (Section 3.1). It is a call-by-value extension of the simply-typed λ-calculus with algebraic effects, natural numbers and general recursion. Moreover, ECPS is a continuation-passing language in which programs are not expected to return.

We answered the above question positively in the context of ECPS by developing the logic $\mathcal{F}$, defined in Chapter 6. Thus, we obtained the first logic whose induced program equivalence coincides with contextual equivalence for algebraic effects (Corollary 7.2.3).

The starting point of our work was a paper by Simpson and Voorneveld [SV18]. They proposed a modal logic that characterises applicative bisimilarity for the EPCF language, but not contextual equivalence. EPCF (Section 2.4) is a variant of ECPS in which programs are written in direct style.

To justify the use of ECPS and to link our work to previous research, we first inves-

tigated the relationship between ECPS and EPCF, in Chapter 4. We proved that EPCF can be embedded into ECPS via a continuation-passing translation which preserves computation trees (Theorem 4.3.1). Then we argued informally that ECPS is more expressive than EPCF.

The remaining chapters were concerned with three forms of program equivalence for ECPS and the relationship between them: applicative bisimilarity, logical equivalence and contextual equivalence. In Chapter 5, we defined applicative bisimilarity for ECPS and proved it compatible (Theorem 5.3.7). Compatibility is a fundamental property that a meaningful program equivalence should satisfy. It says that related programs can be substituted for a variable on opposite sides of a program equation.

Chapter 6 defined the logic $\mathcal{F}$, which expresses properties of ECPS programs. We proved that $\mathcal{F}$-logical equivalence coincides with applicative bisimilarity, and is therefore compatible (Theorem 6.2.3). In Chapter 7, we developed contextual equivalence and proved it compatible. Then, we showed that applicative bisimilarity coincides with contextual equivalence (Theorem 7.2.2). This helped us prove the main result of the dissertation: that $\mathcal{F}$-logical equivalence coincides with contextual equivalence (Corollary 7.2.3).

8.2 Comparison with Previous Work

Other logics of program properties for effects have been proposed. For example, the *evaluation logic* of Pitts [Pit91] concerns general computational effects. It uses two built-in modalities $\Box$ and $\Diamond$ to talk about evaluation of programs. Observations $P \in \mathfrak{P}$ from the logic $\mathcal{F}$ play a similar role to these modalities. The difference is that the contents of the set of observations $\mathfrak{P}$ depend on the effects present in the language, rather than being built into the logic. Moreover, program equivalence induced by evaluation logic is not compared with other operational notions of program equivalence.

Plotkin and Pretnar [PP08] propose a *logic for algebraic effects* which is shown to be sound for establishing different kinds of program equivalence, but not complete in general. According to Pnueli's classification [Pnu77], this logic for algebraic effects is an exogenous logic because computations are allowed to appear inside formulas. In contrast, the logic $\mathcal{F}$ is endogenous because a formula concerns only one computation.

Another difference is that in Plotkin's and Pretnar's logic there is a modality for each algebraic operation. In the logic $\mathcal{F}$, we instead adopt the view that each observation should express a behavioural property of programs. Thus, although observations depend on the effects in the language, they do not depend on the syntax of these effects. So we obtain a greater separation between the logic and the syntax of the programming language.

Simpson's and Voorneveld's work [SV18] is the most closely related to ours. They propose the EPCF logic (Section 2.6), in which modalities specify shapes of computation trees, and show it characterises applicative bisimilarity. Therefore, modalities play a similar role to the observations from $\mathcal{F}$.

The difference is that EPCF formulas, and applicative bisimilarity, check for the return values of programs. Because ECPS is a continuation-passing language, programs do not return, so neither $\mathcal{F}$-formulas nor applicative bisimilarity can check return values. As explained in Chapter 4, EPCF is in fact a fragment of ECPS. Therefore, contextual equivalence in ECPS is more restrictive since there are more program contexts. These

differences provide an insight into why in ECPS we were able to obtain the relationship:

$$\text{applicative bisimilarity} = \mathcal{F}\text{-logical equivalence} = \text{contextual equivalence.}$$

Whereas in EPCF the situation is:

$$\text{applicative bisimilarity} = \textit{EPCF}\text{-logical equivalence} \neq \text{contextual equivalence.}$$

Compared to modal logics used to specify program properties in practice, such as LTL or CTL, the logic $\mathcal{F}$ is not directly suitable for verification because of its infinitary connectives. However, $\mathcal{F}$ achieves its goal of expressing behavioural properties of higher-order programs with algebraic effects. As a result, it can be used as the starting point for designing higher-level logics for algebraic effects that would be suitable for verification.

8.3 Future Work

The examples of effects considered in this work were nondeterminism, probabilistic choice, global store and I/O. A next step would be to consider local store, which is also an algebraic effect [PP02]. Relevant questions here are integrating local store into the framework for generic algebraic effects provided by the logic $\mathcal{F}$, and investigating whether the current relationship between different forms of program equivalence still holds.

Operational notions of program equivalence for higher-order languages with local store have been studied by Pitts and Stark [PS98], who obtained a characterisation of contextual equivalence using a logical relation. Yoshida, Honda and Berger [YHB08] propose an extension of Hoare logic to reason about local store. They prove that this logic characterises contextual equivalence. However, local store remains a challenging effect. We anticipate that the notion of computation tree, on which observations from $\mathcal{F}$ are based, would need to be changed to account for local store.

Finally, one could study a fourth notion of program equivalence for ECPS called *normal-form bisimilarity* [LL07]. This has not been extended to algebraic effects before. One difference between normal-form bisimilarity and applicative bisimilarity is that the latter only relates closed terms, whereas the former also considers open terms.

The definition of normal-form bisimulation is related to game semantics [AM99], which models open programs as strategies in a two-player game. Similarly, the operational semantics of ECPS can be extended to open terms. A satisfaction relation between open terms and logical formulas can be defined based on this operational semantics and the observations in $\mathfrak{P}$. The question here is how does normal-form bisimilarity compare to the program equivalence induced by game satisfaction? and furthermore, how do they both compare to applicative bisimilarity? We conjecture that normal-form bisimilarity is closer to game satisfaction than applicative bisimilarity, but leave these questions for future investigation.

8.4 Personal Reflections

From a technical perspective the project presented many challenges. I learnt about algebraic effects and became familiar with proof techniques such as logical relations, coinduction and Howe's method. I completed a significant number or fairly long inductive proofs which required care and organisation.

The proof of correctness of the CPS translation (Theorem 4.3.1) was more difficult than expected. Two approaches I tried initially failed: an inductive proof using domain theoretic techniques and a coinductive proof. I finally settled on the combination between logical relations and coinduction which proved successful.

The *Categories, Proofs and Processes* course [AT18] I took this year helped me understand some of the abstract material needed for the project, such as coinduction, while the *Principles of Programming Languages* lectures I attended provided useful background on continuations. Other useful courses were *Formal Verification* and *Automata, Logic and Games*, where I learnt about modal logics used in verification to express program properties, such as LTL, CTL and the modal μ-calculus.

Overall, my ability to understand new theoretical material and to carry out complex proofs has improved. Considering this and the novel theoretical contribution that the dissertation makes, I can say that the project has been a success.

Bibliography

[Abr90] S. Abramsky. The lazy λ-calculus. In D.A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.

[Ahm06] A.J. Ahmed. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP*, 2006.

[AM99] S. Abramsky and G. McCusker. Game Semantics. In *Computational Logic*. Springer Berlin Heidelberg, 1999.

[AT18] S. Abramsky and N. Tzevelekos. Introduction to Categories and Categorical Logic. Lecture notes for the MSc in Computer Science, University of Oxford, 2017/18.

[BH09] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[CH07] K. Crary and R. Harper. Syntactic Logical Relations for Polymorphic and Recursive Types. *Electr. Notes Theor. Comput. Sci.*, 172:259–299, 2007.

[CL14] R. Crubillé and U. Dal Lago. On Probabilistic Applicative Bisimulation and Call-by-Value λ-Calculi. In *ESOP*, 2014.

[Fio17] M. Fiore. Denotational Semantics. Lecture notes for Part II of the Computer Science Tripos, University of Cambridge, 2016/17.

[Gor98] A.D. Gordon. Operational Equivalences for Untyped and Polymorphic Object Calculi. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 9–54. Cambridge University Press, 1998.

[Gri90] T. Griffin. A Formulae-as-Types Notion of Control. In *POPL*, 1990.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32(1):137–161, 1985.

[How96] D.J. Howe. Proving Congruence of Bisimulation in Functional Programming Languages. *Inf. Comput.*, 124(2):103–112, 1996.

[HPP06] M. Hyland, G.D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

[Jac16] B. Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016.

[JR11] B. Jacobs and J. Rutten. An Introduction to (co)algbebra and (co)induction. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, chapter 2, pages 38–99. Cambridge University Press, 2011.

[JSV10] P. Johann, A. Simpson, and J. Voigtländer. A Generic Operational Metatheory for Algebraic Effects. In *LICS*, 2010.

[JT11] G. Jaber and N. Tabareau. The Journey of Biorthogonal Logical Relations to the Realm of Assembly Code. In *LOLA*, 2011.

[Koz83] D. Kozen. Results on the Propositional μ-Calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[Las98] S.B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, University of Aarhus, BRICS, December 1998.

[Lev06] P.B. Levy. Infinitary Howe's Method. *Electr. Notes Theor. Comput. Sci.*, 164(1):85–104, 2006.

[LGL17a] U. Dal Lago, F. Gavazzo, and P.B. Levy. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *LICS*, 2017.

[LGL17b] U. Dal Lago, F. Gavazzo, and P.B. Levy. Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method (Long Version). *CoRR*, abs/1704.04647, 2017.

[LL07] S.B. Lassen and P.B. Levy. Typed Normal Form Bisimulation. In *CSL*, 2007.

[LPT03] P.B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182 – 210, 2003.

[LRS93] Y. Lafont, B. Reus, and T. Streicher. Continuations Semantics or Expressing Implication by Negation. Technical Report 9321, Ludwig-Maximilians-Universität, München, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, Heildelberg, 1980.

[Mog91] E. Moggi. Notions of Computation and Monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mor69] J. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1969.

[Pit91] A.M. Pitts. Evaluation Logic. In *IVth Higher Order Workshop*, 1991.

[Pit96] A.M. Pitts. Relational Properties of Domains. *Inf. Comput.*, 127(2):66–90, 1996.

[Pit10] A.M. Pitts. Step-Indexed Biorthogonality: a Tutorial Example. In *Modelling, Controlling and Reasoning About State*, Dagstuhl Seminar Proceedings, 2010.

[Pit11] A.M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, chapter 5, pages 197–232. Cambridge University Press, 2011.

[Plo77] G.D. Plotkin. LCF Considered as a Programming Language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[PP01] G.D. Plotkin and J. Power. Adequacy for Algebraic Effects. In *FOSSACS*, 2001.

[PP02] G.D. Plotkin and J. Power. Notions of Computation Determine Monads. In *FOSSACS*, 2002.

[PP03] G.D. Plotkin and J. Power. Algebraic Operations and Generic Effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.

[PP08] G.D. Plotkin and M. Pretnar. A Logic for Algebraic Effects. In *LICS*, 2008.

[PS98] A.M. Pitts and I.D.B. Stark. Operational reasoning for functions with local state. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[Rey93] J.C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[RT99] J.G. Riecke and H. Thielecke. Typed Exeptions and Continuations Cannot Macro-Express Each Other. In *ICALP*, 1999.

[SU06] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Inc., 2006.

[SV17] A. Simpson and N. Voorneveld. Behavioural equivalence via modalities for algebraic effects, 2017. Unpublished manuscript.

[SV18] A. Simpson and N. Voorneveld. Behavioural equivalence via modalities for algebraic effects. In *ESOP*, 2018.

[Tai67] W.W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967.

[YHB08] N. Yoshida, K. Honda, and M. Berger. Logical Reasoning for Higher-Order Functions with Local State. *Logical Methods in Computer Science*, 4(4), 2008.

Appendix A

Proofs about the CPS translation

Lemma 4.3.6. *If* $(S, M) \rightarrowtail^k (S', M')$ *and* $t \longrightarrow^* t'$ *where* $k < n$ *then:*

$$(S, M)\ \mathcal{S}^n_{\tau,\rho}\ t \iff (S', M')\ \mathcal{S}^{n-k}_{\tau,\rho}\ t'.$$

Proof. Assume $(S, M)\ \mathcal{S}^n_{\tau,\rho}\ t$. To show $(S', M')\ \mathcal{S}^{n-k}_{\tau,\rho}\ t'$ it suffices to show that the two conditions in Definition 4.3.4 are satisfied for $((S', M'), t')$.

Assume that for some $p < n - k$, $(S', M') \longrightarrow^p (S'', \sigma(V; W))$. Then we know $(S, M) \longrightarrow^{p+k} (S'', \sigma(V; W))$ and $p + k < n$. So we can deduce $t \longrightarrow^* t' \longrightarrow^* \sigma(v, x.t'')$ where $V = \bar{l}$ and $v = \bar{l}$, and $\forall l \in \mathbb{N}.\ (S'', W\ \bar{l})\ \mathcal{S}^{n-k-p}_{\tau'',\rho}\ t''[\bar{l}/x]$, as required.

Assume that for some $p \leq n - k$, $(S', M') \longrightarrow^p (id, \mathbf{return}\ V)$. Then $(S, M) \longrightarrow^{p+k} (id, \mathbf{return}\ V)$. So from the initial assumption we know $t \longrightarrow^* t' \longrightarrow^* \downarrow$, as required.

The reverse implication is proved similarly. □

Lemma 4.3.8. *For any configuration* (S, M) *and closed computation* t*:*

$$(\forall n \in \mathbb{N}.\ (S, M)\ \mathcal{S}^n_{\tau,\rho}\ t) \implies (S, M)\ \mathcal{S}_{\tau,\rho}\ t.$$

Proof. Assume $\forall n \in \mathbb{N}.\ (S, M)\ \mathcal{S}^n_{\tau,\rho}\ t$.

To prove $(S, M)\ \mathcal{S}_{\tau,\rho}\ t$, it suffices to find a simulation relation respecting the conditions in Definition 4.3.7 which contains the pair $((S, M), t)$. Because $\mathcal{S}$ is the greatest simulation, this relation will be contained in $\mathcal{S}$.

Let $\bigcap_{n\in\mathbb{N}} \mathcal{S}^n$ be the candidate simulation. Check the condititions in Definition 4.3.7:

1. Assume $(S, M) \rightarrowtail^* (S', \sigma(V; W))$. Then there exists $p \in \mathbb{N}$ such that $(S, M) \rightarrowtail^p (S', \sigma(V; W))$. By assumption we know $\forall m > 0.\ (S, M)\ \mathcal{S}^{p+m}_{\tau,\rho}\ t$. So by the first condition in Definition 4.3.4 we know that $t \longrightarrow^* \sigma(v, x.t')$ where $V = \bar{l}$ and $v = \bar{l}$, and that $\forall l \in \mathbb{N}.\ (S', W\ \bar{l})\ \bigcap_{m>0} \mathcal{S}^m_{\tau',\rho}\ t'[\bar{l}/x]$. For $m = 0$, $(S', W\ \bar{l})\ \mathcal{S}^0_{\tau',\rho}\ t'[\bar{l}/x]$ can be easily seen to hold. So we have $\forall l \in \mathbb{N}.\ (S', W\ \bar{l})\ \bigcap_{m\in\mathbb{N}} \mathcal{S}^m_{\tau',\rho}\ t'[\bar{l}/x]$ as required.

2. Assume $(S, M) \rightarrowtail^p (id, \mathbf{return}\ V)$ for some $p \in \mathbb{N}$. Then from $(S, M)\ \mathcal{S}^p_{\tau,\rho}\ t$ we have $t \longrightarrow^* \downarrow$ as required.

□

Lemma 4.3.10 (Fundamental property of the logical relation)**.** *For any value* $\overrightarrow{x_i : \tau_i} \vdash V : \rho$*, any computation* $\overrightarrow{x_i : \tau_i} \vdash M : \rho$ *and any stack* $\overrightarrow{x_i : \tau_i} \vdash S : \rho \Rightarrow \rho'$ *in EPCF:*

1. $\overrightarrow{x_i : \tau_i} \vdash V \ \mathcal{R}^{\mathfrak{v}}_{\rho} \ V^*$.
2. $\overrightarrow{x_i : \tau_i} \vdash M \ \mathcal{R}^{\mathfrak{c}}_{\rho} \ M^*$.
3. $\overrightarrow{x_i : \tau_i} \vdash S \ \mathcal{R}^{\mathfrak{s}}_{\rho,\rho'} \ S^*$.

Proof. The proof is by induction on the typing derivations of V, M and S. All cases are presented below, starting with **fix** which is the most interesting.

