Relational Reasoning for Effects and Handlers

Craig McLaughlin

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2020

Abstract

This thesis studies relational reasoning techniques for FRANK, a strict functional language supporting algebraic effects and their handlers, within a general, formalised approach for completely characterising observational equivalence.

Algebraic effects and handlers are an emerging paradigm for representing computational effects where primitive operations, which give rise to an effect, are primary, and given semantics through their interpretation by effect handlers. FRANK is a novel point in the design space because it recasts effect handling as part of a generalisation of call-by-value function application. Furthermore, FRANK generalises unary effect handlers to the *n*-ary notion of *multihandlers*, supporting more elegant expression of certain handlers.

There have been recent efforts to develop sound reasoning principles, with respect to observational equivalence, for languages supporting effects and handlers. Such techniques support powerful equational reasoning about code, such as substitution of equivalent sub-terms ('equals for equals') in larger programs. However, few studies have considered a complete characterisation of observational equivalence, and its implications for reasoning techniques. Furthermore, there has been no account of reasoning principles for FRANK programs.

Our first contribution is a formal reconstruction of a general proof technique, *triangulation*, for proving completeness results for observational equivalence. The technique brackets observational equivalence between two structural relations, a *logical* and an *applicative* notion. We demonstrate the triangulation proof method for a pure simply-typed λ-calculus. We show that such results are readily formalisable in an implementation of type theory, specifically AGDA, using state-of-the-art technology for dealing with syntaxes with binding.

Our second contribution is a calculus, ELLA, capturing the essence of FRANK's novel design. In particular, ELLA supports binary handlers and generalises function application to incorporate effect handling. We extend our triangulation proof technique to this new setting, completely characterising observational equivalence for this calculus. We report on our partial progress in formalising our extension to ELLA in AGDA.

Our final contribution is the application of sound reasoning principles, inspired by existing literature, to a variety of ELLA programs, including a proof of associativity for a canonical pipe multihandler. Moreover, we show how leveraging completeness leads, in certain instances, to simpler proofs of observational equivalence.

Acknowledgements

I thank my supervisors, James McKinna and Ian Stark, for their advice, guidance, enthusiasm and encouragement throughout my PhD. I appreciate the time they spent discussing my work and providing thoughtful analysis and answers to my numerous questions. Despite personal and professional hardships, they continued steadfastly to provide invaluable supervision, both academically and administratively; including securing last-minute funding to help me over the finish line. Their influence will have an everlasting impact on my approach to research problems.

I am grateful to Sam Lindley for the role he has played during my studies at Edinburgh. In particular, his suggestion to work on FRANK during a pre-PhD internship proved to have a significant impact on my dissertation topic. Through FRANK, I was able to collaborate with Sam and others, extending my research horizons beyond what they might otherwise have been. Sam has consistently provided valuable career advice, and I very much enjoyed our discussions, both professional and otherwise.

Ohad Kammar has been an inspiring source of enthusiasm and positive energy, consistently appearing at the right moments to inject shots of adrenaline. Ohad provided crucial financial support during the final stages of thesis write-up, enabling me to attend Shonan Meeting No. 146. Yet another source of wisdom, I thank Ohad for supporting me at such a critical time, and always having time to discuss research ideas.

I am grateful to my examiners, John Longley and Robert Atkey, for reading my dissertation so thoroughly, and for the time they spent thinking and commenting on my work. It was a pleasure to discuss my research in great detail with two highly-esteemed experts. My Viva was atypical: given my geographic relocation it was conducted virtually, and I required additional personnel to be present. I would like to thank John and Robert for skilfully navigating these hurdles, and I would like to thank the supporting cast who enabled the Viva to go ahead: Stephen Gilmore in Edinburgh and Rob van Glabbeek in Sydney.

Overall, my time in Edinburgh, and within the LFCS, has been culturally, intellectually and socially stimulating. It is a great city with great people in it. Thank you to those whom I met along the way: for the laughs, good times and thoughtful conversations.

My final thanks goes to my family for their constant unwavering support for whatever it is I choose to do. The journey is a long one and the end of the beginning is finally in sight.

Lay Summary

Computer programs are ubiquitous; from the 'apps' on our mobile phones, laptops, or desktop computers, through to specialised software installed in cars, aeroplanes, and satellites. Every program is designed and written with the intention of performing a certain set of tasks. A task performed by a program may be so-called *observable* by an end user, *e.g.* a web page displayed by a browser, or text typed into a word processor. How can we be sure the program actually does the tasks for which it was designed? The answer to such a question may be critically important for the end users of certain software systems, as exemplified by recent advances in self-driving cars, for example. This dissertation studies reasoning techniques for computer programs with the emphasis on observable behaviour.

We develop a mathematical *model*, a precise and formal approximation, of a computer programming language and the observable behaviour of its programs. Models are only as good as their applicability; better models incorporate more of the features present in real-world computer programs. Given the example software systems above, the language is going to need to interact with its environment, recording or manipulating data received from sensors, for example. Such interactions are collectively known as *computational effects*, or *effects* for short, and are essential to the development of nontrivial software systems. The correctness of software systems hinges on ensuring effects are performed only where the programmer intended them. Therefore, our model precisely specifies where such effects are permitted or prohibited. We achieve such precision in specification by using so-called *type-and-effect systems* and a mathematical theory of effects. A type-and-effect system classifies the elements of a computer program as either effect-producing (a *computation*) or not (a *value*).

We develop reasoning techniques which take advantage of this distinction, and of the effect specifications in our model. We study a fundamental reasoning technique: program *equivalence*. That is, the situation where two programs can be said to exhibit the same observable behaviour. Program equivalence allows correctness properties to be proven by comparing actual program execution on particular input data with expected results. Additionally, we are able to apply *optimisations*, replacement of sections of program code aimed at improving the program's performance during execution. Furthermore, we characterise program equivalence using simpler structural notions, definable by virtue of our type-and-effect discipline. We demonstrate the utility of our reasoning techniques on a number of examples expressed in our model.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Craig McLaughlin*)

It isn't where you came from, it's where you're going that counts.

—The First Lady of Song

Table of Contents

Chapter 1

Introduction

Type theory is concerned with developing mathematical models capable of describing and establishing properties of programming languages. These models are typically much simpler than the fully fledged programming languages used to build large software systems. These simplified models are collectively known as *typed* λ*-calculi*, after Church's [1940] theory of types, which aim to capture the computational essence of full languages.

However, many of the programming languages widely used today bear little resemblance to λ-calculus, incorporating object-oriented and imperative paradigms. Fortunately, there is a class of languages which have a close connection to λ-calculus: *pure functional* programming languages. These languages are *functional* in that they take the concept of a mathematical function to be the primitive construct. These languages are *pure* in the sense that such functions do not alter the environment (e.g. machine state, or user terminal) during their execution, and always return the same result for the same input. Therefore, reasoning principles developed for core typed λ-calculi transfer to pure functional programs. In particular, pure functional programs admit *equational reasoning*: the ability to substitute 'equals for equals' because purity guarantees a consistent result.

Yet, restricting ourselves to pure programs comes at a cost. We are not permitted to manipulate memory locations (persistent state between function calls), interact with the user (via standard input/output streams), or communicate across a network connection. Such phenomena are known as *computational effects* and are essential to the development of nontrivial software systems. The correctness of these software systems hinges on ensuring effects are performed only where the programmer intended them. Most programming languages do not provide support for specifying where such effects

are permitted or prohibited. Furthermore, simply introducing effects into a pure language as primitive constructs would interfere with its equational properties since the same piece of code may exhibit different behaviour on successive executions.

One way to recover reasoning principles is to extend λ-calculus models to incorporate computational effects as a *pure* abstraction. Then, pure functional languages based on these calculi are more expressive and we gain — via the models — the ability to reason about effectful programs. A key technique is to extend the type systems of these languages to statically track the effects used in a program. Such type systems are known as *type-and-effect systems* [Lucassen and Gifford, 1988].

This dissertation is concerned with establishing reasoning principles to prove properties of effectful functional programs for a language equipped with a particular type-and-effect system. Furthermore, we desire strong correctness guarantees for our meta-theoretic results. Before stating our main thesis, we briefly introduce the central topics of the dissertation.

Algebraic Effects and Handlers

There is a growing interest in representing computational effects using the theory of *algebraic effects* [Plotkin and Power, 2001, 2002]. This approach defines an effect as the collection of primitive *operations* which give rise to it accompanied by a set of *equations* governing the behaviour of the operations.

Why choose algebraic effects? One reason is the theory has a mathematically pleasing semantics encapsulating both *denotational* and *operational* semantics. Briefly, denotational semantics gives meaning to a language model by assigning to each language feature a mathematical object; the object is the *denotation* of the feature [Winskel, 1994, Chapter 5]. Operational semantics on the other hand gives meaning to a language model by assigning to each language feature a *computational* meaning; that is, how we expect programs to behave when executed by a computer. This dissertation focuses exclusively on the latter but denotational semantics features prominently, both in the original papers on algebraic effects, and in subsequent work by, for example, Pretnar [2010], Kammar [2014] and Ahman [2017].

From a practical perspective, their popularity stems from the introduction of algebraic effect *handlers* by Plotkin and Pretnar [2013]. An effect handler for an algebraic effect generalises the notion of an exception handler, providing an *interpretation* (a computational meaning) for each source of the effect. This interpretation may or may

not take account of the equations associated to a given algebraic effect. If it does not, then the handler is said to be modelling the *free* theory of the effect. We do not impose any equational theory on handler definitions in this dissertation. That said, our goal is to develop sound reasoning principles which facilitate equational reasoning for effectful programs. Being capable of validating the standard equations of an algebraic effect for a given effect handler is a sensible litmus test for our techniques.

A number of functional languages have been developed which provide first-class support for programming with algebraic effects and their handlers, including EFF [Bauer and Pretnar, 2015], KOKA [Leijen, 2017], and LINKS [Hillerström and Lindley, 2016]. All of these languages are equipped with type-and-effect systems which track the effects occurring throughout programs. As regards reasoning principles for these languages, Bauer and Pretnar's [2014] work on EFF is distinct. The reasoning principles are developed with respect to a denotational semantics. None of the other languages have so far been equipped with sound equational reasoning principles.

The FRANK programming language [Lindley et al., 2017] differs in several ways to the above languages. The most noteworthy novelty being support for *multihandlers*, an n-ary generalisation of the effect handlers presented by Plotkin and Pretnar. So far there has not been any work on establishing reasoning techniques for FRANK programs.

Observational Equivalence

Contextual or *observational equivalence* is one of the most fundamental reasoning techniques since it supports powerful equational reasoning about code, such as rewriting via chains of equivalence proofs and the substitution of equivalent sub-terms ('equals for equals') in larger programs. Roughly speaking, observational equivalence relates programs, or terms, if they *behave equivalently* in any program context [Morris, 1968]. Fundamentally, a program context is some extension of the grammar of terms to include a placeholder for a term, called a *hole*, although the exact definition of 'program context' and 'behave equivalently' varies depending on the features provided by the language under study.

Sadly, quantification over all possible enclosing contexts can also make observational equivalence difficult to prove directly. However, if it can be captured by some restricted form of contexts then the proof burden is reduced without sacrificing reasoning power. Such correspondence results are known as *context lemmas*, following

Milner's [1977] pioneering article, where he proved a context lemma for a typed combinatory logic with first-order function symbols, establishing that *applicative* contexts, which apply the hole to a sequence of arguments, suffice to distinguish terms.

Since then there have been many similar results published for a variety of calculi, spanning a range of programming-language models: from Milner's original through the *CIU* (Closed Instantiations of Uses) theorem [Mason and Talcott, 1991] to 'CIU-like' results for multiple language features. Each showing that to prove observational equivalence between terms it suffices to test only some restricted class of contexts. The particular approach used to obtain the result varies, with many based on Howe's coinductive 'precongruence candidate' method [Howe, 1989, 1996]. In contrast, earlier work for the ν-calculus [Stark, 1994] and ReFS [Pitts and Stark, 1998] used inductive techniques to establish a context lemma. A key part of their argument involves showing that a certain triangle of relations coincide. In this dissertation, we refer to the argument as the *triangulation* proof method. Thus far, triangulation has not been isolated for independent study as a general proof method for proving context lemmas. Additionally, prior work leveraging the technique was not *formally verified* by a *proof assistant*.

Formal Verification and Proof Assistants

Across mathematics, the vast majority of proofs are expressed with 'pen and paper' and verification of their correctness involves manually checking each stage of the proof step-by-step. This strategy is also employed in type theory for developing and verifying the metatheory of a language or calculus. Inevitably, this process is error-prone and occasionally an incorrect proof passes these less-than-perfect checks.

To achieve greater confidence that our proofs are correct we could write a computer program, a *proof checker*, to perform the verification phase for us. The task of the programmer is then to provide an encoding or representation of the proof — a *proof object* — which the program is capable of understanding. Now, we assume that anything which passes the checker has been 'formally verified'. For this assumption to hold, we must trust the proof checker *itself*; that is, we assume the implementation of the proof checker contains no defects which would distort the results of its checking procedure. For this reason, proof checkers are typically small pieces of software on the order of a few thousand lines of code in order to reduce the *trusted code base*. A *proof assistant*, in addition to checking the correctness of an encoded proof, aids

the programmer in constructing the proof in the first place. In this case, the checker is a small kernel of the whole system, isolated from any components not pertinent to checking proof objects.

A large class of such systems build upon the Curry-Howard correspondence [Sørensen and Urzyczyn, 2006] which describes a deep connection between typed λ-calculus and logical systems [1] . Through this correspondence, types are identified with propositions, and programs inhabiting a type correspond to proofs of the associated proposition. Thus, the process of checking a proof object becomes the process of type checking a program term.

This view of logic is *constructive*: to give a proof of a proposition exactly means to give an algorithm for *constructing* an explicit proof object, *i.e.* a term. Only propositions which can be witnessed by a construction are accepted as provable in the system. Martin-Löf's [1984] *intuitionistic type theory* [2], based on the idea of 'propositions-as-types', permits types to contain, or 'depend on', terms. A system with so-called *dependent* types can specify logical properties of individuals (term variables) by quantifying over them and mentioning them in the proposition (type). A number of dependent type theories have since been developed, further extending the original ideas by Martin-Löf, including the calculus of constructions [Coquand and Huet, 1988] and UTT [Luo, 1994] which form the backbone of the proof assistants COQ [Paulin-Mohring, 1993] and AGDA [Norell, 2007], respectively.

Whereas COQ provides *tactics* to facilitate theorem proving, in AGDA proofs are constructed by functional programming with dependent types leveraging extensions of standard techniques, *e.g. inductive families* [Dybjer, 1994], dependent pattern matching [Coquand, 1992], and `with` clauses [McBride and McKinna, 2004].

Inductive families are indexed collections of inductive datatypes generalising the algebraic datatypes familiar to users of HASKELL, for example. Since types may contain terms, the indexing may be a type or a term. A canonical example of indexing by a term is the *vector* datatype, the type of lists indexed by their length (a term of type $\mathbb{N}$).

Pattern matching on a term of an inductive family may yield information regarding the valid instances of its type's indices. McBride and McKinna introduce the `with`

[1] The LCF tradition [Gordon et al., 1979] is another methodology, and forms the basis of the ISABELLE/HOL proof assistant [Lawrence et al., 2019].

[2] The term "intuitionistic" is used interchangeably with the term "constructive". The former is derived from the logical intuitionism of L. E. J. Brouwer. For a historical and philosophical perspective on intuitionistic mathematics see the article by Iemhoff [2019]. Sørensen and Urzyczyn [2006] provide a more formal account making the connection with type theory explicit.

construct which allows discrimination of intermediate computations to occur on the left of a defining equation. By performing the discrimination on the left-hand side instead of using a conventional case analysis on the right-hand side, the intermediate computation is generalised over the arguments and the result type. AGDA's interactive mode directly supports refinement of goals by dependent case analysis and the `with` construct.

There has been increasing interest in providing machine-checked proofs for the meta-theoretic results of λ-calculus models. Besides gaining greater confidence in our results, a mechanised development can also provide assistance in extending an existing model to include new features by highlighting where the changes have affected proofs. Using inductive families, it is possible to give such models *intrinsic* representations which root out ill-formed terms. Examples of terms which could be eliminated 'by construction' are ill-typed or ill-scoped terms. This approach is achieved by indexing the meta-level description of terms, *e.g.* an inductive type, by their type and context.

The intrinsic approach is one particular strategy for object-language representation particularly well-suited for developments using AGDA due to its aforementioned support for dependently typed programming. To our knowledge, AGDA has so far not been applied to formalising results about observational equivalence.

1.1 Thesis Statement

In the previous section, we identified the following three gaps in the literature.

1. Reasoning techniques for FRANK programs;

2. Formalised treatment of *triangulation* as a general proof method for proving context lemmas;

3. Applicability of AGDA for formalising observational equivalence results.

This dissertation contributes to addressing these gaps by considering their confluence in the context of the following central thesis.

There are sound (in)equational reasoning principles applicable to monomorphic FRANK *programs which are amenable to formalisation in an implementation of type theory such as* AGDA.

The aim of this dissertation is to contribute to the development of a complete characterisation of observational equivalence for FRANK programs using the triangulation proof method. We develop a simplified λ-calculus model, ELLA, based on a monomorphic version of FRANK which we argue captures the essence of the full language. We conjecture there exists a translation of FRANK programs into monomorphic ELLA programs. This conjecture is justified by an existing translation of FRANK into a core calculus with only unary handlers and case analysis [Lindley et al., 2017]. One way of viewing ELLA is as the target of a type-annotated compilation phase.

We describe the progress we have made in formally verifying our results using the AGDA programming language and proof development system. In particular, we describe how we leverage an existing state-of-the-art framework for representing syntaxes with binding [Allais et al., 2017]. We describe how the formalisation has assisted in scaling our results to richer calculi, informing us where our extensions have affected the metatheory.

We divide our central thesis into the following subsidiary parts.

1. Triangulation extends to characterise observational equivalence for ELLA and the constituent relations admit a collection of sound reasoning principles;

2. The reasoning principles for ELLA are capable of proving concrete ELLA program approximations.

1.2 Contributions

The contributions of this dissertation are:

- An account of the FRANK programming language, motivating its design and providing examples;

- A formal reconstruction in AGDA of the general triangulation proof method for proving context lemmas;

- A strict functional language, ELLA, — modelled after FRANK— which represents effects using abstract operations and handlers, inspired by the algebraic theory of effects and handlers [Plotkin and Power, 2001; Plotkin and Pretnar, 2013];

- A complete syntactic characterisation of observational approximation and a collection of sound reasoning principles for ELLA;

- Proofs of example ELLA approximations demonstrating the applicability of the reasoning principles;

- We describe the progress we have made in verifying the results of this dissertation using AGDA.

1.3 Dissertation Structure

Chapter 2 introduces the relevant background necessary to understand the technical material presented in this dissertation. We assume as a prerequisite that the reader has basic knowledge of functional programming and familiarity with either HASKELL or ML.

Chapter 3 presents a tutorial on the Frank programming language, and an extended example on using effects and handlers for fine-grained control of effects. We provide points of comparison to other functional programming languages, and in particular HASKELL and its use of *monads* as a pure abstraction for computational effects.

The triangulation proof method is presented in Chapter 4 for the fine-grained call-by-value [Levy, 2004] calculus $\lambda^{\rightarrow}_{FG}$. The intractability of proving *contextual* or *observational* equivalences directly is avoided by leveraging structural notions obtained from the method. We choose to study $\lambda^{\rightarrow}_{FG}$ because of its simplicity, allowing us to focus our exposition on the triangulation proof technique. The evaluation technique of *frame stacks* [Pitts and Stark, 1998; Pitts, 2005] is also introduced which is later extended to the ELLA setting.

Chapter 5 presents the $\mathtt{FOL}^{\mu\triangleright}$ logic for extending our triangulation result to a setting with general recursion. The material in this chapter is heavily inspired by existing logics [Dreyer et al., 2011] and structures [Di Gianantonio and Miculan, 2003] for abstracting *step indices* used to facilitate recursive language features [Ahmed, 2006]. We discuss the relevant background for this chapter *in situ*, after having introduced preliminary concepts in Chapter 4.

Chapter 6 presents the ELLA calculus, a monomorphic λ-calculus model of FRANK. We present the semantics of ELLA in terms of *handler stacks*, a generalisation of the frame stacks from Chapter 4. We extend triangulation to ELLA, proving a context

lemma and deriving a collection of reasoning principles. This chapter substantiates our first subsidiary thesis.

Chapter 7 presents some example ELLA program approximations, proving them using the reasoning techniques. This chapter substantiates our second subsidiary thesis.

Chapter 8 concludes the dissertation and suggests directions for future work.

Chapter 2

Background

This chapter introduces essential background material to understand the technical developments in this dissertation. We focus our attention on the key topics which appear throughout the dissertation, and merit a firm grounding.

We provide an introduction to the theory of algebraic effects and effect handlers upon which our approach for representing computational effects is based. We highlight the advantages of this representation decision compared to alternative choices.

We present the high-level methodology behind the triangulation proof method for proving context lemma results, and the benefits it confers for reasoning about concrete examples.

Lastly, we discuss formalising metatheory within an interactive theorem proving environment, and the variety of representation choices available. We conclude the discussion with a brief description of a meta-language for approximating our concrete models in AGDA.

Auxiliary concepts are spread throughout the remaining chapters and introduced as the need arises.

2.1 Algebraic Effects

Most programming languages permit the programmer to perform operations on the state of the underlying machine and interact with the external world, e.g. printing output to the screen or receiving input from the user. These behaviours, among others, are collectively known as *computational effects*. Being able to reason about programs involving computational effects is crucial if we are to have any confidence that programs behave as the programmer intended.