Case (fix), $M = \mathbf{fix}\ W$**.** Assume $\overrightarrow{x_i : \tau_i} \vdash W \ \mathcal{R}^{\mathfrak{v}}_{(\rho_1 \to \rho_2) \to (\rho_1 \to \rho_2)} \ W^*$. We need to prove:

$$\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i, v_i) : \tau_i}.\ (\overrightarrow{(V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\tau_i}} \implies (\mathbf{fix}\ W[\overrightarrow{V_i/x_i}], (\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},n}_{\rho_1 \to \rho_2}).$$

This is proved by induction on n.

Base case $n = 0$**.** To prove $(\mathbf{fix}\ W[\overrightarrow{V_i/x_i}], (\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},0}_{\rho_1 \to \rho_2}$ we need to check that:

$$\forall (S, k) \in \mathcal{R}^{\mathfrak{s},0}_{\rho_1 \to \rho_2, \rho}.\ (S, \mathbf{fix}\ W[\overrightarrow{V_i/x_i}])\ \mathcal{S}^0_{\rho_1 \to \rho_2, \rho}\ ((\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]\ k).$$

The first condition in the definition of step-indexed similarity holds because there is no $p < 0$. The second condition holds because $(S, \mathbf{fix}\ W[\overrightarrow{V_i/x_i}]) \longrightarrow^0 (id, \mathbf{return}\ V)$ is false.

Induction step $n > 0$**.** The induction hypothesis is:

$$\forall m < n.\ \forall \overrightarrow{(V_i, v_i) : \tau_i}.\ (\overrightarrow{(V_i, v_i) \in \mathcal{R}^{\mathfrak{v},m}_{\tau_i}} \implies (\mathbf{fix}\ W[\overrightarrow{V_i/x_i}], (\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},m}_{\rho_1 \to \rho_2}).$$

Consider some arbitrary $\overrightarrow{(V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\tau_i}}$. By the definition of a step-indexed relation we know $\forall m < n.\ \mathcal{R}^{\mathfrak{v},n}_{\tau_i} \subseteq \mathcal{R}^{\mathfrak{v},m}_{\tau_i}$.

We need to prove that:

$$\forall p \leq n.\ \forall (S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\rho_1 \to \rho_2, \rho}.\ (S, \mathbf{fix}\ W[\overrightarrow{V_i/x_i}])\ \mathcal{S}^p_{\rho_1 \to \rho_2, \rho}\ ((\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]\ k).$$

For $p < n$ this follows immediately from the induction hypothesis if we choose $m = n - 1$.

For the case $p = n$ let:

$$E = \lambda y{:}\rho_1.\mathbf{let}\ \mathbf{fix}\ W[\overrightarrow{V_i/x_i}] \Rightarrow z\ \mathbf{in}\ z\ y$$

$$F = \lambda x{:}\rho_1.\mathbf{let}\ W[\overrightarrow{V_i/x_i}]\ E \Rightarrow w\ \mathbf{in}\ w\ x.$$

From the operational semantics and the CPS translation we can see that:

$$(S, \mathbf{fix}\ W[\overrightarrow{V_i/x_i}]) \rightarrowtail (S, \mathbf{return}\ F)$$

$$(\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]\ k \longrightarrow^*\ k\ F^*.$$

Therefore, using Lemma 4.3.6 it suffices to prove:

$$(S, \mathbf{return}\ F)\ \mathcal{S}^{n-1}_{\rho_1 \to \rho_2, \rho}\ k\ F^*.$$

We have assumed $(S, k) \in \mathcal{R}^{\mathfrak{s},n}_{\rho_1 \to \rho_2, \rho}$ so by definition of $\mathcal{R}$ we know:

$$\forall (V, v) \in \mathcal{R}^{\mathfrak{v},n-1}_{\rho_1 \to \rho_2}.\ (S, \mathbf{return}\ V)\ \mathcal{S}^{n-1}_{\rho_1 \to \rho_2, \rho}\ k\ v.$$

So it is enough to show $(F, F^*) \in \mathcal{R}^{\mathfrak{v},n-1}_{\rho_1 \to \rho_2}$, that is:

$$\forall p_1 < n-1.\ \forall (V_1, v_1) \in \mathcal{R}^{\mathfrak{v},p_1}_{\rho_1}.\ \forall p_2 \leq p_1.\ \forall (S_2, k_2) \in \mathcal{R}^{\mathfrak{s},p_2}_{\rho_2, \rho'}.$$

$$(S_2, F\ V_1)\ \mathcal{S}^{p_2}_{\rho_2, \rho'}\ (\lambda k_1{:}\neg\rho_2^*.F^*\ (v_1, k_1))\ k_2.$$

From the operational semantics we can deduce:

$$(S_1, F\ V_1) \rightarrowtail^2 (S_2 \circ (\mathbf{let}\ (-) \Rightarrow w\ \mathbf{in}\ wV_1), W[\overrightarrow{V_i/x_i}]\ E)$$

$$(\lambda k_1{:}\neg\rho_2^*.F^*\ (v_1, k_1))\ k_2 \longrightarrow^* W^*[\overrightarrow{v_i/x_i}]\ (E^*, \lambda w{:}(\rho_1 \to \rho_2)^*.((\lambda l''{:}\neg\rho_2^*.w\ (v_1, l''))\ k_2)).$$

For $p_2 < 2$, we immediately have that $(S_2, F\ V_1)\ \mathcal{S}^{p_2}_{\rho_2, \rho'}\ (\lambda k_1{:}\neg\rho_2^*.F^*\ (v_1, k_1))\ k_2$ because in the definition of $\mathcal{S}^{p_2}$, the premises of both implications are false.

If $p_2 \geq 2$, then using Lemma 4.3.6 it suffices to show:

$$(S_2 \circ (\mathbf{let}\ (-) \Rightarrow w\ \mathbf{in}\ wV_1), W[\overrightarrow{V_i/x_i}]\ E)\ \mathcal{S}^{p_2-2}_{\rho_1 \to \rho_2, \rho''}$$

$$W^*[\overrightarrow{v_i/x_i}]\ (E^*, \lambda w{:}(\rho_1 \to \rho_2)^*.((\lambda l''{:}\neg\rho_2^*.w\ (v_1, l''))\ k_2)).$$

From the induction hypothesis for W we know that:

$$(W[\overrightarrow{V_i/x_i}], W^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},p_2-1}_{(\rho_1 \to \rho_2) \to (\rho_1 \to \rho_2)}. \tag{A.1}$$

We would like to use this to prove the previous statement. Unpacking the definition of $\mathcal{R}^{\mathfrak{v},p_2-1}_{(\rho_1 \to \rho_2) \to (\rho_1 \to \rho_2)}$ we see that if we prove:

1. $(E, E^*) \in \mathcal{R}^{\mathfrak{v},p_2-2}_{\rho_1 \to \rho_2}$.
2. $(S_2 \circ (\mathbf{let}\ (-) \Rightarrow w\ \mathbf{in}\ wV_1), \lambda w{:}(\rho_1 \to \rho_2)^*.((\lambda l''{:}\neg\rho_2^*.w\ (v_1, l''))\ k_2)) \in \mathcal{R}^{\mathfrak{s},p_2-2}_{\rho_1 \to \rho_2, \rho''}$.

we can then use equation A.1 to obtain the desired result.

We first prove the second statement, which is equivalent to:

$$\forall p_3 \leq p_2 - 2.\ \forall (V_3, v_3) \in \mathcal{R}^{\mathfrak{v},p_3}_{\rho_1 \to \rho_2}.$$

$$(S_2 \circ (\mathbf{let}\ (-) \Rightarrow w\ \mathbf{in}\ wV_1), V_3)\ \mathcal{S}^{p_3}_{\rho_1 \to \rho_2, \rho''}\ (\lambda w{:}(\rho_1 \to \rho_2)^*.((\lambda l''{:}\neg\rho_2^*.w\ (v_1, l''))\ k_2))\ v_3.$$

From the operational semantics:

$$(S_2 \circ (\mathbf{let}\ (-) \Rightarrow w\ \mathbf{in}\ wV_1), V_3) \rightarrowtail (S_2, V_3\ V_1)$$

$$(\lambda w{:}(\rho_1 \to \rho_2)^*.((\lambda l''{:}\neg\rho_2^*.w\ (v_1, l''))\ k_2))\ v_3 \longrightarrow^* v_3\ (v_1, k_2).$$

Therefore, it is enough to show:

$$(S_2, V_3\ V_1)\ \mathcal{S}^{p_3-1}_{\rho_2, \rho''}\ v_3\ (v_1, k_2). \tag{A.2}$$

We have assumed $(V_3, v_3) \in \mathcal{R}^{\mathfrak{v},p_3}_{\rho_1\to\rho_2}$ and $(V_1, v_1) \in \mathcal{R}^{\mathfrak{v},p_1}_{\rho_1} \subseteq \mathcal{R}^{\mathfrak{v},p_3-1}_{\rho_1}$, so by definition of $\mathcal{R}^{\mathfrak{v}}$ we know:

$$(V_3\ V_1, \lambda k{:}\neg\rho_2^*.v_3\ (v_1, k)) \in \mathcal{R}^{\mathfrak{c},p_3-1}_{\rho_2}.$$

We have also assumed $(S_2, k_2) \in \mathcal{R}^{\mathfrak{s},p_2}_{\rho_2,\rho'} \subseteq \mathcal{R}^{\mathfrak{s},p_3-1}_{\rho_2,\rho'}$ so by definition of $\mathcal{R}^{\mathfrak{c}}$ we know:

$$(S_2, V_3\ V_1)\ \mathcal{S}^{p_3-1}_{\rho_2,\rho''}\ (\lambda k{:}\neg\rho_2^*.v_3\ (v_1, k))\ k_2.$$

So from Lemma 4.3.6 we obtain the desired result, equation A.2.

Now prove the first statement, 1: this is equivalent to proving

$$\forall p_4 < p_2 - 2.\ \forall (V_4, v_4) \in \mathcal{R}^{\mathfrak{v},p_4}_{\rho_1}.\ \forall p_5 \le p_4.\ \forall (S_5, k_5) \in \mathcal{R}^{\mathfrak{s},p_5}_{\rho_2,\sigma}.$$
$$(S_5, E\ V_4)\ \mathcal{S}^{p_5}_{\rho_2,\sigma}\ (\lambda k{:}\neg\rho_2^*.E^*\ (v_4, k))\ k_5.$$

Using the operational semantics we have:

$$(S_5, E\ V_4) \rightarrowtail^2 (S_5 \circ (\mathbf{let}\ (-) \Rightarrow z\ \mathbf{in}\ zV_4), \mathbf{fix}\ W[\overrightarrow{V_i/x_i}])$$

$$(\lambda k{:}\neg\rho_2^*.E^*\ (v_4, k))\ k_5 \longrightarrow^* (\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]\ \lambda z{:}(\rho_1\to\rho_2)^*.((\lambda p'{:}\neg\rho_2^*.z\ (v_4, p'))\ k_5).$$

So it is enough to show:

$$(S_5 \circ (\mathbf{let}\ (-) \Rightarrow z\ \mathbf{in}\ zV_4), \mathbf{fix}\ W[\overrightarrow{V_i/x_i}])\ \mathcal{S}^{p_5-2}_{\rho_1\to\rho_2,\sigma}$$
$$(\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]\ \lambda z{:}(\rho_1\to\rho_2)^*.((\lambda p'{:}\neg\rho_2^*.z\ (v_4, p'))\ k_5). \quad \text{(A.3)}$$

From the induction hypothesis on natural numbers we know that:

$$(\mathbf{fix}\ W[\overrightarrow{V_i/x_i}], (\mathbf{fix}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},p_5-2}_{\rho_1\to\rho_2}.$$

Similarly to the proof for S_2 and k_2, statement 2, we can show that:

$$(S_5 \circ (\mathbf{let}\ (-) \Rightarrow z\ \mathbf{in}\ zV_4), \lambda z{:}(\rho_1\to\rho_2)^*.((\lambda p'{:}\neg\rho_2^*.z\ (v_4, p'))\ k_5)) \in \mathcal{R}^{\mathfrak{s},p_5-2}_{\rho_1\to\rho_2,\sigma}.$$

Combining the last two observations and using the definition of $\mathcal{R}^{\mathfrak{c}}$ we obtain the desired result, equation A.3.

Case (op), $M = \sigma(V; W)$ for some $\sigma \in \Sigma$. The induction hypothesis is:

$$\overrightarrow{x_i : \sigma_i} \vdash V\ \mathcal{R}^{\mathfrak{v}}_{\mathbb{N}}\ V^* \quad \text{and} \quad \overrightarrow{x_i : \sigma_i} \vdash W\ \mathcal{R}^{\mathfrak{v}}_{\mathbb{N}\to\tau}\ W^*.$$

We need to prove;

$$\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i, v_i) : \sigma_i}.\ (\forall i.\ (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies$$
$$(\sigma(V; W)[\overrightarrow{V_i/x_i}], \lambda k{:}\neg\tau^*.\sigma(V^*; x.W^*(x, k))[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},n}_{\tau}.$$

Consider $p \le n$ and $(S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau,\rho}$. It suffices to prove:

$$(S, \sigma(V; W)[\overrightarrow{V_i/x_i}])\ \mathcal{S}^{p}_{\tau,\rho}\ (\lambda k{:}\neg\tau^*.\sigma(V^*; x.W^*(x, k))[\overrightarrow{v_i/x_i}])\ k.$$

We prove this by checking the conditions in the definition of $\mathcal{S}^p$. If $p = 0$ the first condition is satisfied immeditaly because of the strict inequality. If $p > 0$ we can choose $p' = 0$ such that $(S, \sigma(V; W)[\overrightarrow{V_i/x_i}]) \rightarrowtail^{p'} (S, \sigma(\overrightarrow{V; W)[V_i/x_i}])$. We see that $(\lambda k{:}\neg\tau^*.\sigma(V^*; x.W^*(x,k))[\overrightarrow{v_i/x_i}])\ k \longrightarrow^* \sigma(\overrightarrow{V^*; x.W^*(x,k))[v_i/x_i}]$ as required. By the induction hypothesis for V we know that $V[\overrightarrow{V_i/x_i}] = \overline{m}$ and $V^*[\overrightarrow{v_i/x_i}] = \overline{m}$ for some $m \in \mathbb{N}$. It remains to show that:

$$\forall l \in \mathbb{N}.\ (S, W[\overrightarrow{V_i/x_i}]\ \bar{l})\ \mathcal{S}^p_{\tau,\rho}\ W^*[\overrightarrow{v_i/x_i}]\ (\bar{l}, k).$$

This follows from the induction hypothesis for W, using Lemma 4.3.6.

The second condition in the definition of $\mathcal{S}^p$ is satisfied because the premise of the implication is false. $(S, \sigma(V; W)[\overrightarrow{V_i/x_i}])$ cannot reduce anymore according to the $\rightarrowtail$ relation, and is not of the form $(S, \mathbf{return}\ V')$.

Case (ret), $M = \mathbf{return}\ W$. From the induction hypothesis we know:

$$\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i, v_i) : \sigma_i}.\ (\forall i.\ (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies (W[\overrightarrow{V_i/x_i}], W^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},n}_{\tau}.$$

We need to prove that:

$$\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i, v_i) : \sigma_i}.\ (\forall i.\ (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies (\mathbf{return}\ W[\overrightarrow{V_i/x_i}], (\mathbf{return}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},n}_{\tau}.$$

So it suffices to show:

$$\forall n \in \mathbb{N}.\ (W[\overrightarrow{V_i/x_i}], W^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},n}_{\tau} \implies (\mathbf{return}\ W[\overrightarrow{V_i/x_i}], (\mathbf{return}\ W)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},n}_{\tau}.$$

Consider $p \leq n$ and $(S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau,\rho}$. We need to show that:

$$(S, \mathbf{return}\ W[\overrightarrow{V_i/x_i}])\ \mathcal{S}^p_{\tau,\rho}\ (\lambda k{:}\neg\tau^*.k\ W^*[\overrightarrow{v_i/x_i}])\ k.$$

Using Lemma 4.3.6 it is enough to show $(S, \mathbf{return}\ W[\overrightarrow{V_i/x_i}])\ \mathcal{S}^p_{\tau,\rho}\ k\ W^*[\overrightarrow{v_i/x_i}]$. This follows from $(S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau,\rho}$ and $(W[\overrightarrow{V_i/x_i}], W^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},n}_{\tau} \subseteq \mathcal{R}^{\mathfrak{v},p}_{\tau}$ by definition of $\mathcal{R}^{\mathfrak{s}}$.