The seminal work by Moggi [1989a] provided a uniform denotational semantic account of computational effects in λ-calculus. The key insight was to represent computational effects, or *notions of computations* as Moggi called them, using monads. In this view, pure *values* of type A are distinguished from *computations*, which may perform effects, of type $T\,A$, where T is the underlying functor of the monad representing the effect(s). Many effects can be represented in this way including nontermination, input and output, nondeterminism, state, exceptions and continuations. The monad can be informally viewed as a transformer on ordinary pure values of a given type. Moggi's semantic account was given categorically by the *Kleisli* category $\mathscr{C}_T$ with respect to an ambient category $\mathscr{C}$ representing the domains for denotational semantics. We give some example computational effects working in the ambient category **Set** of sets and functions.

Example 2.1 (Notions of Computation)**.** [Moggi, 1989a]

- Exceptions: $T(A) = (A + E)$ where E is the set of exceptions;
- Nondeterminism: $T(A) = \mathscr{P}(A)$, the powerset of A;
- State: $T(A) = S \to (A \times S)$ where S is the set of states;
- Continuations: $T(A) = (A \to R) \to R$ where R is the set of results.

This monadic discipline became particularly influential in the functional programming community where monads have been adopted by HASKELL to mimic impurity within a pure setting [Benton et al., 2002; Wadler, 1995].

Example 2.2 (HASKELL State Monad)**.** Consider defining the monad for state in HASKELL. Recall from our previous example, we represent stateful computations as functions from a state S to a pair $A \times S$ consisting of the type A of values produced by running the computation, and S the final state. In HASKELL, we can define a type to express such a notion as follows.

```
newtype State s a = State { runState :: s -> (a,s) }
```

where `State` is defined as a type with one (eponymous) constructor which has a single field, a function `runState` with a type signature matching the above description. Using HASKELL typeclass instances [Marlow, 2010], we define `State` to be a monad as follows.

```
instance Monad (State s) where
  return x = State $ \s -> (x,s)
  m >>= k = State $ \s -> case runState m s of
    (x,s') -> runState (k x) s'
```

To be a valid instance of the `Monad` typeclass, we must provide definitions for the two operations `return`, which injects pure values into the monad, and *bind*, written `>>=`, which composes monadic computations. In this case, we can see that `return` simply pairs the value argument with the state provided. For bind, we run the first computation, producing a result value and a new state which are provided to the continuation `k`. Of course, for `State s` to be a monad in the categorical sense, the `return` and `>>=` definitions above must adhere to the standard monad laws [Mac Lane, 1971]. These laws cannot be captured using HASKELL's type system but are within the purview of dependently typed systems [Agda, 2019b; Gross et al., 2014].

In this section we borrow from HASKELL parlance, using **return** v to denote the injection of a pure value v into a computation.

Dissatisfied by the lack of an operational semantics for Moggi's monadic approach, Plotkin and Power [2001] embarked on describing a class of computational effects known as *algebraic effects*. Informally, the idea is to take as primitive the sources of a particular effect. The sources of an effect are the primitive *operations* which give rise to it. The arguments supplied to an operation are the possible continuations of the computation.

Roughly speaking, from a programming perspective, an operation is considered *algebraic* if it commutes with evaluation contexts [Plotkin and Power, 2003], e.g. for an operation **op** we must have,

$$\mathcal{E}[\mathbf{op}(x_1,\ldots,x_n)] \equiv \mathbf{op}(\mathcal{E}[x_1],\ldots,\mathcal{E}[x_n])$$

for all evaluation contexts $\mathcal{E}$, where $\equiv$ denotes program equivalence (see Section 2.3).

Algebraic effects include input and output, nondeterminism, state, exceptions, but not continuations, as Plotkin and Power [2002] have shown they do not satisfy the required conditions.

Plotkin and Power recognised that certain computational effects could be described by *algebraic theories*. An algebraic theory consists of two components: a signature and a collection of equations over the signature. An *algebraic signature* consists of a collection of operation symbols each with an associated arity corresponding to the

number of possible continuations for the operation. Equations of an algebraic theory are between two computations built up from a context of variables and the operation symbols from the signature.

In this dissertation we focus on *free* algebraic theories, that is we consider operations by themselves without any associated equations. This is the common choice for languages supporting algebraic effects [Bauer and Pretnar, 2014; Hillerström and Lindley, 2016; Leijen, 2017]. However, as mentioned in the introduction, we should expect our reasoning principles to be able to validate the usual equations for particular effect handlers. So, in this section, we present algebraic theories with the accompanying equations. We use '=' for equations which hold by virtue of the given algebraic theory, reserving '$\equiv$' for equivalence of terms which hold operationally.

Example 2.3 (Exceptions)**.** The algebraic theory for exceptions consists of a signature containing an operation **throw**. Since throwing an exception discards the rest of a computation, the **throw** operation takes no arguments. There are no equations for this theory.

Example 2.4 (Nondeterminism)**.** The algebraic theory for nondeterminism consists of a signature containing one binary operation, **or**, which nondeterministically chooses between two alternative computations, M and N: $\mathbf{or}(M,N)$. The equations governing nondeterminism are those of the theory of semilattices:

$$\mathbf{or}(M,M) = M$$

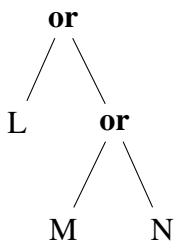
$$\mathbf{or}(M,N) = \mathbf{or}(N,M)$$

$$\mathbf{or}(L,\mathbf{or}(M,N)) = \mathbf{or}(\mathbf{or}(L,M),N)$$

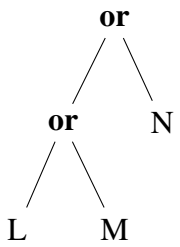
which at least agree with our computational intuition regarding nondeterminism.

We can view programs using algebraic effects as computation trees recording at each node where an operation is invoked. The children of an operation represent the possible results produced by the effect. At a minimum, this viewpoint only requires knowledge of the algebraic signature of an effect. We can quotient the set of such trees by identifying trees which satisfy the same equations.

Example 2.5. We may express $\mathbf{or}(L,\mathbf{or}(M,N))$ in the following tree form:



and modulo the equations imposed by algebraic theory for nondeterminsim, the above tree should be identified with:



In this way, the set of computation trees generated by the algebraic signature are *quotiented* by the equations of the theory.

In order to evaluate such trees, we need to determine an appropriate semantics for interpreting the operations occurring in the tree. How do we get from an algebraic theory of an effect to a semantics for effectful programs? Plotkin and Power [2002] obtain a semantics for a computational effect through its algebraic theory which induces the corresponding computational monad originally studied by Moggi.

We may generalise the notion of algebraic operation given so far in two ways to yield a representation more amenable for inclusion in a programming language [Plotkin and Power, 2003].

First, we may have *parameterised* effects. For instance, in Example 2.3 we gave the algebraic signature for a single exception. Instead, we could have given the signature with respect to a set of exceptions E. We now have a choice as to how to construct the operations of the signature. One approach is to have the signature consist of an operation $\mathbf{throw}_e$ for each $e \in E$. Alternatively, we could add the parameter as an argument to the operation itself. So that for each $e \in E$, **throw** e raises the e exception. We adopt the latter approach in subsequent examples since it accords well with functional programming practice.

Second, the operations we have seen thus far accept a sequence of arguments for every possible continuation. This style would be quite unwieldy for programming.

Instead, we may express operation invocations using *continuation-passing style* (CPS). That is, an operation invocation takes the form

$$\mathbf{op}\ v\ (\lambda x.k)$$

where v is the argument value for any parameter the operation may have, and k represents the rest of the computation. The continuation, $(\lambda x.k)$, accepts the value returned by the operation, if any. To be a bit more formal about the typing of operations, we specify operations with their parameter type (P) and result type (R): $\mathbf{op}[P] : R$. We recast previous examples using this new style.

Example 2.6 (Exceptions)**.** Exceptions discard the rest of the computation. To model this behaviour, we use the empty type, **zero**, as the result type of the operation. This has the consequence that the continuation can never be invoked because it expects an argument of an uninhabited type. Therefore, the operation for the theory is $\mathbf{throw}[E] : \mathbf{zero}$.

Example 2.7 (Nondeterminism)**.** The binary operation for nondeterminism chooses between *two* alternatives. The theory has no parameters. We thus get an operation $\mathbf{choose}[\mathbf{unit}] : \mathbf{bool}$ with the induced equivalence:

$$\mathbf{choose}\ ()\ (\lambda b.\mathbf{if}\ b\ \mathbf{then}\ M\ \mathbf{else}\ N) \equiv \mathbf{or}(M,N).$$

Example 2.8 (State)**.** The algebraic theory of state is a good example of a parameterised theory. The parameter set L specifies the collection of locations, which we assume is finite. The arity of the operations is drawn from S, the range of possible values storable at the locations. The theory has two operations:

$$\mathbf{lkp}[L] : S \qquad\qquad \mathbf{upd}[L \times S] : \mathbf{unit}$$

whose interactions are governed by the following equations:

$$\begin{aligned}
\mathbf{lkp}\ \ell\ (\lambda s.\mathbf{lkp}\ \ell\ (\lambda s'.M\ s\ s')) &= \mathbf{lkp}\ \ell\ (\lambda s.M\ s\ s) && (1)\\
\mathbf{lkp}\ \ell\ (\lambda s.\mathbf{upd}\ (\ell,s)\ (\lambda().M)) &= M && (2)\\
\mathbf{upd}\ (\ell,s)\ (\lambda().\mathbf{upd}\ (\ell,s')\ (\lambda().M)) &= \mathbf{upd}\ (\ell,s')\ (\lambda().M) && (3)\\
\mathbf{upd}\ (\ell,s)\ (\lambda().\mathbf{lkp}\ \ell\ (\lambda s'.M\ s')) &= \mathbf{upd}\ (\ell,s)\ (\lambda().M\ s) && (4)
\end{aligned}$$

for all $\ell \in L$ and $s, s' \in S$, where M represents the rest of the computation tree. There are three additional equations if we consider interactions involving different memory locations, which we omit. For details, see e.g. Plotkin and Power [2002] or Pretnar [2010].

A final switch to more programmer friendly syntax is to consider the *generic effect* arising from an algebraic operation [Plotkin and Power, 2003]. Given an operation **op** invoked using CPS, the generic effect for **op**, written $\mathbf{op}_g$, has the trivial continuation:

$$\mathbf{op}_g\ v \triangleq \mathbf{op}\ v\ (\lambda x.\mathbf{return}\ x)$$

In this dissertation, we usually specify (free) effect theories using the generic effect presentation. The CPS variant can be obtained by using a standard sequencing construct like **let**.

Example 2.9. The generic effect for nondeterminism is the operation, **choice** : **bool**. The original algebraic operation can be simulated by the generic effect in combination with conditional branching:

$$\mathbf{if\ choice\ then}\ M\ \mathbf{else}\ N \equiv \mathbf{or}(M,N).$$

It is rare when a nontrivial program only utilises one effect at a time. A natural next step is to look into how to combine effects. Subsequent to the development of monads for modelling individual effects, monad transformers were originally proposed by Moggi [1989b] in order to combine multiple effects together within the same program. These ideas were later extensively developed, beginning with Liang et al. [1995], as a modular programming technique within the functional programming community.

In HASKELL, a monad transformer is realised as a stack of transformations built on top of each other, heavily utilising the typeclass and class constraint facilities to provide abstraction and modularity. A transformer accepts as a parameter a base monad producing a monad whose operations include the base monad operations in addition to those associated with the transformer.

Example 2.10. Consider the combination of state with some other computational effects expressed in HASKELL. The monad transformer for state is quite similar to the `State` monad we defined in Example 2.2:

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a, s)) }
```

where `m` is a monad. The definition augments the monad `m` with the ability to thread state through the computation. We said a monad transformer produces a monad and indeed that is the case here; when `m` is a monad, `StateT s m` is also a monad:

```
instance Monad m => Monad (StateT s m) where
  return v = StateT (\s -> return (v, s))
```

```
m >>= k = StateT (\s -> do (x, s') <- runStateT m s
                           runStateT (k x) s')
```

where we have used HASKELL's do-notation to evaluate the monadic computation produced by `runStateT`.

Unfortunately, while monad transformers can abstractly describe effectful computations in a modular fashion, concrete instances suffer from the need to lift component monads through the stack [Schrijvers et al., 2019]. To perform operations of the base monad the programmer inserts `lift` operations to convert them to operations on the transformer:

```
class (Monad m, Monad (t m)) => MonadTrans t m where
  lift :: m a -> t m a
```

Algebraic theories admit natural combinations which account for monad transformers for combining algebraic effects [Hyland et al., 2006]. Combinations may be a straightforward *sum* of the operations and equations of the individual theories. Alternatively, there may be additional equations governing the interaction between the algebraic operations of the different theories, e.g. to enforce commutativity (the *tensor*). In either case, the associated mathematical theory prescribes uniform liftings for algebraic operations, improving on previous ad-hoc definitions of lifting in the literature and inspiring subsequent library implementations [Jaskelioff, 2009, 2011].

So far we have traced the development of a mathematical theory for algebraic effects and their operations. Again, the operations represent the sources that give rise to the effect. One aspect we elided in our presentation of exceptions was the notion of exception *handling*. It turns out that if we take the view of algebraic operations being effect *constructors*, then effect handlers are the dual notion: effect *deconstructors* [Plotkin and Pretnar, 2013].

2.2 Effect Handlers

There is a general theory for handling algebraic effects due to Plotkin and Pretnar [2013]. Mathematically it amounts to providing a *model* of the algebraic theory. That is, an effect handler provides an interpretation of the operations, which are the sources for the effect, such that the interpretation satisfies the equations of the theory. The

idea behind effect handlers takes inspiration from Benton and Kennedy [2001], who introduced a generalised exception handling construct:

$$\textbf{try } x \Leftarrow M \textbf{ in } N_r \textbf{ unless } (E_i \Rightarrow N_i)_i$$

where the entire computation evaluates to N_r except in case M throws any of the exceptions E_i in which case it evaluates to N_i. The key addition over ordinary exception handlers is the continuation N_r which binds the result of evaluating M to x. If M throws an exception, it does not return a value so therefore x is not bound in the N_i's.

Plotkin and Pretnar [2013] further generalise the above construct to the handling of arbitrary algebraic effects. If we consider the theory of exceptions, it consists of a single operation $\textbf{throw}[E] : \textbf{zero}$ which, if invoked, discards the rest of the computation i.e. its continuation. In general, for an arbitrary algebraic effect this is not typical and the continuation may contain further operation invocations, e.g. Example 2.5. Thus, we arrive at the following notation for specifying a handler H:

$$H = \{\textbf{return } x \mapsto N_r; (\textbf{op}_i\ x_i\ k_i \mapsto N_i)_i\}$$

where we have given the operation clauses in continuation-passing style: the x_i argument represents the parameter of the $\textbf{op}_i$ operation, whereas k_i represents the continuation. We assume the operations occurring in H are all distinct. For the time being, we also assume H handles the operations of a single effect only.

Once we have a handler H, we may use it to handle effect operations invoked by a computation M using the following handling construct:

$$\textbf{handle } M \textbf{ with } H$$

Example 2.11 (Exceptions)**.** The theory of exceptions is a special case of the general handling mechanism above. We take the theory of Example 2.3 and assume given computations N_r and N_e of the same type. Then, given a computation M, we may handle any exceptions thrown in M by:

$$\textbf{handle } M \textbf{ with } \{\textbf{return } x \mapsto N_r; \textbf{throw } ()\ k \mapsto N_e\}$$

where k cannot be invoked by N_e because k expects an argument of the uninhabited type, **zero**. The above handler trivially models (interprets) the theory of exceptions since there are no equations to satisfy.

From the general definition of a handler H, we see that in the case M returns a value, we get the same behaviour as the generalised exception handler of Benton and

Kennedy. In the situation where M invokes some operation $\mathbf{op}_i \in H$, we find its corresponding replacement N_i. However, in the general case we must take account of the continuation k. The continuation may contain further invocations of the operations handled by H. The mathematically natural approach is to handle those operations with H as well; these *deep* handlers are models of the algebraic theory, and the **handle-with** construct corresponds to the homomorphism arising from the universal property of the free model [Pretnar, 2010]. That is, for an operation **op** such that $(\mathbf{op}\ x\ k \mapsto N) \in H$, the mathematical model validates the equation:

$$\mathbf{handle}\ \mathcal{E}[\mathbf{op}\ v]\ \mathbf{with}\ H \equiv N[v/x, (\lambda y.\mathbf{handle}\ E[y]\ \mathbf{with}\ H)/k]$$

for all evaluation contexts $\mathcal{E}$. Furthermore, H must satisfy the equations associated with the theory which involve **op**.

In the above equation, we say that the handler is 'reinvoked' within the continuation k, or 're-wraps' over the rest of the computation. There is another class of handlers where k is replaced by $\lambda y.\mathcal{E}[y]$ without any re-wrapping of the handler. These handlers are known as *shallow* handlers. The distinction between deep and shallow handlers is due to Kammar et al. [2013]. Viewing a computation as a tree, deep handlers are understood as folds [Meijer et al., 1991] over the tree whereas shallow handlers represent a single case split.

In this dissertation, we focus on shallow handlers since our aim is to characterise FRANK's multihandlers which are shallow, but our examples always wrap the continuation in the original handler, simulating the behaviour of deep handlers. The (shallow) multihandlers studied in this dissertation are capable of handling multiple computations at once and represent a distinct point in the design space. Hillerström and Lindley [2018] show the simulation of unary shallow handlers by unary deep handlers, among other results for shallow effect handlers, but do not study multihandlers.

2.3 The Triangulation Proof Method

The problem of determining equivalence between programs or program fragments (terms) is well-studied and a variety of approaches have been developed [Gordon, 1994; Howe, 1989, 1996; Lassen, 1998a,b; Mason and Talcott, 1991; Pitts, 1997, 2005, 2011; Stark, 1994; Pitts and Stark, 1998]. Since Morris [1968], program equivalence is usually defined in terms of a notion of contextual or observational equivalence. For calculi with deterministic evaluation semantics, observational equivalence arises as the

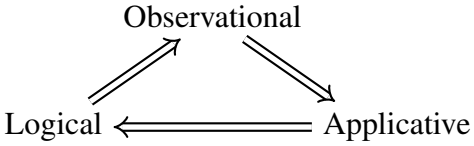
symmetrisation of the associated *approximation* preorder. A term is observationally approximate to another if the first term *behaviourally approximates* the second term in any program context.

There is some variation regarding the notions of 'program context' and 'behave approximately'. For example, program contexts often involve some kind of free-variable capture, and may be a restricted subset of the term grammar. Whereas, approximation may require both terms return equal values at ground type, or for languages with nontermination, simply requiring the two terms equiterminate.

The quantification over program contexts from the language itself turns out to make observational approximation robust — independent of the class of available observations — and self-regulating: a language with more elaborate features has a correspondingly larger collection of testing contexts, *e.g.* Dreyer et al. [2012] exhibit how various language features affect the discriminatory power of contexts.

However, as mentioned in Chapter 1, such quantification over all possible enclosing contexts can also make observational equivalence difficult to prove directly. The motivation behind establishing a context lemma, such as Milner's [1977] original result, is to reduce the class of contexts that need to be considered.

Following earlier proofs for the ν-calculus [Stark, 1994] and ReFS [Pitts and Stark, 1998], we isolate a key part of their argument for independent study: proving a context lemma by showing that a certain triangle of relations coincide. We propose this *triangulation* proof method as a general technique for proving context lemmas.



The triangle links three kinds of relation between terms:

Observational where terms are related by their behaviour in some class of contexts;

Applicative where function terms are related if they take *identical* arguments to related results;

Logical where function terms are related if they take *related* arguments to related results.

While observational relations are defined by quantification over contexts, both applicative and logical relations are given by type structure. This structural character of

these lower relations shapes the proofs along all three sides — in particular the implication from logical to observational arises from the fundamental property of logical relations.

The base of the triangle also shares out the distinctive properties of observational relations for powerful reasoning: applicative approximation is easily shown to be transitive, and logical approximation a congruence.

Observational relations require some chosen class of contexts, and in fact we see the triangle apex ramify into a chain of relations from behaviour in all contexts down through more restricted collections, with each relation naturally implying the next as the set of testing contexts becomes smaller. However, the triangulation result brackets all these observational relations between logical on the left and applicative on the right, proving that they collapse into each other. This collapse — that testing in all contexts is the same as testing over some restricted class — is the content of the context lemma.

2.4 Mechanised Metatheory

We seek to formalise our results in a proof assistant to strength confidence in the correctness of our reasoning techniques, and also to open up the possibility of formally verifying specific program approximation examples.

There is a growing interest in formalising the metatheory of programming languages, in part due to the proliferation of suitable tools including AGDA, COQ, ISABELLE/HOL [Nipkow and Klein, 2014], IDRIS [Brady, 2013], and BELUGA [Pientka, 2008, 2010]. Each system provides a *meta-language* in which to write programs and prove theorems. As mentioned in Chapter 1, AGDA is based on Luo's [1994] UTT, and the Calculus of Inductive Constructions [Paulin-Mohring, 1993] is the basis of the COQ proof assistant. In contrast, ISABELLE/HOL is based on higher-order logic (not dependent type theory), and BELUGA is based on an extension of the Edinburgh Logical Framework [Harper et al., 1993].

The *object-language* is the model we intend to formalise, *e.g* the simply-typed λ-calculus. There are two approaches for embedding an object-language within a meta-language, known as the *shallow* and *deep* embeddings.

A shallow embedding defines the object-language directly in terms of its semantics using meta-language definitions. Shallow embeddings are amenable to extension by simply defining new meta-language definitions for new language constructs. Furthermore, by virtue of their direct semantics it is often easier to compute with compound

terms of a shallow embedding.