Case (let), $M = \mathbf{let}\ N_1 \Rightarrow y\ \mathbf{in}\ N_2$. The induction hypothesis is:

$$\overrightarrow{x_i : \sigma_i} \vdash N_1\ \mathcal{R}^{\mathfrak{c}}_{\tau_1}\ N_1^* \quad \text{and} \quad \overrightarrow{x_i : \sigma_i}, y : \tau_1 \vdash N_2\ \mathcal{R}^{\mathfrak{c}}_{\tau_2}\ N_2^*.$$

We need to prove:

$$\forall n \in \mathbb{N}.\ \forall \overrightarrow{(V_i, v_i) : \sigma_i}.\ (\forall i.\ (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies (\mathbf{let}\ N_1[\overrightarrow{V_i/x_i}] \Rightarrow y\ \mathbf{in}\ N_2[\overrightarrow{V_i/x_i}], (\mathbf{let}\ N_1 \Rightarrow y\ \mathbf{in}\ N_2)^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{c},n}_{\tau_2}.$$

Consider $p \leq n$ and $(S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau_2,\rho}$. For $p \geq 1$ it is enough to show:

$$(S \circ \textbf{let } (-) \Rightarrow y \textbf{ in } N_2[\overrightarrow{V_i/x_i}], N_1)\ \mathcal{S}^{p-1}_{\tau_2,\rho}\ N_1^*[\overrightarrow{v_i/x_i}]\ (\lambda y{:}\tau_1^*.N_2^*[\overrightarrow{v_i/x_i}]\ k).$$

By induction hypothesis for N_1 it is enough to show:

$$(S \circ \textbf{let } (-) \Rightarrow y \textbf{ in } N_2[\overrightarrow{V_i/x_i}], \lambda y{:}\tau_1^*.N_2^*[\overrightarrow{v_i/x_i}]\ k) \in \mathcal{R}^{\mathfrak{s},p-1}_{\tau_1,\rho}.$$

Consider $p_1 \leq p$ and $(W_1, w_1) \in \mathcal{R}^{\mathfrak{v},p_1}_{\tau_1}$. For $p_1 \geq 1$ it is enough to show:

$$(S, N_2[\overrightarrow{V_i/x_i}, W_1/y])\ \mathcal{S}^{p_1-1}_{\tau_2,\rho}\ N_2^*[\overrightarrow{v_i/x_i}, w_1/y]\ k.$$

This follows from the induction hypothesis for N_2.

Case (app), $M = W\ U$**.** If W has type $\tau_1 \to \tau_2$ and U has type τ_1, for each $n \in \mathbb{N}$, we need to prove:

$$\forall p \leq n.\ \forall (S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau_2,\rho}.\ (S, (W\ U)[\overrightarrow{V_i/x_i}])\ \mathcal{S}^{p}_{\tau_2,\rho}\ (\lambda k{:}\neg\tau_2^*.W^*[\overrightarrow{v_i/x_i}]\ (U^*[\overrightarrow{v_i/x_i}], k))\ k.$$

This follows from $(W[\overrightarrow{V_i/x_i}], W^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},p+1}_{\tau_1\to\tau_2,\rho}$ and $(U[\overrightarrow{V_i/x_i}], U^*[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},p}_{\tau_1}$, which we know from the induction hypothesis.

Case (case), $M = \textbf{case } V \textbf{ in } \{Z \Rightarrow N_1,\ S(y) \Rightarrow N_2\}$**.** We need to prove for each $n \in \mathbb{N}$:

$$\forall p \leq n.\ \forall (S, k) \in \mathcal{R}^{\mathfrak{s},p}_{\tau,\rho}.\ (S,\ (\textbf{case } V \textbf{ in } \{Z \Rightarrow N_1,\ S(y) \Rightarrow N_2\})[\overrightarrow{V_i/x_i}])\ \mathcal{S}^{p}_{\tau,\rho}$$

$$(\lambda k{:}\neg\tau^*.\texttt{case } V^* \texttt{ in } \{\texttt{zero} \Rightarrow N_1^*\ k,\ \texttt{succ}(y) \Rightarrow N_2^*\ k\})[\overrightarrow{v_i/x_i}]\ k.$$

We proceed by a case split on whether $V[\overrightarrow{V_i/x_i}] = \overline{0}$ or not. Then the result follows from the induction hypothesis for N_1 or N_2, respectively.

Case (sid), $S = id$**.** In this case we need to prove:

$$\forall n \in \mathbb{N}.\ \forall p \leq n.\ \forall (V, v) \in \mathcal{R}^{\mathfrak{v},p}_{\rho}.\ (id, \textbf{return } V)\ \mathcal{S}^{p}_{\rho,\rho}\ (\lambda x{:}\neg\rho^*.\downarrow)\ v.$$

We show that the conditions in the definition of $\mathcal{S}^p$ hold. The first condition holds because $(id, \textbf{return } V) \not\rightarrowtail$ and $(id, \textbf{return } V)$ is not of the form $(S, \sigma(V; W))$. For the second condition we can choose $p' = 0 \leq p$ such that $(id, \textbf{return } V) \rightarrowtail^{p'} (id, \textbf{return } V)$ and we have $(\lambda x{:}\neg\rho^*.\downarrow)\ v \longrightarrow\downarrow$, as required.

Case (slet), $S = S' \circ \textbf{let } (-) \Rightarrow y \textbf{ in } M$**.** The induction hypothesis is:

$$\overrightarrow{x_i : \sigma_i} \vdash S'\ \mathcal{R}^{\mathfrak{s}}_{\tau_2,\rho}\ S'^* \quad \text{and} \quad \overrightarrow{x_i : \sigma_i}, y : \tau_1 \vdash M\ \mathcal{R}^{\mathfrak{c}}_{\tau_2}\ M^*.$$

We need to prove:

$$\forall n \in \mathbb{N}.\ \forall p \leq n.\ \forall (V, v) \in \mathcal{R}^{\mathfrak{v},p}_{\tau_1}.$$

$$(S'[\overrightarrow{V_i/x_i}] \circ \textbf{let } (-) \Rightarrow y \textbf{ in } M[\overrightarrow{V_i/x_i}], \textbf{return } V)\ \mathcal{S}^{p}_{\tau_1,\rho}\ (\lambda y{:}\tau_1^*.(M^*\ S'^*)[\overrightarrow{v_i/x_i}])\ v.$$

If $p > 0$, from Lemma 4.3.6, it is enough to show that:

$$(S'[\overrightarrow{V_i/x_i}], M[\overrightarrow{V_i/x_i}, V/y])\ \mathcal{S}^{p-1}_{\tau_2,\rho}\ M^*[\overrightarrow{v_i/x_i}, v/y]\ S'^*[\overrightarrow{v_i/x_i}].$$

The result follows from the induction hypothesis for M and S'.

Case (lbd), $V = \lambda y{:}\tau_1.M$**.** The result follows from the induction hypothesis for M, unpacking the definition of $\mathcal{R}^{\mathfrak{v}}_{\tau_1 \to \tau_2}$.

Case (var), $V = y$**.** We know by assumption that $\overrightarrow{x_i : \sigma_i} \vdash y : \tau$. We need to prove:

$$\forall n \in \mathbb{N}.\ (\forall (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies (y[\overrightarrow{V_i/x_i}], y[\overrightarrow{v_i/x_i}]) \in \mathcal{R}^{\mathfrak{v},n}_{\tau}.$$

This is true because $y[\overrightarrow{V_i/x_i}] = V_j$, $y[\overrightarrow{v_i/x_i}] = v_j$ and $\tau = \sigma_j$ for some j.

Case (zero), $V = Z$**.** By definition of $\mathcal{R}^{\mathfrak{v},m}_{\mathbb{N}}$ we know that $(Z, Z) \in \mathcal{R}^{\mathfrak{v},m}_{\mathbb{N}}$ for any $m \in \mathbb{N}$.

Case (unit), $V = \star$**.** Analogous to the previous case.

Case (succ), $V = S(W)$**.** We need to prove that:

$$\forall n \in \mathbb{N}.\ (\forall (V_i, v_i) \in \mathcal{R}^{\mathfrak{v},n}_{\sigma_i}) \implies (S(W[\overrightarrow{V_i/x_i}]), \mathtt{succ}(W^*[\overrightarrow{v_i/x_i}])) \in \mathcal{R}^{\mathfrak{v},n}_{\mathbb{N}}.$$

From the induction hypothesis for W we know that $W[\overrightarrow{V_i/x_i}] = \overline{m}$ and $W^*[\overrightarrow{v_i/x_i}] = \overline{m}$ for some $m \in \mathbb{N}$. So $S(W[\overrightarrow{V_i/x_i}]) = \overline{m+1}$ and $\mathtt{succ}(W^*[\overrightarrow{v_i/x_i}]) = \overline{m+1}$, which gives us the required result. □

Lemma 4.3.12. *The function* $[\![-]\!] : (\vdash) \longrightarrow \mathit{Trees}_\Sigma$ *is a coalgebra morphism in the category of coalgebras for the functor* T*.*

Proof. We will use the definitions of the functor T and coalgebra morphisms c and a from Section 4.3.

First, we describe the ω-CPO structure on Trees_Σ. The partial order $\leq$ satisfies the following:

$$\forall tr \in \mathit{Trees}_\Sigma.\ \bot \leq tr \text{ and } (tr \leq \bot \iff tr = \bot) \tag{A.4}$$

$$\forall tr \in \mathit{Trees}_\Sigma.\ (\downarrow \leq tr \iff tr = \downarrow) \text{ and } (tr \leq \downarrow \iff tr = \bot \text{ or } tr = \downarrow) \tag{A.5}$$

$$\forall \sigma, \sigma' \in \Sigma,\ n, n' \in \mathbb{N},\ \overrightarrow{tr_k}, \overrightarrow{tr'_k} \in \mathit{Trees}_\Sigma.$$

$$\sigma_n(\overrightarrow{tr_k}) \leq \sigma'_{n'}(\overrightarrow{tr'_k}) \quad \text{iff} \quad \sigma = \sigma' \text{ and } n = n' \text{ and } \forall k \in \mathbb{N}.\ tr_k \leq tr'_k. \tag{A.6}$$

From these properties we see that a chain $tr_0 \leq tr_1 \leq tr_2 \leq \ldots$ can have one of the following three forms:

1. $\forall n \in \mathbb{N}.\ tr_n = \bot$. In this case define the least upper bound of the chain to be $\bigsqcup_{n \in \mathbb{N}} tr_n = \bot$.
2. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ tr_n = \downarrow$ and $\forall n < k.\ tr_n = \bot$. So we define the least upper bound to be $\bigsqcup_{n \in \mathbb{N}} tr_n = \downarrow$.
3. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ tr_n = \sigma_i(\overrightarrow{tr^n_j})$ and $\forall n < k.\ tr_n = \bot$. By property A.6 of $\leq$, σ and i need to stay the same for all n. So we can define the least upper bound as: $\bigsqcup_{n \in \mathbb{N}} tr_n = \sigma_i(\bigsqcup_{n \geq k} tr^n_1,\ \bigsqcup_{n \geq k} tr^n_2, \ldots)$.

Now we can define a partial order, $\leq_T$, on $T(\mathit{Trees}_\Sigma) = \{\downarrow, \bot\} + \Sigma \times \mathbb{N} \times (\mathit{Trees}_\Sigma)^\mathbb{N}$ which makes it into an ω-CPO:

$\forall x \in T(\mathit{Trees}_\Sigma).\ \bot \leq_T x$ and $(x \leq_T \bot \iff x = \bot)$

$\forall x \in T(\mathit{Trees}_\Sigma).\ (\downarrow \leq_T x \iff x = \downarrow)$ and $(x \leq_T \downarrow \iff x = \bot$ or $x = \downarrow)$

$\forall \sigma, \sigma' \in \Sigma,\ n, n' \in \mathbb{N},\ \overrightarrow{tr_k}, \overrightarrow{tr'_k} \in \mathit{Trees}_\Sigma.$

$(\sigma, n, (\overrightarrow{tr_k})) \leq_T (\sigma', n', (\overrightarrow{tr'_k}))$ iff $\sigma = \sigma'$ and $n = n'$ and $\forall k \in \mathbb{N}.\ tr_k \leq tr'_k$.

Given this definition, a chain $x_0 \leq_T x_1 \leq_T x_2 \leq_T \ldots$ can only have one of the three forms below:

1. $\forall n \in \mathbb{N}.\ x_n = \bot$. Define the least upped bound as: $\bigsqcup_{n\in\mathbb{N}} x_n = \bot$.
2. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ x_n = \downarrow$ and $\forall n < k.\ x_n = \bot$. Define the least upped bound as: $\bigsqcup_{n\in\mathbb{N}} x_n = \downarrow$.
3. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ x_n = (\sigma, i, (\overrightarrow{tr^n_j}))$ and $\forall n < k.\ x_n = \bot$. Again it must be the case that σ and i are the same for all $n \geq k$ so we define the least upper bound as: $\bigsqcup_{n\in\mathbb{N}} x_n = (\sigma, i, (\bigsqcup_{n\geq k} tr^n_1,\ \bigsqcup_{n\geq k} tr^n_2, \ldots))$.

To prove $[\![-]\!]$ is a coalgebra morphism, we must show that the following diagram commutes:

$$\begin{array}{ccc} (\vdash) & \xrightarrow{\quad [\![-]\!] \quad} & \mathit{Trees}_\Sigma \\ \downarrow a & & \downarrow c \\ \{\downarrow, \bot\} + \Sigma \times \mathbb{N} \times (\vdash)^\mathbb{N} & \xrightarrow{T([\![-]\!])} & \{\downarrow, \bot\} + \Sigma \times \mathbb{N} \times (\mathit{Trees}_\Sigma)^\mathbb{N} \end{array}$$

First show that for any chain $tr_0 \leq tr_1 \leq tr_2 \leq \ldots$ in Trees_Σ:

$$c(\bigsqcup_{n\in\mathbb{N}} tr_n) = \bigsqcup_{n\in\mathbb{N}} c(tr_n).$$

We do a case split on the structure of the chain following the cases that we identified previously:

1. $\forall n \in \mathbb{N}.\ tr_n = \bot$. Then $c(\bigsqcup_{n\in\mathbb{N}} tr_n) = \bot$ and $c(tr_n) = \bot$ for all n so we are done.
2. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ tr_n = \downarrow$ and $\forall n < k.\ tr_n = \bot$. Then $c(\bigsqcup_{n\in\mathbb{N}} tr_n) = c(\downarrow) = \downarrow$ and $\bigsqcup_{n\in\mathbb{N}} c(tr_n) = \bigsqcup_{n\geq k} \downarrow = \downarrow$.
3. $\exists k \in \mathbb{N}$ such that $\forall n \geq k.\ tr_n = \sigma_i(\overrightarrow{tr^n_j})$ and $\forall n < k.\ tr_n = \bot$. In this case:

$$c(\bigsqcup_{n\in\mathbb{N}} tr_n) = c(\bigsqcup_{n\geq k} tr_n) = c(\sigma_i(\bigsqcup_{n\geq k} tr^n_1,\ \bigsqcup_{n\geq k} tr^n_2, \ldots)) = (\sigma, i, (\bigsqcup_{n\geq k} tr^n_1,\ \bigsqcup_{n\geq k} tr^n_2, \ldots))$$

$$\bigsqcup_{n\in\mathbb{N}} c(tr_n) = \bigsqcup_{n\geq k} (\sigma, i, (tr^n_1, tr^n_2, \ldots)) = (\sigma, i, (\bigsqcup_{n\geq k} tr^n_1,\ \bigsqcup_{n\geq k} tr^n_2, \ldots)).$$

so the two sides are equal as required.

Now we can deduce that:

$$c([\![t]\!]) = c(\bigsqcup_{n\in\mathbb{N}} [\![t]\!]_n) = \bigsqcup_{n\in\mathbb{N}} c([\![t]\!]_n). \tag{A.7}$$

On the other side of the diagram we have $T([\![-]\!])(a(t))$ which by definition of T is equal to:

$$T([\![-]\!])(a(t)) = \begin{cases} \downarrow & \text{if } a(t) = \downarrow \\ \bot & \text{if } a(t) = \bot \\ (\sigma, k, \overrightarrow{[\![t'[\overline{i}/x]]\!]}) & \text{if } a(t) = (\sigma, k, \overrightarrow{t'[\overline{i}/x]}). \end{cases}$$

By definition of a the side conditions can be expressed in terms of the reduction behaviour of t:

$$T([\![-]\!])(a(t)) = \begin{cases} \downarrow & \text{if } t \longrightarrow^* \downarrow \\ \bot & \text{if } t \longrightarrow^\infty \\ (\sigma, k, \overrightarrow{\bigsqcup_{n\in\mathbb{N}} [\![t'[\overline{i}/x]]\!]_n}) & \text{if } t \longrightarrow^* \sigma(\overline{k}, x.t'). \end{cases}$$

By the definition of least upper bound in $T(\mathit{Trees}_\Sigma)$ we can rewrite the last case as:

$$T([\![-]\!])(a(t)) = \begin{cases} \downarrow & \text{if } t \longrightarrow^* \downarrow \\ \bot & \text{if } t \longrightarrow^\infty \\ \bigsqcup_{n\in\mathbb{N}} (\sigma, k, \overrightarrow{[\![t'[\overline{i}/x]]\!]_n}) & \text{if } t \longrightarrow^* \sigma(\overline{k}, x.t'). \end{cases} \tag{A.8}$$

To see that equations A.7 and A.8 are in fact equal we proceed by a case analysis on the reduction of t. By definition of $\longrightarrow$ the following cases are exhaustive and only one of them can occur:

1. $t \longrightarrow^* \downarrow$. There exists m such that $t \longrightarrow^m \downarrow$. By definition of $[\![-]\!]$ we can deduce that $\forall n \geq m+1.\ [\![t]\!]_n = \downarrow$ so $c([\![t]\!]_n) = \downarrow$. And $\forall n < m+1.\ [\![t]\!]_n = \bot$. Therefore we have:

   $$\bigsqcup_{n\in\mathbb{N}} c([\![t]\!]_n) = \downarrow .$$

   as required.