A deep embedding on the other hand is a representation of the object-language abstract syntax using a meta-language data structure, *e.g.* algebraic datatypes, or more generally, inductive families [Dybjer, 1994]. Deep embeddings are useful when multiple interpretations of the syntax are required, *e.g.* evaluation semantics, substitution, pretty printer, *etc*. It is straightforward to operate over the syntax by traversal of the defining data structure but computing with concrete terms is more involved because the embedding cannot take advantage of meta-language constructs.

In this dissertation, we model our calculi using deep embeddings since our aim is to prove meta-theoretic properties by traversal over syntax trees, rather than the provision of a single semantic interpretation.

There are different styles of deep embedding depending on whether we choose to define constructs by syntax or semantics. A key decision is how to represent variables. A *first-order* representation provides a syntactic representation of variables, de Bruijn [1972] indices being a widely used example. The de Bruijn encoding represents variables $\mathbf{var}_k$ as numeric offsets k, generated by zero and successor, into an environment.

Example 2.12 (de Bruijn Encoding)**.** The **K** combinator, $\lambda x.\lambda y.x$, is expressed in the de Bruijn encoding as $\lambda(\lambda(\mathbf{var}_{\mathbf{succ}\,0}))$.

An alternative encoding is *higher-order abstract syntax* (HOAS) where λ-abstraction in the object-language is represented by λ-abstraction in the meta-language; hence, variables in the object-language are identified with variables in the meta-language. Weak HOAS uses an auxiliary parameter to represent binders with another constructor injecting the parameter into the type of terms. This weaker form is used in proof assistants — *e.g.* Coq [Despeyroux et al., 1995] — whose inductive datatypes must be *strictly positive*, and consequently no self-reference may occur in the domain of a nested function type.

The final axis of variation we shall consider is between the *intrinsic* and *extrinsic* approaches [Altenkirch and Reus, 1999; Reynolds, 2000]. In the extrinsic approach, the term syntax of the object-language is first defined, then a separate definition carves out the well-typed and well-scoped terms. Theorems regarding well-typed and-scoped terms must carry around this additional predicate. In contrast, intrinsic specifications augment the definition of the term syntax with the type and scoping information such that only well-typed and-scoped terms can be constructed. Intrinsic encodings are defined using inductive families.

Example 2.13 (STLC as an Inductive Family)**.** We adopt notation similar to McBride and McKinna [2004] for presenting the encoding of the simply typed λ-calculus ($\lambda^{\to}$) as an inductive family. The definition given here is with respect to a meta-language which we introduce shortly. For now, assume `Set` is the type to which all user-defined types belong.

Our $\lambda^{\to}$ calculus has booleans and arrow types.

$$\textbf{data}\ \frac{}{\mathsf{Ty} : \mathtt{Set}}\ \textbf{where}\ \frac{}{\mathsf{bool} : \mathsf{Ty}}\quad \frac{A : \mathsf{Ty} \quad B : \mathsf{Ty}}{A \to B : \mathsf{Ty}}$$

Contexts (Cx) are just snoc lists of types formed by $[]$ and $:<$. The above inductive datatype captures the grammar for types.

We capture our de Bruijn variable representation in the following inductive family.

$$\textbf{data}\ \frac{A : \mathsf{Ty} \quad \Gamma : \mathsf{Cx}}{\mathsf{Var}\ A\ \Gamma : \mathtt{Set}}\ \textbf{where}\ \frac{}{\mathsf{zero} : \mathsf{Var}\ A\ (\Gamma :< A)}\quad \frac{k : \mathsf{Var}\ A\ \Gamma}{\mathsf{succ}_{\mathsf{B}}\ k : \mathsf{Var}\ A\ (\Gamma :< B)}$$

where the subscript B represents an *implicit* Ty argument; we usually omit subscripts for implicit arguments when clear from the context. The above inductive family does not capture a mere grammar for variable occurrence. Rather, the extra expressivity of inductive families captures *judgements*. We could have expressed the above inductive family in a more traditional inference rule style as follows.

$$\frac{}{0 : (A \in \Gamma, A)}\ \text{ZERO}\qquad \frac{k : (A \in \Gamma)}{\mathbf{succ}\, k : (A \in \Gamma, B)}\ \text{SUCC}$$

We are now in a position to define the syntax for $\lambda^{\to}$ terms intrinsically.

$$\textbf{data}\ \frac{A : \mathsf{Ty} \quad \Gamma : \mathsf{Cx}}{\mathsf{Trm}\ A\ \Gamma : \mathtt{Set}}\ \textbf{where}$$

$$\frac{k : \mathsf{Var}\ A\ \Gamma}{\mathsf{var}\ k : \mathsf{Trm}\ A\ \Gamma}\quad \frac{}{\mathsf{tt} : \mathsf{Trm}\ \mathsf{bool}\ \Gamma}\quad \frac{}{\mathsf{ff} : \mathsf{Trm}\ \mathsf{bool}\ \Gamma}$$

$$\frac{M : \mathsf{Trm}\ B\ (\Gamma :< A)}{\lambda\, M : \mathsf{Trm}\ (A \to B)\ \Gamma}\quad \frac{M : \mathsf{Trm}\ (A \to B)\ \Gamma \quad N : \mathsf{Trm}\ A\ \Gamma}{\mathsf{app}\ M\ N : \mathsf{Trm}\ B\ \Gamma}$$

which, just like Var, corresponds to a collection of inference rules:

$$\frac{k : (A \in \Gamma)}{\Gamma \vdash \mathbf{var}_k : A}\ \textsf{var} \qquad \frac{}{\Gamma \vdash b : \mathbf{bool}}\ \textsf{bool}\ (b = \mathbf{tt}, \mathbf{ff})$$

$$\frac{\Gamma, A \vdash M : B}{\Gamma \vdash \lambda_A M : A \to B}\ \lambda \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \textsf{app}$$

where our inference rule presentation denotes keywords, types, and defined constants using **boldface**. Given this interpretation of inductive families as collections of inference rules, in the rest of this dissertation we opt to give only the inference rule presentation for judgements, focussing on the mathematics rather than its rendering in a meta-language syntax.

Aydemir et al. [2005] proposed a set of challenges to motivate an exploration of solutions to mechanising metatheory. As observed by Lee et al. [2012], solutions to the challenge problems have to consider a number of recurrent aspects of formalisations including variable binding techniques, object-language representations, capture-avoiding substitution and its standard properties *e.g.* commutation with weakening and more general *renamings* of free variables. Benton et al. [2012] describe an intrinsic first-order development of a simply-typed λ-calculus and System F in Coq. They demonstate an abstraction of substitutions and renamings into a general framework of *maps* operating over the term syntax. Allais et al. [2017] follow suit by constructing an Agda framework — referred to as the ACMM framework — for generic traversals over syntax which abstractly capture concrete instances such as substitution and renaming, amongst many other examples. Allais et al. go further by generalising the maps of Benton et al. to a notion of *semantics* supporting arbitrary thinnings of the context under binders. More significantly, their framework provides for a generic definition of *simulation* where the result of two semantics may be related, a generalisation of the *fundamental theorem* of logical relations proven by Benton et al. [2012] for a set-theoretic denotational semantics. Yet more generality is achieved by a notion of *fusible semantics* which generically proves relatedness between the composite of two semantics, and a third. Through ACMM, standard properties of substitution and renaming, and combinations thereof can be proven once and for all. ACMM was further extended by Allais et al. [2018] to operate over *generic descriptions* of syntaxes with binding.

$$\begin{aligned}(\text{Terms})\ M,N,A,B ::= {} & x \mid \Pi x : A\,.\,B \mid \mathtt{Set}_i \\ & \mid \Sigma x : A\,.\,B \mid \mathsf{proj}_1(M) \mid \mathsf{proj}_2(M) \mid \langle M,N\rangle_A \mid 1 \mid \top\end{aligned}$$

Figure 2.1: The meta-language $ML_{\mathtt{UTT}}$.

For mechanising the results of this dissertation, we choose AGDA because of its support for intrinsic terms, and the availability of the state-of-the-art ACMM framework by Allais et al. [2017]. While we do not have machine-checked proofs for all of our results, we report on the progress we have made towards this goal, and justify our belief that our development should scale to the remaining results. The AGDA encoding of most of the mathematics in this dissertation is left implicit leveraging the *judgements as types* principle as described in Example 2.13. Where we require to be explicit, *e.g.* for justifying the adequacy of our representation(s), we seek recourse from a meta-language, $ML_{\mathtt{UTT}}$, presented in Figure 2.1 which approximates the meta-language provided by AGDA. In particular, we are explicit about our encoding of a modal logic (Chapter 5) within which we formalise the results of Chapter 6. While first-order logic is supported by AGDA (and hence $ML_{\mathtt{UTT}}$) by fiat, we require a more sophisticated base logic within which to reason about our FRANK-inspired calculus ELLA.

$ML_{\mathtt{UTT}}$ is a Martin-Löf [1984] type theory with universes and inductive families; essentially the predicative part of Luo's [1994] UTT. The meta-language has dependent function types (Π-types), a cumulative hierarchy of type universes ($\mathtt{Set}_i$, for $i = 0, 1, \ldots$), dependent sum types (Σ-types), projection and pairing constructs, and the unit type, 1, and its only constructor, $\top$. Additionally, further base types (*e.g.* sum types) can be introduced as necessary by defining them using inductive families.

We utilise the following simplified notation:

- $A \to B$ for $\Pi x : A\,.\,B$ if $x \notin B$;
- $A \times B$ for $\Sigma x : A\,.\,B$ if $x \notin B$;
- $\mathtt{Set}$ for $\mathtt{Set}_0$.

We do not give a semantics for $ML_{\mathtt{UTT}}$, appealing instead to our concrete AGDA model. Additionally, we point to Goguen's [1995] metatheoretic results for a more elaborate calculus, featuring an impredicative hierarchy of propositions, proving type-checking is decidable, and by Curry-Howard, so too is proof checking. For situations

where our proofs or definitions rely on dependent pattern matching or other facilities provided by AGDA but not explicitly defined for $ML_{\mathtt{UTT}}$, we refer to Norell's [2007] results for a similar meta-language.

Chapter 3

FRANK: A Language for Effectful Functional Programming

In this chapter we introduce FRANK, a strict functional language supporting computational effects and effect handlers, which motivates our technical developments in subsequent chapters, and influences the design of our effectful language ELLA introduced in Chapter 6.

The design of FRANK is inspired by the theory of algebraic effects [Plotkin and Power, 2001, 2002] and algebraic effect handlers [Plotkin and Pretnar, 2013]. There are several novel design features of FRANK, including its effect type system and approach to effect polymorphism. Perhaps the most significant departure from existing languages and calculi in this area is FRANK's approach to handling effects and defining effect handlers.

This chapter is based on the publications:

- Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 500–514, 2017. doi:10.1145/3093333.3009897;
- Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming (special issue on algebraic effects and handlers)*, 30:e9, 2020. doi:10.1017/S0956796820000039.

The journal article is an extended and updated account of FRANK. In comparison to the POPL'17 publication, the journal version describes a small-step operational semantics for the surface language, and extends the language with *adaptors*; FRANK's answer to the problem of *effect pollution* (see Section 3.11). I contributed the design

and implementation of the initial FRANK compiler as described in the POPL'17 publication. I contributed to the formalisation of adapting effects presented in the journal submission.

The majority of this chapter is an extended tutorial on programming in FRANK which is based on the tutorials presented in the above publications. We highlight FRANK's support for smoothly integrating effectful and pure code, and its novel approach to defining and using effect handlers. Some examples given in this chapter serve as case studies for the techniques to be introduced later in the dissertation.

In the latter part of this chapter, we present a larger example application not previously published elsewhere. We walkthrough the Frank implementation of type inference for the Hindley-Milner type system. This implementation is based on Gundry et al.'s [2010] 'variables-in-context' algorithm for uniformly solving type inference and unification problems (see also [Gundry, 2013]). The goal is to demonstrate an implementation of the algorithm using an effects and handlers approach in contrast to Gundry's monadic implementation.

The implementation of FRANK is available online at the following URL:

`https://www.github.com/frank-lang/frank`

3.1 Pure Functional Programming

FRANK supports standard pure functional programming concepts such as functions and algebraic (inductive) datatypes. We may define standard examples of inductive datatypes as follows.