2. $t \longrightarrow^\infty$. For each $n \in \mathbb{N}$ there exists t_n such that $t \longrightarrow^n t_n$. Therefore $\forall n \in \mathbb{N}.\ [\![t]\!]_n = [\![t_n]\!]_0 = \bot$ so by the definition of least upper bound in $T(\mathit{Trees}_\Sigma)$:

   $$\bigsqcup_{n\in\mathbb{N}} c([\![t]\!]_n) = \bot.$$

3. $t \longrightarrow^* \sigma(\overline{k}, x.t')$. There exists $m \in \mathbb{N}$ such that $t \longrightarrow^m \sigma(\overline{k}, x.t')$. By definition of $[\![-]\!]_n$ we can deduce that:

   $$\forall n \geq m+1.\ [\![t]\!]_n = \sigma_k([\![t'[\overline{0}/x]]\!]_{n-m-1}, [\![t'[\overline{1}/x]]\!]_{n-m-1}, \ldots)$$

   $$\forall n < m+1.\ [\![t]\!]_n = \bot.$$

   From this we know that:

   $$\bigsqcup_{n\in\mathbb{N}} c([\![t]\!]_n) = \bigsqcup_{n\geq m+1} c([\![t]\!]_n) = \bigsqcup_{n\geq m+1} (\sigma, k, \overrightarrow{[\![t'[\overline{i}/x]]\!]_{n-m-1}}) = T([\![-]\!])(a(t)).$$

□

Proposition 4.3.14. *For any well-typed EPCF configuration (S, M), where $S : \tau \Rightarrow \rho$, and any ECPS computation t:*

$$((S, M), t) \in \mathcal{S}_{\tau,\rho} \cap \mathcal{S}'_{\tau,\rho} \implies [\![t]\!] = |S, M|[\downarrow /l_1, \downarrow /l_2, \ldots].$$

Proof. By coinduction. Define the relation $\mathcal{B} \subseteq (\mathit{Stack} \times \mathit{Comp}) \times (\vdash)$ as:

$$\mathcal{B} = \bigcup_{\tau,\rho} (\mathcal{S}_{\tau,\rho} \cap \mathcal{S}'_{\tau,\rho}).$$

We will show that $\mathcal{B}$ is a bisimulation in the abstract sense of Definition 3.3.1. Consider the following morphism of type $\mathcal{B} \longrightarrow T(\mathcal{B})$:

$$r((S, M), t) = \begin{cases} \downarrow & \text{if } (S, M) \rightarrowtail^* (id, \mathbf{return}\ V) \\ \bot & \text{if } (S, M) \rightarrowtail^\infty \\ (\sigma, n, \overrightarrow{((S', W\bar{l}), t'[\bar{l}/x])}) & \text{if } (S, M) \rightarrowtail^* (S', \sigma(\overline{n}; W)), \text{which implies} \\ & \text{from definition of } \mathcal{S} \text{ that } t \longrightarrow^* \sigma(\overline{n}, x.t'). \end{cases}$$

This morphism makes the following diagram commute, so it makes $\mathcal{B}$ into a bisimulation:

$$\begin{array}{ccccc} \mathit{Satck} \times \mathit{Comp} & \xleftarrow{\pi_1} & \mathcal{B} & \xrightarrow{\pi_2} & (\vdash) \\ \downarrow{\scriptstyle b} & & \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle a} \\ T(\mathit{Stack} \times \mathit{Comp}) & \xleftarrow{T(\pi_1)} & T(\mathcal{B}) & \xrightarrow{T(\pi_2)} & T(\vdash) \end{array}$$

By definition of b and r we have:

$$\forall((S, M), t) \in \mathcal{B}.\ b(\pi_1((S, M), t)) = T(\pi_1)(r((S, M), t)).$$

For the second equation: we know that for any $((S, M), t) \in \mathcal{B}$:

$$a(\pi_2((S, M), t)) = a(t) = \begin{cases} \downarrow & \text{if } t \longrightarrow^* \downarrow \\ \bot & \text{if } t \longrightarrow^\infty \\ (\sigma, k, \overrightarrow{t'[\overline{n}/x]}) & \text{if } t \longrightarrow^* \sigma(\overline{k}, x.t'). \end{cases} \tag{A.9}$$

$$T(\pi_2)(r((S, M), t)) = \begin{cases} \downarrow & \text{if } (S, M) \rightarrowtail^* (id, \mathbf{return}\ V) \\ \bot & \text{if } (S, M) \rightarrowtail^\infty \\ (\sigma, n, \overrightarrow{t'[\bar{l}/x]}) & \text{if } (S, M) \rightarrowtail^* (S', \sigma(\overline{n}; W)), \text{which implies} \\ & \text{from definition of } \mathcal{S} \text{ that } t \longrightarrow^* \sigma(\overline{n}, x.t'). \end{cases} \tag{A.10}$$

Expressions A.9 and A.10 can be proved equal by using the definitions of $\mathcal{S}$ and $\mathcal{S}'$. If $(S, M) \rightarrowtail^* (S', \sigma(\overline{n}; W))$ then we know by definition of $\mathcal{S}$ that $t \longrightarrow^* \sigma(\overline{n}, x.t')$. If $(S, M) \rightarrowtail^* (id, \mathbf{return}\ V)$ then again by definition of $\mathcal{S}$ we have $t \longrightarrow^* \downarrow$. If $(S, M) \rightarrowtail^\infty$, because the reduction relation $\rightarrowtail$ is deterministic we know that $(S, M) \not\rightarrowtail^* (id, \mathbf{return}\ V)$ and $(S, M) \not\rightarrowtail^* (S', \sigma(\overline{n}; W))$ for any $n \in \mathbb{N}$. Therefore, by definition of $\mathcal{S}'$, $t \not\longrightarrow^* \downarrow$ and

$t \not\longrightarrow^* \sigma(v, x.t')$. So by the definition of $\longrightarrow$ it must be the case that $t \longrightarrow^\infty$, as required. We can prove the reverse implication by making an assumption about the reduction of t and proceeding analogously.

By assumption, we know $((S, M), t) \in \mathcal{B}$. As discussed in Section 4.3, $Trees_\Sigma$ is a final coalgebra. From Lemmas 4.3.12 and 4.3.13 we know that $[\![-]\!]$ and β are coalgebra morphisms into it. Using the fact that $\mathcal{B}$ is a bisimulation, we can apply the coinduction proof principle, Proposition 3.3.2, to deduce:

$$[\![t]\!] = \beta(S, M) = |S, M|^* = |S, M|[\downarrow /l_1, \downarrow /l_2, \ldots].$$

as required. □

Appendix B

Proofs about Applicative Bisimilarity

Proposition 5.2.3. *Applicative $\mathfrak{P}$-bisimilarity coincides with the intersection between applicative $\mathfrak{P}$-similarity and its converse:*

$$(\sim) = (\precsim) \cap (\precsim)^{op}.$$

Proof. We prove each inclusion in turn.

"$\subseteq$". Bisimilarity is a bisimulation, hence also a simulation so $(\sim) \subseteq (\precsim)$. Therefore $(\sim)^{op} \subseteq (\precsim)^{op}$. But bisimilarity is by definition symmetric so $(\sim) = (\sim)^{op}$ so we can deduce that $(\sim) \subseteq (\precsim)^{op}$. Hence, $(\sim) \subseteq (\precsim) \cap (\precsim)^{op}$ as required.

"$\supseteq$". The strategy is to check that $(\precsim) \cap (\precsim)^{op}$ is a symmetric simulation. Consider $(s,t) \in (\precsim) \cap (\precsim)^{op}$. Then $s \precsim t$ and $t \precsim s$, which implies $t \precsim^{op} s$ and $t \precsim s$. Therefore, $(t,s) \in (\precsim) \cap (\precsim)^{op}$ and $(\precsim) \cap (\precsim)^{op}$ is symmetric.

The relation $(\precsim) \cap (\precsim)^{op}$ satisfies the first three conditions in the definition of simulation because we know $\precsim$ is a simulation. For the fourth condition assume:

$$(v,u) \in ((\precsim) \cap (\precsim)^{op})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}.$$

Then $v \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} w$ and $w \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} v$ and since $\precsim$ is a simulation we know that:

$$\forall \vdash \overrightarrow{u_i : A_i}.\ v(\overrightarrow{u_i}) \precsim^{\mathfrak{c}} w(\overrightarrow{u_i})$$

$$\forall \vdash \overrightarrow{u_i : A_i}.\ w(\overrightarrow{u_i}) \precsim^{\mathfrak{c}} v(\overrightarrow{u_i}).$$

Therefore:

$$\forall \vdash \overrightarrow{u_i : A_i}.\ (v(\overrightarrow{u_i}), w(\overrightarrow{u_i})) \in ((\precsim) \cap (\precsim)^{op})^{\mathfrak{c}}.$$

So we have shown that $(\precsim) \cap (\precsim)^{op}$ is a symmetric simulation and it is therefore included in the union of all symmetric simulations $\sim$. □

Lemma 5.3.1. *Applicative $\mathfrak{P}$-similarity is a preorder. Applicative $\mathfrak{P}$-bisimilarity is an equivalence relation.*

Proof. Prove that similarity is reflexive and transitive.

Reflexivity. Let $\mathcal{I}^{\mathfrak{v}}_A$ and $\mathcal{I}^{\mathfrak{c}}$ be the identity relations. We will show $(\mathcal{I}^{\mathfrak{v}}_A, \mathcal{I}^{\mathfrak{c}}) \subseteq \precsim$ so $\precsim$ is reflexive. For this it suffices to show $(\mathcal{I}^{\mathfrak{v}}_A, \mathcal{I}^{\mathfrak{c}})$ is a simulation. Conditions 1 and 2 are satisfied by definition. For condition 3 assume $s\ \mathcal{I}^{\mathfrak{c}}\ t$. Then it must be the case that $s = t$ so $[\![s]\!] = [\![t]\!]$. Therefore $\forall P \in \mathfrak{P}.\ ([\![s]\!] \in P \implies [\![t]\!] \in P)$. For condition 4 assume $v\ \mathcal{I}^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ u$. Then $v = u$ so the result follows from the definition of $\mathcal{I}^{\mathfrak{c}}$.

Transitivity. First assume that there exist some closed computations such that $r \precsim s \precsim t$. Then we know from the definition of similarity and transitivity of implication that:

$$\forall P \in \mathfrak{P}.\ [\![r]\!] \in P \implies [\![t]\!] \in P.$$

Since similarity is the greatest relation with property 3, it must be the case that $r \precsim t$.

Assume $u \precsim v \precsim w$. Then there exist simulations $\mathcal{R}$ and $\mathcal{S}$ such that $u\ \mathcal{R}^{\mathfrak{v}}_A\ v$ and $v\ \mathcal{S}^{\mathfrak{v}}_A\ w$. Proceed by a case distinction on A.

If $A = \texttt{nat}$ or $A = \texttt{unit}$, then it must be the case that $u = v = w$. Let $\mathcal{T}^{\mathfrak{v}}_A = \{(u, w)\}$, $\mathcal{T}^{\mathfrak{v}}_{B \neq A} = \emptyset$, $\mathcal{T}^{\mathfrak{c}} = \emptyset$. The relation $(\mathcal{T}^{\mathfrak{v}}_A, \mathcal{T}^{\mathfrak{v}}_{B\neq A}, \mathcal{T}^{\mathfrak{c}})$ is a simulation because $u = w$, and is therefore included in $\precsim$.

If $A = \neg(A_1, \ldots, A_n)$, consider the candidate simulation $\mathcal{T}^{\mathfrak{v}}_A = \{(u, w)\}$, $\mathcal{T}^{\mathfrak{v}}_{B\neq A} = \emptyset$, $\mathcal{T}^{\mathfrak{c}} = \{(u\ (w_1, \ldots, w_n), w\ (w_1, \ldots, w_n)) \mid\vdash w_1 : A_1, \ldots, \vdash w_n : A_n\}$. Condition 4 is satisfied by definition of $\mathcal{T}^{\mathfrak{c}}$. Condition 3 is satisfied because $u\ \mathcal{R}^{\mathfrak{v}}_A\ v$ and $v\ \mathcal{S}^{\mathfrak{v}}_A\ w$ imply:

$$\forall \vdash w_1 : A_1, \ldots, \vdash w_n : A_n.\ \forall P \in \mathfrak{P}.\ [\![u(w_1, \ldots, w_n)]\!] \in P \implies [\![w(w_1, \ldots, w_n)]\!] \in P.$$

From Proposition 5.2.3 we know that:

$$(\sim) = (\precsim) \cap (\precsim)^{op}.$$

The relation $(\precsim) \cap (\precsim)^{op}$ is reflexive and transitive because similarity is, and it is by definition symmetric. So $\sim$ is an equivalence relation. □

Lemma 5.3.4. *Consider a well-typed relation $\mathcal{R}$ that is a preorder. The compatibility rules* (COMP6)*,* (COMP7)*,* (COMP8) *and* (COMP10) *are equivalent to the conjunction of their single-premise versions.*

Proof. As an example, consider the case of ((COMP7)$\Longleftrightarrow$(COMP7L) and (COMP7R$_i$)). The proof for the other cases is analogous.

Assume that (COMP7) is true and that

$$\Gamma, x : \neg(\overrightarrow{A}) \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A})}\ v'$$

$$\Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i \text{ for each } i.$$

By reflexivity of $\mathcal{R}$ we know $\Gamma \vdash w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w_i$ for each i. So we can apply rule (COMP7) to deduce the conclusion of rule (COMP7L).

Also by reflexivity, we know $\Gamma, x : \neg(\overrightarrow{A}) \vdash v\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A})}\ v$ and $\Gamma \vdash w_j\ \mathcal{R}^{\mathfrak{v}}_{A_j}\ w_j$ for each $j \neq i$. Now apply rule (COMP7) to deduce the conclusion of rule (COMP7R$_i$).

For the reverse implication assume that the rules (COMP7L) and (COMP7R$_i$) are true and that:

$$\Gamma, x : \neg(\overrightarrow{A}) \vdash v \ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A})} \ v'$$

$$\Gamma \vdash w_i \ \mathcal{R}^{\mathfrak{v}}_{A_i} \ w'_i \text{ for each } i.$$

Because $\mathcal{R}$ is well-typed we know that all the values considered above are well-typed. Therefore, we can apply (COMP7L) and (COMP7R$_i$) one by one to obtain:

$$\Gamma \vdash (\mu x.v)(\overrightarrow{w}) \ \mathcal{R}^{\mathfrak{c}} \ (\mu x.v')(\overrightarrow{w})$$

$$\Gamma \vdash (\mu x.v')(w_1, w_2, \ldots, w_n) \ \mathcal{R}^{\mathfrak{c}} \ (\mu x.v')(w'_1, w_2, \ldots, w_n)$$

$$\ldots$$

$$\Gamma \vdash (\mu x.v')(w'_1, \ldots, w'_i, \ldots, w'_{n-1}, w_n) \ \mathcal{R}^{\mathfrak{c}} \ (\mu x.v')(w'_1, \ldots, w'_i, \ldots, w'_{n-1}, w'_n)$$

so by transitivity of $\mathcal{R}$ we have $\Gamma \vdash (\mu x.v)(\overrightarrow{w}) \ \mathcal{R}^{\mathfrak{c}} \ (\mu x.v')(\overrightarrow{w'})$ as required. □

B.1 Howe's Method

Lemma 5.4.3 (From [SV17, Appendix])**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is reflexive:*

1. *The Howe extension of $\mathcal{R}$, $\mathcal{R}^{\mathcal{H}}$, is compatible and hence reflexive.*
2. *$\mathcal{R}^{\circ} \subseteq \mathcal{R}^{\mathcal{H}}$.*

Proof. 1. From the definitions of compatibility and compatible refinement we see that $\widehat{\mathcal{R}^{\mathcal{H}}} \subseteq \mathcal{R}^{\mathcal{H}}$ implies that $\mathcal{R}^{\mathcal{H}}$ is compatible.

We know $\mathcal{R}$ is reflexive. Therefore $\mathcal{R}^{\circ}$ is reflexive. Let Id be the identity well-typed open relation. Then:

$$\widehat{\mathcal{R}^{\mathcal{H}}} = Id \circ \widehat{\mathcal{R}^{\mathcal{H}}} \subseteq \mathcal{R}^{\circ} \circ \widehat{\mathcal{R}^{\mathcal{H}}} = \mathcal{R}^{\mathcal{H}}.$$

To show $\mathcal{R}^{\mathcal{H}}$ is reflexive, that is, for all terms s, $s\mathcal{R}^{\mathcal{H}}s$ we can proceed by induction on s using the fact that $\mathcal{R}^{\mathcal{H}}$ is compatible.

2. Now we know that $\mathcal{R}^{\mathcal{H}}$ is reflexive. By the definition of compatible refinement we can easily see that this implies $\widehat{\mathcal{R}^{\mathcal{H}}}$ is reflexive. Therefore:

$$\mathcal{R}^{\circ} = \mathcal{R}^{\circ} \circ Id \subseteq \mathcal{R}^{\circ} \circ \widehat{\mathcal{R}^{\mathcal{H}}} = \mathcal{R}^{\mathcal{H}}.$$

□

Lemma 5.4.4 (From [SV17, Appendix])**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is transitive:*

$$\mathcal{R}^{\circ} \circ \mathcal{R}^{\mathcal{H}} \subseteq \mathcal{R}^{\mathcal{H}}.$$

Proof. Consider three terms s, t and r, either values or computations, such that $s\ \mathcal{R}^{\mathcal{H}}\ t\ \mathcal{R}^{\circ}\ r$. By definition of $\mathcal{R}^{\mathcal{H}}$ this means there exists a term t' such that:

$$s\ \widehat{\mathcal{R}^{\mathcal{H}}}\ t'\ \mathcal{R}^{\circ}\ t\ \mathcal{R}^{\circ}\ r.$$

Since $\mathcal{R}$ is transitive, $\mathcal{R}^{\circ}$ is also transitive. Therefore:

$$s\ \widehat{\mathcal{R}^{\mathcal{H}}}\ t'\ \mathcal{R}^{\circ}\ r$$

so by definition of $\mathcal{R}^{\mathcal{H}}$ we have $s\ \mathcal{R}^{\mathcal{H}}\ r$. □

Lemma 5.4.5 (Substitutivity)**.** *Given a well-typed relation $\mathcal{R}$ on closed terms that is transitive, its Howe extension satisfies the following two value-substitutivity properties:*

1. $\overrightarrow{x_i : A_i}, y : B \vdash s\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ t$ *and* $\overrightarrow{x_i : A_i} \vdash v\ \mathcal{R}_B^{\mathcal{H},\mathfrak{v}}\ w \implies \overrightarrow{x_i : A_i} \vdash s[v/y]\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ t[w/y]$.
2. $\overrightarrow{x_i : A_i}, y : B \vdash u\ \mathcal{R}_C^{\mathcal{H},\mathfrak{v}}\ u'$ *and* $\overrightarrow{x_i : A_i} \vdash v\ \mathcal{R}_B^{\mathcal{H},\mathfrak{v}}\ w \implies \overrightarrow{x_i : A_i} \vdash u[v/y]\ \mathcal{R}_C^{\mathcal{H},\mathfrak{v}}\ u'[w/y]$.

Proof. We prove the two statements by induction on s and u:

If s is a computation then $\overrightarrow{x_i : A_i}, y : B \vdash s\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ t$ was derived using rule (HC) so it must be the case that:

$$\overrightarrow{x_i : A_i}, y : B \vdash s\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{c}}\ t' \tag{B.1}$$

$$\overrightarrow{x_i : A_i}, y : B \vdash t'\ \mathcal{R}^{\circ,\mathfrak{c}}\ t. \tag{B.2}$$

By the definition of open extension we know from equation B.2 that:

$$\overrightarrow{x_i : A_i} \vdash t'[w/y]\ \mathcal{R}^{\circ,\mathfrak{c}}\ t[w/y].$$

If we can prove

$$\overrightarrow{x_i : A_i} \vdash s[v/y]\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{c}}\ t'[w/y]$$

then we could use rule (HC) to deduce the desired result, $\overrightarrow{x_i : A_i} \vdash s[v/x]\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ t[w/x]$.

Case $s = (\mu z.u)(\overrightarrow{w_i})$**.** It must be the case that equation B.1 was obtained using rule (C7) so we know that:

$$t' = (\mu z.u')(\overrightarrow{w_i'})$$

$$\overrightarrow{x_i : A_i}, y : B, z : \neg(\overrightarrow{C}) \vdash u\ \mathcal{R}^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{C})}\ u'$$

$$\overrightarrow{x_i : A_i}, y : B \vdash w_i\ \mathcal{R}^{\mathcal{H},\mathfrak{v}}_{C_i}\ w_i' \text{ for each } i.$$

By induction hypothesis for u and $\overrightarrow{w_i}$ we can deduce:

$$\overrightarrow{x_i : A_i}, z : \neg(\overrightarrow{C}) \vdash u[v/y]\ \mathcal{R}^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{C})}\ u'[w/y]$$

$$\overrightarrow{x_i : A_i} \vdash w_i[v/y]\ \mathcal{R}^{\mathcal{H},\mathfrak{v}}_{C_i}\ w_i'[w/y] \text{ for each } i$$

and then apply rule (C7) to get $\overrightarrow{x_i : A_i} \vdash s[v/y]\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{c}}\ t'[w/y]$.

Case $s = \downarrow$. In this case equation B.1 was obtained from rule (C9), so $t' = \downarrow$ and $s[v/y] = t'[w/y] = \downarrow$. We can then deduce $\overrightarrow{x_i : A_i} \vdash s[v/y]\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{c}}\ t'[w/y]$ by rule (C9).

Cases $s = u\ (\overrightarrow{w_i})$, $s = \sigma(u, x.s')$, $s = \mathtt{case}\ u\ \mathtt{in}\ \{\mathtt{zero} \Rightarrow s',\ \mathtt{succ}(z) \Rightarrow s''\}$. Analogous to the case $s = (\mu z.u)(\overrightarrow{w_i})$.

If u is a value then $\overrightarrow{x_i : A_i}, y : B \vdash u\ \mathcal{R}_C^{\mathcal{H},\mathfrak{v}}\ u'$ was derived using rule (HV) so it must be the case that:

$$\overrightarrow{x_i : A_i}, y : B \vdash u\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{v}}_C\ u'' \tag{B.3}$$

$$\overrightarrow{x_i : A_i}, y : B \vdash u''\ \mathcal{R}_C^{\circ,\mathfrak{v}}\ u'. \tag{B.4}$$

From equation B.4 by the definition of open extension we have:

$$\overrightarrow{x_i : A_i} \vdash u''[w/y]\ \mathcal{R}_C^{\circ,\mathfrak{v}}\ u'[w/y] \tag{B.5}$$

so using rule (HV) it suffices to prove:

$$\overrightarrow{x_i : A_i} \vdash u[v/y]\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{v}}_C\ u''[w/y]. \tag{B.6}$$

Case $u = \lambda\overrightarrow{z_j}{:}\overrightarrow{C_j}.r$. It must be the case that equation B.3 was obtained by rule (C3) so:

$$u'' = \lambda\overrightarrow{z_j}{:}\overrightarrow{C_j}.r'$$

$$\overrightarrow{x_i : A_i}, y : B, \overrightarrow{z_j : C_j} \vdash r\ \mathcal{R}^{\mathcal{H},\mathfrak{c}}\ r'.$$

By induction hypothesis for r and applying rule (C3) we can deduce:

$$\overrightarrow{x_i : A_i} \vdash \lambda\overrightarrow{z_j}{:}\overrightarrow{C_j}.r[v/y]\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{v}}_{\neg(\overrightarrow{C_j})}\ \lambda\overrightarrow{z_j}{:}\overrightarrow{C_j}.r'[w/y].$$

Case $u = \mathtt{succ}(u')$. Analogous to the previous case.

Cases $u = \mathtt{zero}$ and $u = \star$. Analogous to the case $s = \downarrow$.

Case $u = z$. **If** $z \neq y$ then $z = x_j$ for some j. Then equation B.3 must have been the conclusion of rule (C1) so $u'' = z$. By rule (C1) we have:

$$\overrightarrow{x_i : A_i} \vdash x_j\ \widehat{\mathcal{R}^{\mathcal{H}}}^{\mathfrak{v}}_C\ x_j$$

which is what we had to prove.

If $z = y$ then instead of going through equation B.6 we will prove directly $\overrightarrow{x_i : A_i} \vdash u[v/y]\ \mathcal{R}_C^{\mathcal{H},\mathfrak{v}}\ u'[w/y]$, that is:

$$\overrightarrow{x_i : A_i} \vdash v\ \mathcal{R}_C^{\mathcal{H},\mathfrak{v}}\ u'[w/y].$$

Because $\mathcal{R}$ is transitive, we can apply Lemma 5.4.4 to obtain: $\mathcal{R}^{\circ} \circ \mathcal{R}^{\mathcal{H}} \subseteq \mathcal{R}^{\mathcal{H}}$. We already know equation B.5:

$$\overrightarrow{x_i : A_i} \vdash u''[w/y]\ \mathcal{R}_C^{\circ,\mathfrak{v}}\ u'[w/y]$$

so we just need to show:

$$\overrightarrow{x_i : A_i} \vdash v \; \mathcal{R}_C^{\mathcal{H},\mathfrak{v}} \; u''[w/y].$$

Equation B.3 must be the conclusion of rule (C1) so $u'' = y$. Then we are left to prove:

$$\overrightarrow{x_i : A_i} \vdash v \; \mathcal{R}_C^{\mathcal{H},\mathfrak{v}} \; w$$

which we know by the initial assumption. □

Lemma 5.4.6. *Consider a well-typed closed relation $\leq$ that is a $\mathfrak{P}$-simulation. For any closed values v and w:*

$$\vdash v \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w \implies v = w.$$

Proof. Since v is a closed value of type `nat` there exists $n \in \mathbb{N}$ such that $v = \overline{n}$. The equation $\vdash \overline{n} \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w$ must have been derived by rule (HV) so:

$$\vdash \overline{n} \; \widehat{\leq^{\mathcal{H}}}_{\mathtt{nat}}^{\mathfrak{v}} \; u \quad \text{and} \quad \vdash u \leq_{\mathtt{nat}}^{\circ,\mathfrak{v}} w.$$

But u and w are closed so $u \leq_{\mathtt{nat}}^{\mathfrak{v}} w$. Since $\leq$ is a simulation it follows that $u = w$. So we know:

$$\vdash \overline{n} \; \widehat{\leq^{\mathcal{H}}}_{\mathtt{nat}}^{\mathfrak{v}} \; w. \tag{B.7}$$

We prove by induction on n that:

$$\forall \vdash w : \mathtt{nat}. \; \vdash \overline{n} \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w \implies \overline{n} = w.$$

Base case, $n = 0$**.** From $\vdash \overline{n} \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w$ we deduce equation B.7, which must be the conclusion of rule (C4). Therefore $\overline{n} = w = \mathtt{zero}$.

Induction step. Assume $\overline{n+1} \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w'$ for some arbitrary $\vdash w' : \mathtt{nat}$. Then equation B.7 becomes:

$$\vdash \mathtt{succ}(\overline{n}) \; \widehat{\leq^{\mathcal{H}}}_{\mathtt{nat}}^{\mathfrak{v}} \; w'.$$

This must be the conclusion of rule (C5) so $w' = \mathtt{succ}(w'')$ and $\overline{n} \leq_{\mathtt{nat}}^{\mathcal{H},\mathfrak{v}} w''$.

We can instantiate the induction hypothesis with the last equation to obtain $w'' = \overline{n}$. This means that $\overline{n+1} = \mathtt{succ}(\overline{n}) = \mathtt{succ}(w'') = w'$ as required. □

Lemma 5.4.7 (Key Lemma)**.** *Consider a decomposable set of Scott-open observations $\mathfrak{P}$. Consider a well-typed closed relation $\leq$ that is a preorder and a $\mathfrak{P}$-simulation. For any closed computations s and t, $\vdash s \leq^{\mathcal{H},\mathfrak{c}} t$ implies:*

$$\forall n \in \mathbb{N}. \; \forall P \in \mathfrak{P}. \; [\![s]\!]_n \in P \implies [\![t]\!] \in P.$$

Proof. We prove by induction on $n \in \mathbb{N}$ that:

$$\forall n \in \mathbb{N}. \; \forall \text{ computations } s', t'. \; \vdash s' \leq^{\mathcal{H},\mathfrak{c}} t' \implies (\forall P \in \mathfrak{P}. \; [\![s']\!]_n \in P \implies [\![t']\!] \in P).$$

Base case, $n = 0$**.** By definition $[\![s']\!]_0 = \bot$. Assume that $[\![s']\!]_0 \in P$. Then by upwards closure of P it follows that $\mathit{Tree}_\Sigma \subseteq P$, so we also have $[\![t']\!] \in P$, as required.

Induction step. The induction hypothesis is:

$$\forall k < n+1 \in \mathbb{N}.\ \forall \text{ computations } s', t'.$$
$$\vdash s' \leq^{\mathcal{H},\mathfrak{c}} t' \implies (\forall P \in \mathfrak{P}.\ [\![s']\!]_k \in P \implies [\![t']\!] \in P).$$

Assume that $\vdash s \leq^{\mathcal{H},\mathfrak{c}} t$. This must be the conclusion of rule (HC) so there exists r such that:

$$\vdash s \leq^{\widehat{\mathcal{H}}^{\mathfrak{c}}} r \quad \text{and} \quad \vdash r \leq^{\circ,\mathfrak{c}} t.$$

But r and t are closed terms so we in fact know $r \leq^{\mathfrak{c}} t$. Because $\leq$ is a simulation it follows that:

$$\forall P' \in \mathfrak{P}.\ [\![r]\!] \in P' \implies [\![t]\!] \in P'.$$

Therefore, it suffices to show:

$$\forall P \in \mathfrak{P}.\ [\![s]\!]_{n+1} \in P \implies [\![r]\!] \in P.$$

To do this we proceed by a case split on the structure of s.

Case $s = v(w_1, \ldots, w_n)$**.** Because s is a well-typed closed computation, it must be the case that $s = (\lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.s')\ (w_1, \ldots, w_n)$.

The equation $\vdash s \leq^{\widehat{\mathcal{H}}^{\mathfrak{c}}} r$ must have been obtained by rule (C3) so it must be the case that:

$$r = (\lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.r')\ (w'_1, \ldots, w'_n) \tag{B.8}$$
$$\vdash \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.s' \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.r' \tag{B.9}$$
$$\vdash w_i \leq^{\mathcal{H},\mathfrak{v}}_{A_i} w'_i \text{ for each } i. \tag{B.10}$$

By definition of $[\![-]\!]_{(-)}$ we know $[\![s]\!]_{n+1} = [\![(\lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.s')\ (w_1, \ldots, w_n)]\!]_{n+1} = [\![s'[\overrightarrow{w_i/x_i}]]\!]_n$, and similarly for r. So we only need to prove:

$$\forall P \in \mathfrak{P}.\ [\![s'[\overrightarrow{w_i/x_i}]]\!]_n \in P \implies [\![r'[\overrightarrow{w'_i/x_i}]]\!] \in P.$$

Equation B.9 must have been obatined by rule (HV) do there exists p such that:

$$\vdash \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.s' \ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ p \tag{B.11}$$
$$\vdash p \leq^{\circ,\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.r'. \tag{B.12}$$

Equation B.11 must be the conclusion of rule (C3) so:

$$p = \lambda \overrightarrow{x_i}{:}\overrightarrow{A_i}.p' \quad \text{and} \quad \overrightarrow{x_i : A_i} \vdash s' \leq^{\mathcal{H},\mathfrak{c}} p'.$$

Because $\leq$ is transitive we can apply Lemma 5.4.5 to obtain the substitutivity property for $\leq^{\mathcal{H}}$. Using this and equation B.10 we can deduce:

$$\vdash s'[\overrightarrow{w_i/x_i}] \leq^{\mathcal{H},\mathfrak{c}} p'[\overrightarrow{w'_i/x_i}].$$

Apply the induction hypothesis for this to obtain:

$$\forall P \in \mathfrak{P}.\ [\![s'[\overrightarrow{w_i/x_i}]]\!]_n \in P \implies [\![p'[\overrightarrow{w_i'/x_i}]]\!] \in P.$$

We know that $\vdash \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.p' \leq^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.r'$ from equation B.12. By the definition of simulation we then have:

$$\forall P \in \mathfrak{P}.\ [\![p'[\overrightarrow{w_i'/x_i}]]\!] \in P \implies [\![r'[\overrightarrow{w_i'/x_i}]]\!] \in P.$$

From here we obtain the required result:

$$\forall P \in \mathfrak{P}.\ [\![s'[\overrightarrow{w_i/x_i}]]\!]_n \in P \implies [\![r'[\overrightarrow{w_i'/x_i}]]\!] \in P.$$

Case $s = \sigma(v, x.s')$**.** In this case $\vdash s \ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{c}}\ r$ is the conclusion of rule (C8) so we know:

$$r = \sigma(v', x.r')$$

$$\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\mathtt{nat}} v'$$

$$x : \mathtt{nat} \vdash s' \leq^{\mathcal{H},\mathfrak{c}} r'.$$

Using $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\mathtt{nat}} v'$ and Lemma 5.4.6 deduce $v = v' = \overline{k}$.