```
data Zero =
data Unit = unit
data Bool = tt | ff

data Nat       = zero    | suc Nat
data Maybe X   = nothing | just X
data Pair  X Y = pair X Y
data List  X   = nil     | cons X (List X)
```

A datatype definition is introduced using the keyword `data` followed by the name for the type. Possible instances of the type — the data constructors — are separated by '`|`'. In the above snippet, we defined the datatype `Bool` with two constructors `tt:Bool`

and `ff:Bool`. In the case of the `Zero` type there are no constructors, denoting the empty type. A datatype may take a list of type parameters which can appear in their constructors, such as the `Pair` type which has one data constructor `pair`, containing a value of type `X` as its first argument and a value of type `Y` as its second argument. By convention, type constructors are given uppercase initials, as are their parameters, whereas data constructors are lowercase initially. Both type and data constructors are prefix. Datatypes may be recursive as demonstrated by the definition of `List X` where the `cons` constructor takes a value of type `List X` as its second argument. Data constructors are not curried and must be fully applied (cf. Haskell [Marlow, 2010]).

In practice, while we may define such primitive datatypes directly, for convenience, many of the above types are given built-in status. In particular, the type of lists is built-in and supported by convenient syntactic sugar.

```
nil = []    cons x nil = [x]    cons x xs = x :: xs
```

The remaining built-in types are characters (`Char`), integers (`Int`), and strings (`String`) — which are really `List Char`.

Besides top-level datatype declarations, FRANK supports defining top-level functions. Consider defining an append function for `Lists`. First, we declare its type signature:

```
append : {List X -> List X -> List X}
```

Ignoring the braces (‘{}’) for the moment, the above type declares `append` to be a function accepting two arguments, both lists containing elements of type `X`, returning another list of the same type. The `X` occurring in the type signature is a (value) type variable which is implicitly universally quantified over the entire signature. Using ‘`->`’ between arguments may be slightly misleading; like data constructors, FRANK functions are not curried and must be fully applied.

Functions are defined using pattern matching clauses. We may define `append` by pattern matching on the first argument as follows.

```
append    []     ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

Example 3.1 (Append Lists)**.** We may apply `append` to any lists containing the same element type. For example,

```
append [1,2] [3,4] ⟹ [1,2,3,4]
```

where we informally denote evaluation using $\Longrightarrow$.

3.2 Doing and Being

In developing notions of computation, Moggi [1989a] made a distinction between pure values and effectful computations. A value is an inert datum without any computational character, e.g. the integer 5, whereas an effectful computation may alter or interact with the external world during its evaluation. This distinction is exemplified in Levy's call-by-push-value (CBPV) calculus [Levy, 2004] which treats values and computations as separate syntactic categories; hence Levy's slogan, "a value is, a computation does." We briefly alluded to this distinction in Chapter 2, using **return** v to lift a value v to a computation. Inspiration for FRANK is drawn from Levy's CBPV in that we separate value and computation types, and programming in FRANK amounts to alternating between *doing* (performing) a computation and *being* (returning) a value.

The full form of a computation type in FRANK is quite complicated at first sight. We elect to give the definition in stages. Here is the first candidate.

Definition 3.2 (Computation Type I)**.** A computation type C is an arrow type

$$A_1 \to \cdots \to A_n \to B$$

where the A_i's represent argument value types, and the B represents a result value type.

The value types (ranged over by A, B) consist of value type variables (ranged over by X), datatypes and *suspended* computation types. A suspended computation type injects a computation type C into a value type denoted by $\{C\}$. Suspended computation types are known as *thunk* types in CPBV, written $U\,\underline{B}$, for computation type $\underline{B}$.

In FRANK — as in CPBV — a value has a value type, a computation has a computation type, and variables bind only to values. Thus, we may bind variables to suspended computations. When executing or applying a suspended computation we sometimes say its evaluation has been *forced*, borrowing Levy's terminology. Given a variable `f` bound to a suspended computation, we write `f!` to force its evaluation. It corresponds to `force f` in CPBV. Unless `f` is nullary we omit the '!' as a syntactic convenience.

Example 3.3 (Append Type)**.** The type ascribed to the `append` function above is a suspended computation type `{List X -> List X -> List X}`. Wrapping a computation type within braces ('{}') injects it into a value type. The `append` variable is bound to a suspended computation.

By allowing suspended computations as arguments, FRANK is capable of expressing standard higher-order functions such as `map`:

```
map : {{X -> Y} -> List X -> List Y}
map       f          []     =
map       f      (x :: xs) = f x :: map f xs
```

with the usual operational meaning.

Example 3.4 (Adding One)**.** We may use our newly defined higher-order `map` function to add one to every integer in a list:

```
map {n -> n + 1} [1,2,3] ⟹ [2,3,4]
```

This example defines an anonymous suspended computation, `{n -> n + 1}`. In general, anonymous suspended computations consist of a sequence of pattern matching clauses separated by '`|`', all enclosed within braces. A nullary computation is just written within braces. Just like top-level definitions, fully applying them forces their evaluation. In CPBV, the suspended computation could be written: `thunk` $(\lambda n.n+1)$.

3.3 Computational Effects

FRANK supports computational effects in a style inspired by the theory of algebraic effects. In particular, effects are represented by the sources that give rise to them: operation symbols with associated signatures.

The programmer may define their own effects using *interfaces*. An interface, akin to an algebraic signature, is introduced using the keyword **`interface`** and consists of the collection of operations corresponding to the sources of the effect being defined. These operations, known as *commands* in FRANK, are given in their generic effect form. In particular, a command c is given by $c : A_1 \to \cdots \to A_n \to B$, where the A_i's are the n argument value types and B is the result value type. Much like datatypes, interfaces accept type parameters which may be referenced by the commands. Some examples of interfaces are as follows.

Example 3.5 (Abort)**.** Recall the theory of exceptions can be given by (assuming a single exception) the generic effect operation $\mathbf{throw}[\mathbf{unit}] : \mathbf{zero}$. In FRANK, this effect is defined by:

```
interface Abort = raise : Zero
```

where `Abort` is the name of the effect consisting of a single nullary operation `raise` which returns a value of type `Zero` when invoked.

Example 3.6 (State)**.** We model the global state signature presented in Example 2.8. For simplicity, assume we are dealing with only one state cell (so L has cardinality 1). Thus, the operations are:

$$\mathbf{lkp}[\mathbf{unit}] : S \qquad\qquad \mathbf{upd}[S] : \mathbf{unit}$$

which can be modelled in FRANK by:

```
interface State S = get : S
                  | put : S -> Unit
```

The `State` interface has two commands, `get` and `put`, for manipulating a single state cell of type `S`. The `put` command expects an argument of type `S` and returns a value of type `Unit`. The `get` command is nullary and returns a value of type `S`. When multiple commands make up the signature of an effect, they are separated by '`|`'.

Example 3.7 (Granularity of Effects)**.** It is possible to divide the above `State` effect into two separate interfaces: a sender and a receiver.

```
interface Send S = send : S -> Unit

interface Recv S = recv : S
```

We shall see later how we may combine `Send` and `Recv` to mimic the behaviour of the composite `State` effect.

Now that we have defined a number of effects, we would like to program with them. The FRANK type system statically tracks effects occurring in the program. We record which effects may be used as part of computation types. It is time to generalise the form of a computation type we gave earlier.

Definition 3.8 (Computation Types II)**.** A computation type C is an arrow type

$$A_1 \to \cdots \to A_n \to [\Sigma]B$$

where the A_i's are argument value types as before. The key difference is the result type which is now a composite of an *ability* Σ and a value type B. Syntactically, an ability consists of a sequence of effect interfaces. Semantically, an ability specifies which effects may occur when the computation is executed.

Given this updated definition, we can explain the type ascribed to commands from an effect interface. Unlike data constructors, commands are first-class values. A command c from effect interface I with signature $c : A_1 \to \cdots \to A_n \to B$ has the type

$\{A_1 \to \cdots \to A_n \to [I]B\}$. Intuitively, a command is a source of its defining effect so it gives rise to a suspended computation which, when forced, causes the effect.

Example 3.9 (Storing Data)**.** Consider defining a computation that stores an integer value in a state cell. We could use the `State` effect to write this program with the type:

```
sendInt : {Int -> [State Int]Unit}
```

Since the command `put` has type `{S -> [State S]Unit}` we can define `sendInt` simply:

```
sendInt n = put n
```

where the type `S` is instantiated to `Int`.

The above example demonstrates we can define top-level suspended computations which are not pure functions. We introduce the term *operator* to refer to top-level suspended computations with types captured by Definition 3.8 and its subsequent refinements. Thus, a pure function is an operator where the ability is the empty sequence. We retain the term 'suspended computation' to refer more generally to operator arguments with types captured by Definition 3.8 which may be an operator or a command.

3.4 Effect Polymorphism

Effect polymorphism in FRANK is supported by abilities. We gave a simplified account of abilities in the last section.

Abilities are not just a sequence of concrete effect instances, but also come with a *seed*. The seed is either $\varnothing$ which represents purity, or ε which is the distinguished effect variable standing for the effects permitted by the environment — the *ambient ability*.

Secondly, abilities may contain multiple instances of the same effect. By convention, the right-most occurrence is the *active* instance of the effect, meaning that command invocations resolve to that instance. We say the right-most occurrence *shadows* the other instances occurring in the ability. In the presence of effect polymorphism multiple instances are quite useful, allowing a program to occur within any effectful context. Section 3.12 discusses manipulating the order of effect instances.

Example 3.10 (Some Abilities)**.** We give some examples of abilities with seeds.