By reflexivity of $\leq^{\mathcal{H}}$ (Lemma 5.4.3) we have $\vdash \overline{m} \leq^{\mathcal{H},\mathfrak{v}}_{\mathtt{nat}} \overline{m}$. Using $x : \mathtt{nat} \vdash s' \leq^{\mathcal{H},\mathfrak{c}} r'$ we obtain by substitutivity (Lemma 5.4.5) that:

$$\forall m \in \mathbb{N}.\ \vdash s'[\overline{m}/x] \leq^{\mathcal{H},\mathfrak{c}} r'[\overline{m}/x].$$

Applying the induction hypothesis to this we we obtain:

$$\forall m \in \mathbb{N}.\ \forall P' \in \mathfrak{P}.\ [\![s'[\overline{m}/x]]\!]_n \in P' \implies [\![r'[\overline{m}/x]]\!] \in P'. \tag{B.13}$$

By definition of $[\![-]\!]_{(-)}$ it suffices to prove:

$$\forall P \in \mathfrak{P}.\ \sigma_k([\![s'[\overline{0}/x]]\!]_n, \ldots, [\![s'[\overline{k}/x]]\!]_n, \ldots) \in P \implies \sigma_k([\![r'[\overline{0}/x]]\!], \ldots, [\![r'[\overline{k}/x]]\!], \ldots) \in P.$$

Assume $\sigma_k([\![s'[\overline{0}/x]]\!]_n, \ldots, [\![s'[\overline{k}/x]]\!]_n, \ldots) \in P$. Then by decomposability of $\mathfrak{P}$, Definition 5.3.6, we know there exist observations $\overrightarrow{P'_m}$ such that:

$$\forall\overrightarrow{p'_m} \in \overrightarrow{P'_m}.\ \sigma_k(\overrightarrow{p'_m}) \in P$$

$$\text{for each } m \in \mathbb{N} \quad [\![s'[\overline{m}/x]]\!]_n \in P'_m.$$

By equation B.13 we can deduce that for all $m \in \mathbb{N}$, $[\![r'[\overline{m}/x]]\!] \in P'_m$. So we have

$$\sigma_k([\![r'[\overline{0}/x]]\!], \ldots, [\![r'[\overline{k}/x]]\!], \ldots) \in P$$

as required.

Case $s = (\mu x.v)(\overrightarrow{w_i})$**.** In this case $s \widehat{\leq^{\mathcal{H}}}^{\mathfrak{c}} r$ is the conclusion of rule (C7) so:

$$r = (\mu x.v')(\overrightarrow{w_i'}) \tag{B.14}$$

$$x : \neg(\overrightarrow{A_i}) \vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} v' \tag{B.15}$$

$$w_i \leq^{\mathcal{H},\mathfrak{v}}_{A_i} w_i' \text{ for each } i. \tag{B.16}$$

By definition of $[\![-]\!]_{(-)}$ it suffices to prove:

$$\forall P \in \mathfrak{P}.\ [\![v[\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y})/x]\ (\overrightarrow{w_i})]\!]_n \in P \implies [\![v'[\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y})/x]\ (\overrightarrow{w_i'})]\!] \in P.$$

By equation B.15 we can deduce using context weakening that:

$$\overrightarrow{y : A_i}, x : \neg(\overrightarrow{A_i}) \vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} v'.$$

Because $\leq$ is reflexive, we can apply Lemma 5.4.3 to deduce $\leq^{\mathcal{H}}$ is reflexive. Therefore:

$$\overrightarrow{y : A_i} \vdash y_i \leq^{\mathcal{H},\mathfrak{v}}_{A_i} y_i.$$

From the last two equations and from rule (C7) we can deduce:

$$\overrightarrow{y : A_i} \vdash (\mu x.v)(\overrightarrow{y}) \ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{c}}\ (\mu x.v')(\overrightarrow{y}).$$

By reflexivity of $\leq$ we know:

$$\overrightarrow{y : A_i} \vdash (\mu x.v')(\overrightarrow{y}) \leq^{\circ,\mathfrak{c}} (\mu x.v')(\overrightarrow{y}).$$

Using the last two equations and rule (HC) we obtain:

$$\overrightarrow{y : A_i} \vdash (\mu x.v)(\overrightarrow{y}) \leq^{\mathcal{H},\mathfrak{c}} (\mu x.v')(\overrightarrow{y}).$$

From this, using rule (C3), we have:

$$\vdash \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y})\ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}).$$

By reflexivity of $\leq$ we have:

$$\vdash \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}) \leq^{\circ,\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}).$$

From the last two equations, by rule (HV), we obtain:

$$\vdash \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y}) \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}).$$

Using this last equation and equation B.15: $x : \neg(\overrightarrow{A_i}) \vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} v'$, we obtain by substitutivity for $\leq^{\mathcal{H}}$, Lemma 5.4.5, that:

$$\vdash v[(\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y}))/x] \leq^{\mathcal{H},\mathfrak{v}}_{\neg(\overrightarrow{A_i})} v'[(\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}))/x].$$

From Lemma 5.4.3 we can deduce $\leq^{\mathcal{H}}$ is compatible. Using the last equation and equation B.16, $\vdash w_i \leq^{\mathcal{H},\mathfrak{v}}_{A_i} w_i'$ for each i, we obtain by compatibility:

$$\vdash v[(\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y}))/x]\ (\overrightarrow{w_i}) \leq^{\mathcal{H},\mathfrak{c}} v'[(\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y}))/x]\ (\overrightarrow{w_i'}).$$

By applying the induction hypothesis to this we obtain the desired result:

$$\forall P \in \mathfrak{P}.\ [\![v[\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v)(\overrightarrow{y})/x]\ (\overrightarrow{w_i})]\!]_n \in P \implies [\![v'[\lambda\overrightarrow{y}{:}\overrightarrow{A_i}.(\mu x.v')(\overrightarrow{y})/x]\ (\overrightarrow{w_i'})]\!] \in P.$$

Case $s = \texttt{case}\ v\ \texttt{in}\ \{\texttt{zero} \Rightarrow s',\ \texttt{succ}(y) \Rightarrow s''\}$**.** In this case $\vdash s\ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{c}}\ r$ is the conclusion of rule (C10) so we know:

$$r = \texttt{case}\ v'\ \texttt{in}\ \{\texttt{zero} \Rightarrow r',\ \texttt{succ}(y) \Rightarrow r''\}$$

$$\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{nat}} v'$$

$$\vdash s' \leq^{\mathcal{H},\mathfrak{c}} r'$$

$$y : \texttt{nat} \vdash s'' \leq^{\mathcal{H},\mathfrak{c}} r''.$$

Using $\vdash v \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{nat}} v'$ we can apply Lemma 5.4.6 to deduce $v = v'$.

If $v = v' = \texttt{zero}$ then it suffices to prove:

$$\forall P \in \mathfrak{P}.\ [\![s']\!]_n \in P \implies [\![r']\!] \in P.$$

We can deduce this using the induction hypothesis for s' and r' and $\vdash s' \leq^{\mathcal{H},\mathfrak{c}} r'$.

If $v = v' = \texttt{succ}(w)$ **where** $\vdash w : \texttt{nat}$ then it suffices to prove:

$$\forall P \in \mathfrak{P}.\ [\![s''[w/y]]\!]_n \in P \implies [\![r''[w/y]]\!] \in P.$$

By reflexivity of $\leq^{\mathcal{H}}$ (Lemma 5.4.3) we have $\vdash w \leq^{\mathcal{H},\mathfrak{v}}_{\texttt{nat}} w$. We can use this, $y : \texttt{nat} \vdash s'' \leq^{\mathcal{H},\mathfrak{c}} r''$ and substitutivity for $\leq^{\mathcal{H}}$ (Lemma 5.4.5), to deduce:

$$\vdash s''[w/y] \leq^{\mathcal{H},\mathfrak{c}} r''[w/y].$$

We then get the desired result from the induction hypothesis for $s''[w/y]$ and $r''[w/y]$.

Case $s = \downarrow$**.** In this case $\vdash s\ \widehat{\leq^{\mathcal{H}}}^{\mathfrak{c}}\ r$ is the conclusion of rule (C9) so $r = \downarrow$. Then $[\![s]\!]_{n+1} = [\![r]\!] = \downarrow$. Therefore we have the required result:

$$\forall P \in \mathfrak{P}.\ [\![s]\!]_{n+1} \in P \implies [\![r]\!] \in P.$$

□

Lemma 5.4.9. *Given a well-typed open relation $\mathcal{R}$ that is reflexive and has the two substitutivity properties from Lemma 5.4.5, and a well-typed closed relation $\mathcal{S}$ then:*

$$\textit{if } \mathcal{R} \textit{ restricted to closed terms is included in } \mathcal{S} \textit{ then } \mathcal{R} \subseteq \mathcal{S}^{\circ}.$$

Proof. Consider terms s and t, values or computations, such that $\overrightarrow{x_i : A_i} \vdash s\ \mathcal{R}\ t$. By reflexivity of R we know that for any values $\vdash \overrightarrow{v_i : A_i}$ we have $\vdash v_i\ \mathcal{R}^{\mathfrak{v}}\ v_i$. Therefore we can apply the substitutivity property of $\mathcal{R}$ to obtain:

$$\forall \vdash \overrightarrow{v_i : A_i}.\ \vdash s[\overrightarrow{v_i/x_i}]\ \mathcal{R}\ t[\overrightarrow{v_i/x_i}].$$

From here we can deduce by assumption that:

$$\forall \vdash \overrightarrow{v_i : A_i}.\ \vdash s[\overrightarrow{v_i/x_i}]\ \mathcal{S}\ t[\overrightarrow{v_i/x_i}]$$

so by the definition of open extension we have:

$$\overrightarrow{x_i : A_i} \vdash s\ \mathcal{S}^{\circ}\ t.$$

□

Lemma 5.4.10. *Given a $\mathfrak{P}$-simulation $\mathcal{R}$, its reflexive-transitive closure, $\mathcal{R}^*$ is also a $\mathfrak{P}$-simulation.*

Proof. We check all the conditions in the definition of $\mathfrak{P}$-simulation in turn:

1. $\vdash v\ \mathcal{R}^{*,\mathfrak{v}}_{\mathtt{unit}}\ w \implies v = w = \star$. Assume $\vdash v\ \mathcal{R}^{*,\mathfrak{v}}_{\mathtt{unit}}\ w$. The only closed value of type `unit` is $\star$ so $v = w = \star$.

2. $\vdash v\ \mathcal{R}^{*,\mathfrak{v}}_{\mathtt{nat}}\ w \implies v = w$. Assume $v\ \mathcal{R}^{*,\mathfrak{v}}_{\mathtt{nat}}\ w$. Then by the definition of reflexive-transitive closure there must exist a chain of values $v_1, \ldots, v_k$ such that $v_1 = v$ and $v_k = w$ and $v_{i-1}\ \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}}\ v_i$ for each $i > 1$. If $k = 1$ then $v = w$. If $k > 1$, we can use the fact that $\mathcal{R}$ is a simulation to deduce $v_{i-1} = v_i$ for each $i > 1$. So by transitivity $v = w$.

3. $\vdash s\ \mathcal{R}^{*,\mathfrak{c}}\ t \implies \forall P \in \mathfrak{P}.\ (\llbracket s \rrbracket \in P \implies \llbracket t \rrbracket \in P)$. Assume $s\ \mathcal{R}^{*,\mathfrak{c}}\ t$. There exists a chain of computations $s_1, \ldots, s_k$ such that $s_1 = s$ and $s_k = t$ and $s_{i-1}\ \mathcal{R}^{\mathfrak{c}}\ s_i$ for each $i > 1$. If $k = 1$ then $s = t$ so we have $\forall P \in \mathfrak{P}.\ (\llbracket s \rrbracket \in P \implies \llbracket t \rrbracket \in P)$ as required. If $k > 1$ then for each $i > 1$:

   $$\forall P \in \mathfrak{P}.\ (\llbracket s_{i-1} \rrbracket \in P \implies \llbracket s_i \rrbracket \in P).$$

   From here we can deduce the desired result by transitivity of implication.

4. $\vdash v\ \mathcal{R}^{*,\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ w \implies \forall \vdash u_1 : A_1, \ldots, \vdash u_n : A_n.\ v(u_1, \ldots, u_n)\ \mathcal{R}^{*,\mathfrak{c}}\ w(u_1, \ldots, u_n)$. Similar to the previous two cases. It uses the fact that $v\ \mathcal{R}\ u\ \mathcal{R}\ w$ implies $v\ \mathcal{R}^*\ w$.

□

Lemma 5.4.11. *Given a well-typed compatible relation $\mathcal{R}$, its reflexive-transitive closure $\mathcal{R}^*$ is also compatible.*

Proof. The proof is similar to the proof of Lemma 5.4.10 in that it expands $s\ \mathcal{R}^*\ t$ into a chain of terms related by $\mathcal{R}$. It also uses the fact that $\mathcal{R}$ compatible implies $\mathcal{R}$ is reflexive, Lemma 5.4.3. □

Lemma 5.4.12 (From [Las98])**.** *Given a well-typed closed relation $\mathcal{R}$ the following holds:*

if $\mathcal{R}^\circ$ is reflexive and symmetric, then $\mathcal{R}^{\mathcal{H}*}$ is symmetric.

Where S^* denotes the reflexive-transitive closure of a relation $\mathcal{S}$.

Proof. By examining the compatible refinement rules we can observe that for any relation $\mathcal{S}$:

$$\widehat{\mathcal{S}^{op}} = \widehat{\mathcal{S}}^{op}. \tag{B.17}$$

Since $\mathcal{R}^\circ$ is reflexive, $\mathcal{R}$ is also reflexive. Therefore we can apply Lemma 5.4.3 to deduce:

$$\mathcal{R}^\circ \subseteq \mathcal{R}^{\mathcal{H}} \tag{B.18}$$

and $\mathcal{R}^{\mathcal{H}}$ compatible.

Because $\mathcal{R}^{\mathcal{H}}$ is compatible, $\mathcal{R}^{\mathcal{H}*}$ is also compatible using Lemma 5.4.11. By definition of compatibility and compatible refinement we see that $\mathcal{R}^{\mathcal{H}*}$ compatible implies:

$$\widehat{\mathcal{R}^{\mathcal{H}*}} \subseteq \mathcal{R}^{\mathcal{H}*}. \tag{B.19}$$

Using the fact that $\mathcal{R}^{\circ}$ is symmetric, equations B.17, B.19 and B.18, and the fact that taking the reflexive-transitive closure and the converse of a relation are commutative operations we obtain:

$$\mathcal{R}^{\circ} \circ \widehat{\mathcal{R}^{\mathcal{H}*op}} = \mathcal{R}^{\circ op} \circ \widehat{\mathcal{R}^{\mathcal{H}*op}} = \mathcal{R}^{\circ op} \circ \widehat{\mathcal{R}^{\mathcal{H}*}}^{op} \subseteq$$

$$\mathcal{R}^{\circ op} \circ \mathcal{R}^{\mathcal{H}*op} \subseteq \mathcal{R}^{\mathcal{H}op} \circ \mathcal{R}^{\mathcal{H}*op} = \mathcal{R}^{\mathcal{H}op} \circ \mathcal{R}^{\mathcal{H}op*} = \mathcal{R}^{\mathcal{H}op*} = \mathcal{R}^{\mathcal{H}*op}.$$

This means that $\mathcal{R}^{\mathcal{H}*op}$ is a solution $\mathcal{S}$ to the inequation $\mathcal{R}^{\circ} \circ \widehat{\mathcal{S}} \subseteq \mathcal{S}$. So the relation $\mathcal{R}^{\mathcal{H}*op}$ is closed under the rules (HC) and (HV). But $\mathcal{R}^{\mathcal{H}}$ is the least relation closed under those rules. Therefore:

$$\mathcal{R}^{\mathcal{H}} \subseteq \mathcal{R}^{\mathcal{H}*op}.$$

Consider some terms s and t, values or computations, such that $s\ \mathcal{R}^{\mathcal{H}*}\ t$. Then there exists a sequence of terms $s_1, \ldots, s_n$ such that $s_1 = s$ and $s_n = t$ and $s_1\ \mathcal{R}^{\mathcal{H}}\ s_2\ \mathcal{R}^{\mathcal{H}}\ \ldots\ \mathcal{R}^{\mathcal{H}}\ s_n$. Therefore $s_1\ \mathcal{R}^{\mathcal{H}*op}\ s_2\ \mathcal{R}^{\mathcal{H}*op}\ \ldots\ \mathcal{R}^{\mathcal{H}*op}\ s_n$.

From here we can deduce $s\ \mathcal{R}^{\mathcal{H}*op}\ t$, which means $t\ \mathcal{R}^{\mathcal{H}*}\ s$. So $\mathcal{R}^{\mathcal{H}*}$ is symmetric as required. □

Appendix C

Proofs about Logical Equivalence

Proposition 6.3.1. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations:*

1. *Applicative $\mathfrak{P}$-similarity, $\precsim$, coincides with the logical preorder induced by the logic $\mathcal{V}^+$, $\sqsubseteq_{\mathcal{V}^+}$. Therefore, the open extension of $\sqsubseteq_{\mathcal{V}^+}$ is compatible.*

2. *Applicative $\mathfrak{P}$-bisimilarity, $\sim$, coincides with the logical equivalence induced by the logic $\mathcal{V}$, $\equiv_{\mathcal{V}}$. Therefore, the open extension of $\equiv_{\mathcal{V}}$ is compatible.*

Proof. 1. Consider two arbitrary closed computations s and t. Since similarity is the greatest simulation:

$$s \precsim^{\mathfrak{c}} t \quad \text{iff} \quad \forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P$$

$$s \sqsubseteq_{\mathcal{V}^+} t \quad \text{iff} \quad \forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P.$$

So we can see that $s \precsim^{\mathfrak{c}} t \iff s \sqsubseteq_{\mathcal{V}^+} t$ as required.

For values v and u assume first that $v \precsim^{\mathfrak{v}}_A u$. We need to prove $v \sqsubseteq_{\mathcal{V}^+} u$, that is:

$$\forall \phi : A \in \mathcal{V}^+.\ (v \models_{\mathcal{V}^+} \phi \implies u \models_{\mathcal{V}^+} \phi).$$

We proceed by a case distinction on the type A.

Case $A = \texttt{unit}$**.** The only formulas of this type are $true$, the empty conjunction, and $false$, the empty disjunction. All values satisfy $true$ so in this case we are done. No values satisfy $false$ so the implication above holds trivially.

Case $A = \texttt{nat}$**.** From the definition of simulation we know that $v = u$. We continue by induction on the formula ϕ.

If $\phi = \{m\}$, assume $v \models_{\mathcal{V}^+} \{m\}$. By definition of satisfaction this means $v = \overline{m}$ so $u = \overline{m}$. Therefore $u \models_{\mathcal{V}^+} \{m\}$ as required.

If $\phi = \wedge_{i \in I} \phi_i$ **or** $\phi = \vee_{i \in I} \phi_i$, the result follows from the induction hypothesis for ϕ_i.

Case $A = \neg(B_1, \ldots, B_n)$**.** From $v \precsim^{\mathfrak{v}}_A u$ we know that:

$$\forall \vdash w_1 : A_1, \ldots, \vdash w_n : A_n.\ \forall P \in \mathfrak{P}.\ [\![v\ (\overrightarrow{w_i})]\!] \in P \implies [\![u\ (\overrightarrow{w_i})]\!] \in P.$$

We proceed by induction on ϕ.

If $\phi = (w_1, \ldots, w_n) \mapsto P$, assume $v \models_{\mathcal{V}^+} (w_1, \ldots, w_n) \mapsto P$, which means $[\![v\ (\overrightarrow{w_i})]\!] \in P$. Therefore, by assumption $[\![u\ (\overrightarrow{w_i})]\!] \in P$ so $u \models_{\mathcal{V}^+} (w_1, \ldots, w_n) \mapsto P$ as required.

If $\phi = \wedge_{i \in I} \phi_i$ **or** $\phi = \vee_{i \in I} \phi_i$, the result follows from the induction hypothesis.

Now assume that for values v and u, $v \sqsubseteq_{\mathcal{V}^+} u$. To show $v \precsim^{\mathfrak{v}}_A u$ we proceed by a case distinction on A.

Case $A = \texttt{unit}$**.** The only closed value of type `unit` is $\star$ so $v = u = \star$. Since $\precsim$ is the greatest simulation, this is enough to establish $v \precsim^{\mathfrak{v}}_{\texttt{unit}} u$.

Case $A = \texttt{nat}$**.** Since v is closed $v = \overline{n}$ for some $n \in \mathbb{N}$. Therefore $v \models_{\mathcal{V}^+} \{n\}$. Since $v \sqsubseteq_{\mathcal{V}^+} u$ we have that $u \models_{\mathcal{V}^+} \{n\}$ so $u = \overline{n} = v$ as required.

Case $A = \neg(B_1, \ldots, B_n)$**.** We need to prove that:

$$\forall \vdash w_1 : A_1, \ldots, \vdash w_n : A_n.\ \forall P \in \mathfrak{P}.\ [\![v\ (\overrightarrow{w_i})]\!] \in P \implies [\![u\ (\overrightarrow{w_i})]\!] \in P.$$

Assume $[\![v\ (\overrightarrow{w_i})]\!] \in P$. Then $v \models_{\mathcal{V}^+} (w_1, \ldots, w_n) \mapsto P$ so $w \models_{\mathcal{V}^+} (w_1, \ldots, w_n) \mapsto P$. Therefore $[\![w\ (\overrightarrow{w_i})]\!] \in P$ as required.

So we have proved $v \precsim^{\mathfrak{v}}_A u \iff v \sqsubseteq_{\mathcal{V}^+} u$.

2. For computations we know that $s \sim^{\mathfrak{c}} t$ if and only if:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \iff [\![t]\!] \in P.$$

So we can see this is equivalent to $s \equiv_{\mathcal{V}} t$.

For values, assume $v \equiv_{\mathcal{V}} u$, that is:

$$\forall \phi : A \in \mathcal{V}.\ (v \models_{\mathcal{V}} \phi \iff u \models_{\mathcal{V}} \phi). \tag{C.1}$$

We need to prove $v \sim^{\mathfrak{v}}_A u$. We proceed by a case split on the type A. The proof is the same as in point 1, except that in the case $A = \neg(B_1, \ldots, B_n)$ we need to prove an equivalence. This is done by using the fact that equation C.1 is now an equivalence.

Now assume $v \sim^{\mathfrak{v}}_A u$. We need to prove $v \equiv_{\mathcal{V}} u$, that is:

$$\forall \phi : A \in \mathcal{V}.\ (v \models_{\mathcal{V}} \phi \iff u \models_{\mathcal{V}} \phi).$$

As in point 1, we proceed by a case distinction on the type A.

In case $A = \texttt{unit}$ we have new formulas apart from $true$ and $false$ due to the addition of negation. However, they are all semantically equivalent to either $true$ or $false$, so we are done.

In cases $A = \texttt{nat}$ and $A = \neg(B_1, \ldots, B_n)$ proceed by induction on $\phi : A$ as before. The $\implies$ direction of the proof is the same. The $\impliedby$ direction can be obtained from the previous one by choosing $\phi = \neg\varphi$. The inductions have an additional case:

If $\phi = \neg\phi'$. By the induction hypothesis we know:

$$v \models_{\mathcal{V}} \phi' \iff u \models_{\mathcal{V}} \phi'$$

which is equivalent to

$$v \not\models_{\mathcal{V}} \phi' \iff u \not\models_{\mathcal{V}} \phi'$$

which is in turn equivalent to

$$v \models_{\mathcal{V}} \neg\phi' \iff u \models_{\mathcal{V}} \neg\phi'.$$

This is what we had to prove.

□

Theorem 6.3.4. *Given a decomposable set $\mathfrak{P}$ of Scott-open observations, the logics $\mathcal{F}$ and $\mathcal{V}$ are equi-expressive.*

Proof. We need to prove the same statements as in Theorem 6.3.3, where $\mathcal{V}^+$ is replaced by $\mathcal{V}$ and $\mathcal{F}^+$ is replaced by $\mathcal{F}$. We will point out where the proofs need to be modified.

Statement 1. The proof for computation formulas $P \in \mathfrak{P}$ remains the same because these formulas do not change when adding negation.

For values, we prove by induction on the derivation of $\phi : A$ the following property:

$$\Phi(\phi, A) = (\phi : A \implies (\forall \vdash v : A.\ v \models_{\mathcal{F}} \phi \iff v \models_{\mathcal{V}} \phi^\flat)).$$

All the cases from the proof of Theorem 6.3.3 stay the same, but now we have an additional case (NEG).

In this case, $\phi = \neg\phi'$. Assume $\phi : A$. Then we know $\phi' : A$ so we can apply the induction hypothesis to get:

$$\forall \vdash v : A.\ v \models_{\mathcal{F}} \phi' \iff v \models_{\mathcal{V}} \phi'^\flat.$$

This is equivalent to:

$$\forall \vdash v : A.\ v \models_{\mathcal{F}} \neg\phi' \iff v \models_{\mathcal{V}} \neg\phi'^\flat$$

which is what we had to prove.

Statement 2. For computation formulas the equivalence is proved the same as in Theorem 6.3.3.

For value formulas we proceed by induction on the type A, as in the proof of Theorem 6.3.3. The case $A = \texttt{nat}$ stays the same. In the $A = \texttt{unit}$ case, we now have more formulas than *true* and $false$ because of the addition of negation. However, all these new formulas are semantically equivalent to *true* and $false$, so their semantics does not change when translated. Therefore, the equivalence we need to prove is true. In the case $A = \neg(B_1, \ldots, B_n)$ we do an induction on the formula ϕ.

We observe that $(\sqsubseteq_{\mathcal{F}}) = (\equiv_{\mathcal{F}})$ because $\mathcal{F}$ contains negation. The proof of this goes as follows:

$$\forall \phi \in \mathcal{F}.\ v \models_{\mathcal{F}} \phi \implies u \models_{\mathcal{F}} \phi$$

implies that

$$\forall \phi' \in \mathcal{F}.\ v \models_{\mathcal{F}} \neg\phi' \implies u \models_{\mathcal{F}} \neg\phi'$$

so

$$\forall \phi' \in \mathcal{F}.\ u \models_{\mathcal{F}} \phi' \implies v \models_{\mathcal{F}} \phi'.$$

Using $(\sqsubseteq_{\mathcal{F}}) = (\equiv_{\mathcal{F}})$, we see that $\sqsubseteq_{\mathcal{F}}$ is compatible. Therefore, the proof of the case $\phi = (w_1, \ldots, w_n) \mapsto P$ is the same as in Theorem 6.3.3. The cases $\phi = \vee_{i\in I}\varphi_i$ and $\phi = \wedge_{i\in I}\varphi_i$ remain unchanged.

There is one new case, namely $\phi = \neg\phi'$. By the induction hypothesis for ϕ' we know that:

$$\forall \vdash v : \neg(B_1, \ldots, B_n).\ v \models_{\mathcal{V}} \phi' \iff v \models_{\mathcal{F}} \phi'^{\sharp}.$$

This is equivalent to:

$$\forall \vdash v : \neg(B_1, \ldots, B_n).\ v \models_{\mathcal{V}} \neg\phi' \iff v \models_{\mathcal{F}} \neg\phi'^{\sharp}$$

which is what we had to prove. □

Appendix D

Proofs about Contextual Equivalence

Proposition 7.1.3. *The contextual preorder $\sqsubseteq_{ctx}$ is a preorder, and is moreover compatible and $\mathfrak{P}$-adequate. Thus, it is the greatest compatible and $\mathfrak{P}$-adequate preorder.*

Proof. First prove that $\sqsubseteq_{ctx} = \bigcup \mathbb{CA}$ is a preorder. To prove reflexivity, we show that the open identity relation, $\mathcal{I}$, is in $\mathbb{CA}$. From the compatibility rules we can see $\mathcal{I}$ is compatible. Given $\vdash s\ \mathcal{I}\ s$ it follows that $(\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![s]\!] \in P)$ so $\mathcal{I}$ is adequate. Hence, $\mathcal{I} \in \mathbb{CA}$, as required.

To show transitivity it suffices to show that the composition of relations in $\mathbb{CA}$ is itself in $\mathbb{CA}$. Consider two relations $\mathcal{R}$ and $\mathcal{S}$ which are compatible and adequate.

We can show that $\mathcal{S} \circ \mathcal{R}$ is adequate. Consider $\vdash s\ \mathcal{R}^{\mathfrak{c}}\ t\ \mathcal{S}^{\mathfrak{c}}\ r$. Since $\mathcal{R}$ and $\mathcal{S}$ are adequate we know that:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P$$

$$\forall P \in \mathfrak{P}.\ [\![t]\!] \in P \implies [\![r]\!] \in P.$$

Therefore:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![r]\!] \in P$$

which means $\mathcal{S} \circ \mathcal{R}$ is adequate.

To prove $\mathcal{S} \circ \mathcal{R}$ is compatible we check each of the rules in the definition of compatibility (Definition 5.3.3) in turn. Since $\mathcal{R}$ and $\mathcal{S}$ are compatible we know $\Gamma \vdash x\ \mathcal{R}^{\mathfrak{v}}_A\ x$ and $\Gamma \vdash x\ \mathcal{S}^{\mathfrak{v}}_A\ x$ so $\Gamma \vdash x\ (\mathcal{S} \circ \mathcal{R})^{\mathfrak{v}}_A\ x$. Therefore, $\mathcal{S} \circ \mathcal{R}$ satisfies (COMP1). Rules (COMP2), (COMP4) and (COMP9) are proved similarly.

Consider $\Gamma, \overrightarrow{x_i : A_i} \vdash s\ \mathcal{R}^{\mathfrak{c}}\ s'\ \mathcal{S}^{\mathfrak{c}}\ s''$. Then by compatibility of $\mathcal{R}$ and $\mathcal{S}$ we know that:

$$\Gamma \vdash \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).s\ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).s'\ \mathcal{S}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).s''.$$

So

$$\Gamma \vdash \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).s\ (\mathcal{S} \circ \mathcal{R})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda(\overrightarrow{x_i}){:}(\overrightarrow{A_i}).s''.$$

Therefore $\mathcal{S} \circ \mathcal{R}$ satisfies (COMP3). Rule (COMP5) is proved similarly.

Consider $\Gamma, x : \neg(\overrightarrow{A_i}) \vdash v \ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \ v' \ \mathcal{S}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \ v''$ and $\Gamma \vdash w_i \ \mathcal{R}^{\mathfrak{v}}_{A_i} \ w_i' \ \mathcal{S}^{\mathfrak{v}}_{A_i} \ w_i''$ for each i. By compatibility of $\mathcal{R}$ and $\mathcal{S}$ we infer that:

$$\Gamma \vdash (\mu x.v)(\overrightarrow{w_i}) \ \mathcal{R}^{\mathfrak{c}} \ (\mu x.v)(\overrightarrow{w_i'}) \ \mathcal{S}^{\mathfrak{c}} \ (\mu x.v)(\overrightarrow{w_i''})$$

so

$$\Gamma \vdash (\mu x.v)(\overrightarrow{w_i}) \ (\mathcal{S} \circ \mathcal{R})^{\mathfrak{c}} \ (\mu x.v)(\overrightarrow{w_i''}).$$

Therefore, $\mathcal{S} \circ \mathcal{R}$ satisfies (COMP7). Proving rules (COMP6), (COMP8) and (COMP10) are satisfied is similar.

Now we prove that $\sqsubseteq_{ctx}$ is compatible. We have already shown $\sqsubseteq_{ctx}$ is reflexive so rules (COMP1), (COMP2), (COMP4) and (COMP9) are satisfied.

Suppose $\Gamma, \overrightarrow{x_i : A_i} \vdash s \ (\sqsubseteq_{ctx})^{\mathfrak{c}} \ t$. Then there exists a relation $\mathcal{R} \in \mathbb{CA}$ such that $\Gamma, \overrightarrow{x_i : A_i} \vdash s \ \mathcal{R}^{\mathfrak{c}} \ t$. By compatibility of $\mathcal{R}$ we know that:

$$\Gamma \vdash \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.s \ \mathcal{R}^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \ \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.t$$

so since $\mathcal{R} \subseteq (\sqsubseteq_{ctx})$ we have:

$$\Gamma \vdash \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.s \ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})} \ \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.t.$$

Therefore, $\sqsubseteq_{ctx}$ satisfies rule (COMP3). Similarly, we can prove $\sqsubseteq_{ctx}$ satisfies (COMP5).

Since we have proved $\sqsubseteq_{ctx}$ is a preorder, we can apply Lemma 5.3.4 to deduce that the compatibility clauses (COMP6), (COMP7), (COMP8) and (COMP10) are equivalent to their single-premise versions. We then use the same reasoning as before to show that these single-premise rules are satisfied.

Now show that $\sqsubseteq_{ctx}$ is adequate. Consider $\vdash s \ (\sqsubseteq_{ctx})^{\mathfrak{c}} \ t$. Then there exists a relation $\mathcal{R} \in \mathbb{CA}$ such that $\vdash s \ \mathcal{R}^{\mathfrak{c}} \ t$. So by adequacy of $\mathcal{R}$ we have:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P$$

as required.

Therefore, we have shown that $\sqsubseteq_{ctx}$ is a compatible adequate preorder so we are done. □

Proposition 7.1.5. *Contextual equivalence is the intersection of the contextual preorder with its converse:*

$$(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}.$$

Proof. We prove each inclusion in turn.

"$\subseteq$". Consider two terms, values or computations, $s \equiv_{ctx} t$. By definition of $\equiv_{ctx}$ there exists a compatible and biadequate relation $\mathcal{R}$ such that $s \ \mathcal{R} \ t$. Since $\mathcal{R}$ is biadequate, it is also adequate, so $\mathcal{R} \in \mathbb{CA}$. Thus, $\mathcal{R} \subseteq (\sqsubseteq_{ctx})$ and $s \sqsubseteq_{ctx} t$.