$[\varnothing]$	no effects occur
$[\varepsilon]$	only effects supported by the environment occur
$[\varepsilon \mid \text{Send Int}]$	any effects supported by the environment, plus sending integers
$[\varnothing \mid \text{State Int}]$	only state commands for `Int` occur
$[\varnothing \mid \text{State Int}, \text{State Bool}]$	two `State` instances but only state commands for `Bool` occur

A novel feature of FRANK is that, by default, all operators are *implicitly* effect polymorphic. The type signatures assigned to operators in Section 3.1 are polymorphic in the ambient ability, meaning they may be freely used within a pure or effectful context.

An interesting consequence is illustrated by the type signature of `map`. The previous signature given for `map` is syntactic sugar for:

```
map : {{X -> [ε]Y} -> List X -> [ε]List Y}
```

We see that `map` permits its first argument to perform any and all of the effects supported by the ambient. Another way to read the type is that `map` enforces the effects of its first argument to match exactly the effects supported by the environment.

Example 3.11. We can leverage the implicit effect polymorphism of `map` to perform an effectful task for all elements in a list.

```
sendList : {List X -> [Send X]Unit}
sendList      xs    = map send xs; unit
```

where '`x; y`' is sequential composition which returns the value of `y`.

Crucially, the `Send` interface appears in the ability of `sendList`. Hence, when `map` is invoked in the body of `sendList`, the ambient ability $[\varepsilon]$ contains the `Send X` effect. Consequently, we may invoke `send` commands on the list elements. Recalling the type ascribed to commands, we see how the typing of `map`'s first argument, `send`, matches the ability provided to `map` by the environment. If we were to omit the `Send` effect from the ability of `sendList`, the definition above would generate a compile-time type mismatch.

Given the more explicit type signature for `map`, we reproduce the definition given earlier in Figure 3.1. Comparing with other functional languages, our definition of `map` is analogous to HASKELL's `map` function:

```
map : {{X -> [ε]Y} -> List X -> [ε]List Y}
map       f             []     =
map       f          (x :: xs) = f x :: map f xs
```

Figure 3.1: The explicit map definition.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

However, HASKELL's `map` is a pure function whereas our typing permits effectful computations to occur during the processing of the list elements. So, our type signature is as general as `map`'s monadic variant:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

Hence, we have written something as permissive as ML's `map` yet retain control of effects.

There is no effect inference in FRANK. The ambient ability makes clear which effects are possible at any program point. Operator type signatures always specify an ability but FRANK's support for convenient syntactic sugar allows the programmer to omit it in some cases. Application sites push effects *inwards*, locally expanding what is available.

In designing Frank we have sought to maintain the benefits of effect polymorphism whilst avoiding the need to write effect variables in source code. While we may be explicit about their presence if desired, there are no explicit effect variables in any of the examples in this chapter.

3.5 Evaluation Order

FRANK has a strict left-to-right evaluation strategy. So far we have only seen examples of operators being applied to values. More generally, an operator is applied to potentially effectful computations which are evaluated to values in left-to-right order [1]. The next example highlights computations-as-arguments and the effect polymorphism implicit in computation types.

[1] Evaluation to values is in keeping with Definition 3.8. We refine this viewpoint in Section 3.8.

Example 3.12. We revisit Example 3.1 using an effect to retrieve one of the lists from some source. In this example, the effect polymorphism in type signatures is made explicit.

```
combine : {List X -> [ε|Recv (List X)]List X}
combine ys = append recv! ys
```

where append operator has type signature:

```
append : {List X -> List X -> [ε]List X}
```

which says append is polymorphic in the effects provided by the environment. The environment, or ambient ability, of append corresponds to the ability specified in the result type of `combine` which contains the `Recv (List X)` effect. So, the arguments of append may invoke `recv` commands as part of a computation which must eventually produce a list of type `X`; in particular, the computation `recv!` is a permitted argument to append.

By exploiting the distinction between a value and a computation, we can define standard control mechanisms as operators. For the sequential composition operators, we evaluate both arguments before returning the desired value.

```
fst : {X -> Y -> X}     snd : {X -> Y -> Y}
fst   x   y  = x        snd   x   y  = y
```

Due to implicit effect polymorphism, either argument may be effectful. FRANK's evaluation order prioritises the effects in the first argument before those in the second argument. We see that `snd x y` behaves as sequential composition `x; y`.

Occasionally, we may want to avoid evaluating an argument upon application. This behaviour is achieved by suspending the computation.

Example 3.13 (Conditional Branching)**.** Consider defining the traditional conditional operator:

```
if : {Bool -> {X} -> {X} -> X}
if   tt     t     f   = t!
if   ff     t     f   = f!
```

The second and third arguments to `if` are suspended computations which return values of type `X`. By using suspended computations, evaluation of the branches is prevented. Once we know the result of the condition, we can explicitly force the desired branch to execute. The `if` operator can be used to find the meaning of life:

```
if ultimateQuestion! {42} {0}
```

where `ultimateQuestion:{Bool}`.

Frank has a *bidirectional* type system where terms are distinguished by whether their types are *inferred* or *checked* [Pierce and Turner, 2000; Pfenning, 2004]. An advantage of bidirectional type checking is that in many common cases type annotations can be omitted by switching between checking and inference contexts. In particular, anonymous suspended computations can omit type annotations whenever they can be checked against a type, e.g. when supplied as an argument to a higher-order operator. In Example 3.4, we saw an example of this where `{n -> n + 1}` could be supplied unadorned with type information, its type being inferred through unification of the arguments to `map`.

Note however that we may not write `{n -> n + 1} 2`, for example, because in this case the anonymous suspended computation appears in an inference context, not a checking context. One solution is to bind the definition to an operator, `addOne`, ascribed with a type:

```
addOne : {Int -> Int}
addOne n = n+1
```

which can then be used as `addOne 2` since we can infer the type of `addOne`.

Alternatively, we may define an operator which puts the anonymous suspended computation into a checking context and solves for its type through unification:

```
case : X -> {X -> Y} -> Y
case   x        f     = f x
```

Reversing the order of application results in more aesthetically pleasing code and mimics inline case expressions. For example, we may define short-circuit AND by case splitting on the result of the first computation.

```
shortAnd : Bool -> {Bool} -> Bool
shortAnd    x        c     = case x {tt -> c! | ff -> ff}
```

3.6 Direct Style for Monadic Programming

In Example 3.12 we saw that FRANK supports direct applicative style programming. That is, we can directly supply one computation as the input to another via, for example, an application. A key feature enabling this style is the clear separation of effects

from values in the syntax of types. By contrast, HASKELL does not have such a clear type-level distinction and programs written in HASKELL which perform effects utilise do-notation so as to disambiguate *effect values* from pure values.

Example 3.14. Consider defining a computation which repeatedly receives lists, concatenating them together, until one is empty. In HASKELL, the signature for our receiver would be given by:

```
catter :: Receive (List a) (List a)
```

where `Receive r a` is the monad for a computation which receives values of type `r` and returns a value of type `a`. The `Receive` monad is defined by the algebraic datatype:

```
data Receive r a = Recv (r -> Receive r a)
                 | Return a
```

The `Monad` instance for this datatype is defined by [2]:

```
instance Monad (Receive r) where
  return x = Return x
  Return x >>= f = f x
  Recv g >>= f = Recv (\r -> (g r) >>= f)
```

Thus, `catter` is a value representing a monadic computation, performing `Receive` effects when executed; its type does not correspond to a pure value but an *effect value*. However, the type signature `Receive (List a) (List a)` does not disclose its effectful character. Indeed, it looks identical to a pure value type signature such as `Pair (List a) (List a)` or `Either (List a) (List a)`. HASKELL programmers use do-notation to unambiguously parse a type as an effect value:

```
catter = do
  xs <- recv
  case xs of
    [] -> return []
    xs -> do ys <- catter; return (xs ++ ys)
```

where the `recv :: Receive r r` operation requests a value of type `r` from the enclosing environment. Hence, `recv` is a monadic computation and cannot be supplied

[2] We are omitting the instances for `Functor` and `Applicative`, but their definitions perform similar plumbing of the received value through their respective operators.

directly to HASKELL's `case` construct which only operates on pure values. So, we bind `recv` to `xs` using `do`-notation which has the consequence of executing the monadic computation and assigning the value result to `xs`.

FRANK's version of `catter` can be expressed in direct style, without the need for extra plumbing like `do`-notation. However, as discussed in Section 3.2, we must still distinguish between *doing* an effectful computation and *being* its suspended counterpart. For example, we saw that `recv!` represents *doing* an effectful computation which performs `Recv` effects whereas `recv` is a first-class value, a suspended computation of type `{[Recv X]X}`, which produces a `Recv` effect only when forced to do so [3].

Example 3.15. Using the distinction between values and computations, effectful computations can be invoked in-place by explicitly forcing them to execute. We recast the concatenation program in FRANK:

```
catter  : {[Recv (List X)]List X}
catter! = case recv! { [] -> []
                     | xs -> append xs catter!}
```

Note the equation for `catter` has a '!' symbol left of '='. Its inclusion is to highlight the fact that `catter` is bound to a *suspended* computation which has the behaviour of its defining equation only when forced.

Consider a second example, implementing the C/C++ postfix increment operation. The operation fetches the current value of a stored integer, increments it and stores the result before returning the original value. In the following code snippets, we assume a standard implementation of state (Example 2.2) which, in particular, satisfies equation (1) in Example 2.8. HASKELL programmers must introduce bindings to express the desired evaluation order explicitly.

```
next :: State Int Int
next = do x <- get
          put x+1
          return x
```

On the other hand, FRANK avoids explicit plumbing due to its fixed evaluation order, leading to a more straightforward, direct implementation:

```
next  : {[State Int]Int}
next! = fst get! (put (get! + 1))
```

[3] Recall the use of '!' forces a suspended computation to evaluate.

3.7 Polymorphic Commands

Recall from Example 3.5 the definition of the `Abort` effect:

```
interface Abort = raise : Zero
```

We would like to use the `Abort` effect to cease the execution of *any* computation when an error condition is triggered. To accomplish this, we need a command which can be invoked at any type. The `raise` command alone does not suffice since it returns a value of the empty type, `Zero`. However, since `Zero` is uninhabited we can eliminate the value returned by `raise!` by empty case analysis:

```
abort : {[Abort]X}
abort! = case raise! {}
```

where `abort` is a nullary operator polymorphic in its return value `X`.

The original version of FRANK only supported parametric commands, requiring the above gymnastics to derive a polymorphic type for a command. Convent [2017] added support for *polymorphic* commands [Kammar et al., 2013]. A polymorphic command c is given by $c\ \overline{X} : A_1 \to \cdots \to A_n \to B$, where A_i's and B are value types, as before, and the X_j's in $\overline{X}$ are value type variables which are universally quantified over the command's signature, *i.e.* they may appear in the A_i's or B.

Example 3.16 (Polymorphic Abort)**.** We can define a simpler version of `Abort` using polymorphic commands as follows.

```
interface Abort = abort X : X
```

The `abort` command universally quantifies over the value type variable `X`, accepts no arguments, and returns a value of type `X`. Thus, `abort` is a polymorphic command and may be instantiated at any type. It is guaranteed to end computation as there is no way to provide its continuation with a value for a generic `X`.