Notice that $(\sqsubseteq_{ctx})^{op} = \bigcup\{\mathcal{S}^{op} \mid \mathcal{S} \in \mathbb{CA}\}$. If a relation $\mathcal{S}$ is compatible then we see by the definition of compatibility that $\mathcal{S}^{op}$ is also compatible. So $(\sqsubseteq_{ctx})^{op}$ is the union of all relations $\mathcal{S}$ that are compatible and have the property:

$$\forall s', t'.\ \vdash s' \ \mathcal{R}^{\mathfrak{c}} \ t' \implies \forall P \in \mathfrak{P}.\ [\![t']\!] \in P \implies [\![s']\!] \in P.$$

Because $\mathcal{R}$ is biadequate it has the above property. So $\mathcal{R} \subseteq (\sqsubseteq_{ctx})^{op}$. Therefore, $s \ (\sqsubseteq_{ctx})^{op} \ t$. We have shown that $(s, t) \in (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$ as required.

"$\supseteq$". From Proposition 7.1.3 we know $\sqsubseteq_{ctx}$ is compatible so $(\sqsubseteq_{ctx})^{op}$ is also compatible. From the definition of compatibility we can see that the intersection of two compatible relations is also compatible so $(\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$ is compatible.

Let s and t be two closed computations such that $(s,t) \in (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$. Then by adequacy of $\sqsubseteq_{ctx}$ (Propostion 7.1.3) we have:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P$$

$$\forall P \in \mathfrak{P}.\ [\![t]\!] \in P \implies [\![s]\!] \in P$$

which means that $(\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op}$ is biadequate.

Therefore $(\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op} \in \mathbb{CAS}$ so $(\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op} \subseteq (\equiv_{ctx})$. □

Theorem 7.2.2. *Consider a decomposable set of Scott-open observations $\mathfrak{P}$ that is consistent. Then:*

1. *The open extension of applicative $\mathfrak{P}$-similarity, $\precsim^{\circ}$, coincides with the contextual preorder, $\sqsubseteq_{ctx}$.*

2. *The open extension of applicative $\mathfrak{P}$-bisimilarity, $\sim^{\circ}$, coincides with contextual equivalence, $\equiv_{ctx}$.*

Proof. **We first show** $(\precsim^{\circ}) = (\sqsubseteq_{ctx})$. We have shown in Section 7.2 that $\precsim^{\circ}$ is included in $\sqsubseteq_{ctx}$.

Now we need to show $(\sqsubseteq_{ctx}) \subseteq (\precsim^{\circ})$. We first show that $\sqsubseteq_{ctx}$ restricted to closed terms is included in $\precsim$, then extend this to open terms. To do this, we show $\sqsubseteq_{ctx}$ restricted to closed terms is a simulation by checking it satisfies the four conditions in the definition of simulation.

1. Assume $\vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\mathtt{unit}}\ u$. The only closed value of type `unit` is $\star$ so $v = u = \star$ as required.

2. Assume $\vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\mathtt{nat}}\ u$. This is shown in Section 7.2.

3. Assume $\vdash s\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ t$. Because $\sqsubseteq_{ctx}$ is adequate (Proposition 7.1.3) we have the desired result:
$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P.$$

4. Assume $\vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(A_1,\ldots,A_n)}\ u$. Consider arbitrary values $\vdash w_1 : A_1, \ldots, \vdash w_n : A_n$. Because $\sqsubseteq_{ctx}$ is a preorder it is reflexive so $w_i\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A_i}\ w_i$ for each i. We know $\sqsubseteq_{ctx}$ is compatible so we obtain:
$$v(w_1, \ldots, w_n)\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ u(w_1, \ldots, w_n)$$
as required.

So we have established $(\sqsubseteq_{ctx}) \subseteq (\precsim)$ for closed terms. Now we need to prove $(\sqsubseteq_{ctx}) \subseteq (\precsim^{\circ})$ in general. To do this, we consider computations and each type of value separately. The cases for natural numbers and computations are shown in Section 7.2 so we only show the cases for type `unit` and function values.

If $\overrightarrow{x_j : A_j} \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\mathtt{unit}}\ w$ then each of v and w can either be $\star$ or a variable x_i. In any case, because $\star$ is the only closed value of type `unit`, we know that:

$$\forall(\vdash \overrightarrow{u_i : A_i}).\ v[\overrightarrow{u_i/x_i}] = w[\overrightarrow{u_i/x_i}] = \star.$$

This means that:

$$\forall(\vdash \overrightarrow{u_i : A_i}).\ \vdash v[\overrightarrow{u_i/x_i}] \precsim^{\mathfrak{v}}_{\mathtt{unit}} w[\overrightarrow{u_i/x_i}]$$

so by the definition of open extension we have that $\overrightarrow{x_j : A_j} \vdash v \precsim^{\circ,\mathfrak{v}}_{\mathtt{unit}} w$ as required.

If $\overrightarrow{x_j : A_j} \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(B_1,\ldots,B_n)}\ w$ then by compatibility and reflexivity of $\sqsubseteq_{ctx}$ we know that

$$\forall(\vdash \overrightarrow{u_i : B_i}).\ \overrightarrow{x_j : A_j} \vdash v(\overrightarrow{u_i})\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ w(\overrightarrow{u_i})$$

and again by compatibility:

$$\forall(\vdash \overrightarrow{u_i : B_i}).\ \vdash \lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).v(\overrightarrow{u_i})\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_j})}\ \lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).w(\overrightarrow{u_i}).$$

Using $(\sqsubseteq_{ctx}) \subseteq (\precsim)$ we can deduce:

$$\forall(\vdash \overrightarrow{u_i : B_i}).\ \vdash \lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).v(\overrightarrow{u_i})\ \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{A_j})}\ \lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).w(\overrightarrow{u_i})$$

so by the definition of $\precsim$ we know that

$$\forall(\vdash \overrightarrow{p_j : A_j}).\ \forall(\vdash \overrightarrow{u_i : B_i}).\ \vdash (\lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).v(\overrightarrow{u_i}))(\overrightarrow{p_j})\ \precsim^{\mathfrak{c}}\ (\lambda(\overrightarrow{x_j}){:}(\overrightarrow{A_j}).w(\overrightarrow{u_i}))(\overrightarrow{p_j}).$$

From Lemma 5.2.4 we know reduction preserves similarity so:

$$\forall(\vdash \overrightarrow{p_j : A_j}).\ \forall(\vdash \overrightarrow{u_i : B_i}).\ \vdash v[\overrightarrow{p_j/x_j}]\ (\overrightarrow{u_i})\ \precsim^{\mathfrak{c}}\ w[\overrightarrow{p_j/x_j}]\ (\overrightarrow{u_i}).$$

By the definition of similarity for function values this means that:

$$\forall(\vdash \overrightarrow{p_j : A_j}).\ \vdash v[\overrightarrow{p_j/x_j}]\ \precsim^{\mathfrak{v}}_{\neg(\overrightarrow{B_i})}\ w[\overrightarrow{p_j/x_j}]$$

so by the definition of open extension

$$\overrightarrow{x_j : A_j} \vdash v\ \precsim^{\circ,\mathfrak{v}}_{\neg(\overrightarrow{B_i})}\ w$$

as required.

Now show that $(\sim^{\circ}) = (\equiv_{ctx})$**.** We have shown that $(\precsim^{\circ}) = (\sqsubseteq_{ctx})$ so:

$$(\precsim^{op})^{\circ} = (\precsim^{\circ})^{op} = (\sqsubseteq_{ctx})^{op}$$

because taking the converse of a relation and its open extension are commutative operations. From Proposition 7.1.5 we know that:

$$(\equiv_{ctx}) = (\sqsubseteq_{ctx}) \cap (\sqsubseteq_{ctx})^{op} = (\precsim^{\circ}) \cap (\precsim^{op})^{\circ}.$$

We can show that $(\precsim^{\circ}) \cap (\precsim^{op})^{\circ} = ((\precsim) \cap (\precsim^{op}))^{\circ}$. The equation:

$$\overrightarrow{x_i : A_i} \vdash s \precsim^{\circ} t \quad \text{and} \quad \overrightarrow{x_i : A_i} \vdash s(\precsim^{op})^{\circ}t$$

is equivalent to

$$\forall(\vdash \overrightarrow{u_i : A_i}).\ \vdash s[\overrightarrow{u_i/x_i}] \precsim t[\overrightarrow{u_i/x_i}] \text{ and } \vdash s[\overrightarrow{u_i/x_i}] \precsim^{op} t[\overrightarrow{u_i/x_i}]$$

which in turn is equivalent to $(s,t) \in ((\precsim) \cap (\precsim^{op}))^\circ$, as required.

Therefore, we know that:

$$(\equiv_{ctx}) = ((\precsim) \cap (\precsim^{op}))^\circ$$

and by Proposition 5.2.3 we know that:

$$(\equiv_{ctx}) = ((\precsim) \cap (\precsim^{op}))^\circ = (\sim)^\circ$$

which is what we had to prove. □

Lemma 7.3.5. *Contextual preorder defined with contexts, $\leq_{ctx}$, is a compatible and adequate preorder. Hence, it is included in contextual preorder defined coinductively, $\sqsubseteq_{ctx}$.*

Proof. First prove $\leq_{ctx}$ is a preorder. Given terms $\Gamma \vdash s = t$ we know that for any context C, $[\![C[s]]\!] = [\![C[t]]\!]$. Therefore $\Gamma \vdash s \leq_{ctx} t$, so $\leq_{ctx}$ is reflexive. Because implication is transitive we can see that $\leq_{ctx}$ is also transitive.

To prove $\leq_{ctx}$ is adequate consider closed computations $\emptyset \vdash s\ (\leq_{ctx})^{\mathfrak{c}}\ t$. Then by definition of $\leq_{ctx}$, choosing $C^{\mathfrak{c}}_{\mathfrak{c}} = [-]^{\mathfrak{c}}$, we know that:

$$\forall P \in \mathfrak{P}.\ [\![s]\!] \in P \implies [\![t]\!] \in P$$

as required.

To prove $\leq_{ctx}$ is compatible show that is satisfies each compatibiliy rule. Rules (COMP1), (COMP2), (COMP4) and (COMP9) are satisfied by reflexivity.

For rule (COMP3) assume $\Gamma, \overrightarrow{x_i : A_i} \vdash s\ (\leq_{ctx})^{\mathfrak{c}}\ t$. Consider an arbitrary context $C'^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma \vdash \neg(\overrightarrow{A_i})) \Rightarrow (\emptyset \vdash)$. Now consider the context:

$$C^{\mathfrak{c}}_{\mathfrak{c}} = C'^{\mathfrak{v}}_{\mathfrak{c}}[\lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.[-]^{\mathfrak{c}}] : (\Gamma, \overrightarrow{x_i : A_i} \vdash) \Rightarrow (\emptyset \vdash).$$

Instantiate the assumption that s and t are in the contextual preorder with $C^{\mathfrak{c}}_{\mathfrak{c}}$ to deduce that:

$$\forall P \in \mathfrak{P}.\ [\![C'^{\mathfrak{v}}_{\mathfrak{c}}[\lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.[s]^{\mathfrak{c}}]]\!] \in P \implies [\![C'^{\mathfrak{v}}_{\mathfrak{c}}[\lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.[t]^{\mathfrak{c}}]]\!] \in P.$$

Then we know $\Gamma \vdash \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.s\ (\leq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.t$ which is what we had to prove. Rule (COMP5) can be proved similarly choosing:

$$C^{\mathfrak{v}}_{\mathfrak{c}} = C'^{\mathfrak{v}}_{\mathfrak{c}}[\mathtt{succ}([-]^{\mathfrak{v}})].$$

Using the fact that $\leq_{ctx}$ is a preorder we can apply Lemma 5.3.4 to replace the four compatibility rules that we still need to prove with their single-premise versions. Proving these single-premise rules hold is analogous to proving (COMP3). As an example, we prove rule (COMP7L).

Assume $\Gamma, x : \neg(\overrightarrow{A_i}) \vdash v\ (\leq_{ctx})^{\mathfrak{v}}_{\neg(\overrightarrow{A_i})}\ v'$ and $\Gamma \vdash \overrightarrow{w_i : A_i}$. Consider an arbitrary context $C'^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma \vdash) \Rightarrow (\emptyset \vdash)$ and the context:

$$C^{\mathfrak{v}}_{\mathfrak{c}} = C'^{\mathfrak{c}}_{\mathfrak{c}}[(\mu x.[-]^{\mathfrak{v}})(\overrightarrow{w_i})].$$

Using the context typing rules (VV-ID), (VC-MUL) and Lemma 7.3.3 we can deduce that:

$$C^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma, x : \neg(\overrightarrow{A_i}) \vdash) \Rightarrow (\emptyset \vdash).$$

Therefore we can apply the assumption about v and v' to get:

$$\forall P \in \mathfrak{P}.\ [\![C'^{\mathfrak{c}}_{\mathfrak{c}}[(\mu x.[v]^{\mathfrak{v}})(\overrightarrow{w_i})]]\!] \in P \implies [\![C'^{\mathfrak{c}}_{\mathfrak{c}}[(\mu x.[v']^{\mathfrak{v}})(\overrightarrow{w_i})]]\!] \in P$$

which means $\Gamma \vdash (\mu x.v)(\overrightarrow{w_i})\ (\leq_{ctx})^{\mathfrak{c}}\ (\mu x.v')(\overrightarrow{w_i})$ as required.

We know that $\sqsubseteq_{ctx}$ is the greatest compatible and adequate relation (Lemma 7.1.3) and we have shown $\leq_{ctx}$ is compatible and adequate. Therefore $(\leq_{ctx}) \subseteq (\sqsubseteq_{ctx})$. □

Lemma 7.3.6. *Contextual preorder defined coinductively, $\sqsubseteq_{ctx}$, is closed under program contexts, that is:*

1. *If $\Gamma' \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A}\ u$ and $C^{\mathfrak{v}}_{\mathfrak{v}} : (\Gamma' \vdash A) \Rightarrow (\Gamma \vdash B)$ then $\Gamma \vdash C^{\mathfrak{v}}_{\mathfrak{v}}[v]\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{B}\ C^{\mathfrak{v}}_{\mathfrak{v}}[u]$.*
   And the analogous statement for $C^{\mathfrak{v}}_{\mathfrak{c}}$.
2. *If $\Gamma' \vdash s\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ t$ and $C^{\mathfrak{c}}_{\mathfrak{c}} : (\Gamma' \vdash) \Rightarrow (\Gamma \vdash)$ then $\Gamma \vdash C^{\mathfrak{c}}_{\mathfrak{c}}[s]\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ C^{\mathfrak{c}}_{\mathfrak{c}}[t]$.*
   And the analogous statement for $C^{\mathfrak{c}}_{\mathfrak{v}}$.

Proof. By induction on the typing derivation of C. The two base cases (VV-ID) and (CC-ID) follow from the assumptions $\Gamma' \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A}\ u$ and $\Gamma' \vdash s\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ t$ respectively.

In the case (VV-LBD):

$$C^{\mathfrak{v}}_{\mathfrak{v}} = \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.C'^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash A) \Rightarrow (\Gamma \vdash \neg(\overrightarrow{A_i}))$$

$$\text{where } C'^{\mathfrak{v}}_{\mathfrak{c}} : (\Gamma' \vdash A) \Rightarrow (\Gamma, x : \neg(\overrightarrow{A_i}) \vdash).$$

Assume $\Gamma' \vdash v\ (\sqsubseteq_{ctx})^{\mathfrak{v}}_{A}\ u$. Then by induction hypothesis for $C'^{\mathfrak{v}}_{\mathfrak{c}}$ we know that:

$$\Gamma, x : \neg(\overrightarrow{A_i}) \vdash C'^{\mathfrak{v}}_{\mathfrak{c}}[v]\ (\sqsubseteq_{ctx})^{\mathfrak{c}}\ C'^{\mathfrak{v}}_{\mathfrak{c}}[u]$$

so by compatibility of $\sqsubseteq_{ctx}$, rule (COMP3) we can deduce that:

$$\Gamma \vdash \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.C'^{\mathfrak{v}}_{\mathfrak{c}}[v]\ (\sqsubseteq_{ctx})^{\mathfrak{c}}_{\neg(\overrightarrow{A_i})}\ \lambda\overrightarrow{x_i}{:}\overrightarrow{A_i}.C'^{\mathfrak{v}}_{\mathfrak{c}}[u]$$

which is what we had to prove. Cases (CV-LBD) and (VV-NAT) are analogous.

For the remaining cases, we use the fact that $\sqsubseteq_{ctx}$ is a preorder, so the single-premise compatibility rules from Lemma 5.3.4 hold. The proof then proceeds similarly to the proof of (VV-LBD): apply the induction hypothesis then use one of the single-premise compatibility rules. □