3.8 Effect Handlers: Handling by Application

In call-by-value functional programming, function application has a straightforward semantics. First, the term in functional position is reduced to a lambda abstraction then the argument to the function is reduced to a value. Beta reduction follows substituting the argument value into the body of the function. The resulting term is reduced to produce a final value.

Existing languages incorporating effects and handlers introduce a special construct for handling effects akin to the **handle-with** construct we introduced in Chapter 2; see Section 3.14 for more on these languages. In contrast, FRANK generalises the notion of function application from call-by-value languages to encompass effect handling since, as mentioned in Section 3.5, arguments to operators may be effectful computations. Thus, an operator generalises both call-by-value functions and effect handlers, and operator application is synonymous with both call-by-value function application and effect handling. Incorporating effect handling into operator definitions requires one final refinement of the computation type definition for FRANK.

Definition 3.17 (Computation Types III)**.** A computation type C is an arrow type

$$\langle \Delta_1 \rangle A_1 \to \cdots \to \langle \Delta_n \rangle A_n \to [\Sigma] B$$

where the key difference from Definition 3.8 is the arguments are now composed of an A_i value type, as before, and an *adjustment* Δ_i, written between chevrons. Syntactically, an adjustment is a sequence of effect interfaces. Semantically, an adjustment specifies the effects which are *handled* (i.e. interpreted) by the operator at the respective argument position. An adjustment extends the ambient ability for a particular argument position — by convention, the identity extension is omitted.

Given Definition 3.17, a function is simply a special case of an operator where the adjustment is the identity extension. A unary operator with a non-identity adjustment is an effect handler and a *multihandler* is the n-ary counterpart where two or more arguments have non-identity adjustments.

The pattern grammar for pattern matching clauses is extended to facilitate the handling of effects. We recall the `State` interface given in Example 3.6:

```
interface State S = get : S
                  | put : S -> Unit
```

Example 3.18 (Interpreting State)**.** We can define an effect handler for handling stateful computations capable of manipulating one global memory cell. We define a binary effect handler `state` where the first argument carries the current value of the cell, and the second argument is the stateful computation to be executed. The signature of `state` is:

```
state : {S -> <State S>X  -> X}
```

declaring `state` to be an operator with two arguments where `S` is a type variable representing the type of values which are storable, and `X` is a type variable representing the type of the eventual return value of the stateful computation.

The type signature for `state` specifies an adjustment in the second argument position specifying that the operator handles commands from the `State S` effect. Thus, in the second argument the ambient ability is extended to include the `State S` effect, permitting `get` and `put` to be invoked.

How do we define the `state` operator? If the second argument to `state` is a value of type `X`, we can just return it as the overall result:

```
state  _        x      = x
```

Since the pattern `x` only matches *values* there are still more cases to consider. In particular, FRANK has patterns to describe the handling of a command known as *request* patterns. These request patterns consist of a command paired with a continuation representing the rest of the computation. The adjustment of the second argument makes clear there must be at least one clause which matches each command for `State` [4]. Therefore, we must specify how to handle `get` and `put` commands invoked by the second argument. For `get`, our stateful computation is *requesting* the current value of the store.

```
state  s    <get -> k>  = state s (k s)
```

A request pattern is given between chevrons containing: the name of the command, followed by arguments to the command, if any; the symbol `->`; and a pattern variable representing the continuation, in this case given by `k`.

Handling `get` requires we provide the current value of the store to k; thus, we apply k to s. Recall FRANK has shallow handlers which do not automatically re-invoke the handler on the continuation. Thus, we explicitly re-invoke `state` by applying it to the current store value and the rest of the stateful computation in order to handle further stateful commands.

Lastly, we must handle `put` which involves updating the store with the new value we have been provided:

```
state  _    <put s -> k> = state s (k unit)
```

[4] Checking that there is at least one matching clause for any sequence of possible arguments is known as *coverage* checking, and for a collections of clauses which satisfy it, we say they *cover* their arguments. While we insist that pattern matching clauses cover their arguments, we do not formalise the notion in this dissertation. See Lindley et al. [2017] for an algorithm which can be used to check coverage for FRANK.

```
state : {S -> <State S>X  -> X}
state    _          x       = x
state    s     <get -> k>   = state s (k s)
state    _    <put s -> k> = state s (k unit)
```

Figure 3.2: The `state` effect handler for handling stateful computations.

again, invoking the continuation and applying the `state` handler. The `state` handler as a whole is presented in Figure 3.2.

Example 3.19 (Stateful Computation)**.** The following example demonstrates the `state` operator interpreting the `get` and `put` commands invoked by its second argument. Define the following indexing function:

```
index : {List X -> List (Pair Int X)}
index     xs     = state 0 (map {x -> pair next! x} xs)
```

which can be used in an example program manipulating a single state cell:

```
index "abc" ⟹ [pair 0 `a', pair 1 `b', pair 2 `c']
```

As demonstrated by `state`, shallow handlers can straightforwardly express deep handlers using explicit recursion. Deep handlers have also been shown to encode shallow handlers [Kammar et al., 2013; Hillerström and Lindley, 2018].

3.9 Handling on Multiple Arguments

Through n-ary operators, FRANK permits the programmer to handle distinct adjustments at each argument position leading to so-called multihandlers. In contrast, existing languages based on effects and handlers have restricted support to unary handlers [Bauer and Pretnar, 2015; Hillerström and Lindley, 2016; Leijen, 2017; Biernacki et al., 2019b]. We give a concrete example of defining and using a multihandler in this section: the prototypical `pipe` multihandler.

Example 3.20. The `pipe` multihandler mediates communication between a producer and a consumer. The type signature for `pipe` is:

```
pipe : {<Send X>Unit  ->  <Recv X>Y  -> [Abort]Y}
```

expressing the communication by handling commands of `Send X` on its first argument and commands of `Recv X` on its second argument. The first argument returns a value of type `Unit`, whereas the second may have any value type, given by type variable `Y`, which is also the overall result type of the operator. Additionally, the operator may `Abort` during its execution as specified by its ability.

Defining `pipe` by pattern matching, the first equation facilitates the communication between producer and consumer:

```
pipe   <send x -> s>    <recv -> r>  = pipe (s unit) (r x)
```

providing the sent value `x` to the consumer computation `r`. We re-invoke `pipe` on the continuations of each argument to handle further communication.

What if the receiver is finished communicating? Then the second argument returns a value of type `Y` which we may return as the overall result. We do this no matter whether the first argument is a value or a `send` command, meaning any further `send` commands are discarded. This semantics is expressed by the second equation:

```
pipe        <_>              y       = y
```

where `<_>` is a catch-all pattern matching *either* a unit value or a `send` command. The catch-all pattern matches against only those commands which are to be handled at this argument position. Any other commands are automatically *forwarded* to the nearest dynamically enclosing handler for that command. No operator is allowed to intercept commands for effects which are not specified by the adjustments in its type.

The third and final equation says what to do if the consumer expects a value but the producer has stopped sending. We consider this to be a communication error and an `abort` is raised to indicate that:

```
pipe        unit            <_>      = abort!
```

There seems to be a certain amount of arbitrariness in the definition of `pipe`. The above clauses prioritise receiving data such that any surplus is discarded and any deficit is an error. We discuss the implications of alternative definitions in Chapter 7.

The complete definition of `pipe` is given in Figure 3.3 for reference. Using `sendList` from Example 3.11 and `catter` from Example 3.15, `pipe` may be used as follows.

```
pipe (sendList ["do","be",""]) catter! ⟹ "dobe"
```

where the empty string in the sent list signifies to `catter` that communication has ceased.

```
pipe : <Send X>Unit  ->  <Recv X>Y  -> [Abort]Y
pipe   <send x -> s>    <recv -> r>  =
  pipe (s unit) (r x)
pipe        <_>              y       = y
pipe        unit            <_>      = abort!
```

Figure 3.3: The complete pipe multihandler definition.

Furthermore, multiple instances of `pipe` can be combined with a mediator which both produces and consumes. For example, given

```
spacer  : {[Send String, Recv String]Unit}
spacer! = send recv!; send " "; spacer!
```

the example above can be amended to

```
pipe (sendList ["do","be",""])
     (pipe spacer! catter!) ⟹ "do be "
```

where `recv` commands generated by `spacer` are picked up by the outer `pipe` while the inner `pipe` interprets its `send`s. In this example, we see implicit *forwarding* in action. When `spacer!` invokes the `recv` command, the match against `pipe`'s first column of patterns fails because `pipe` does not handle `Recv` in its first argument. Therefore, the `recv` command is forwarded to its nearest handler, in this case the outer `pipe`.

We expect `pipe` to be associative so that:

```
pipe (pipe sendDoBe! spacer!) catter! ⟹ "do be "
```

where

```
sendDoBe! = sendList ["do","be",""]
```

Indeed, the techniques developed in Chapter 6 will be applied in Chapter 7 to prove `pipe` is associative for all possible arguments.

It is worth noting that any program defined using simultaneous multihandling can in fact be expressed using mutually recursive unary handlers [Lindley et al., 2017]. Indeed, Kammar et al. [2013] implement deep and shallow *unary* handler versions of pipe using their Haskell effects library. The implementations are significantly more complex than the one presented here, requiring the handlers for each process to maintain a suspended computation of its partner to sustain the interaction. In particular, the deep handler implementation depends on a non-trivial mutually recursive data type.

While other systems such as Multicore OCaml [Dolan et al., 2015] and Eff [Bauer and Pretnar, 2015] offer improved syntax over Kammar et al.'s library, a programming burden remains. We believe the FRANK approach is cleaner, leveraging well-known functional programming techniques (pattern matching and recursion) to completely capture the interaction.

3.10 Exception Handling

FRANK's effect type system sometimes leads to more precise typing than seen with other systems. Consider implementing an exception handler:

```
catch : <Abort>X    ->  {X} -> X
catch       x           _   = x
catch   <raise -> _>  h   = h!
```

The first argument is a computation which may abort. The second argument is an alternative computation to run in the case of failure. There is no constraints on the ambient ability in which `catch` is executed. In particular, recall from Section 3.4, that abilities may contain multiple effect instances. So, the ambient ability may or may not provide the `Abort` effect; determined at the call site for `catch`. In contrast, consider the typing of catch given by the Haskell mtl library [Gill, 2018]:

```
catchError :: MonadError () m => m a -> (() -> m a) -> m a
```

Here, `catchError` is given a type such that both argument computations may raise an exception. The typing does not restrict abort to be a local ability of the first computation. Leijen [2014] makes a similar observation in KOKA's treatment of exceptions (see Section 3.14 for more on KOKA).

Due to implicit effect polymorphism, FRANK's typing is more precise: permitting the alternative computation to `abort` if and only if the environment offers to handle it. Additionally, `catch` may only handle `abort` commands of the first computation argument, whereas the type of `catchError` is not similarly constrained.

3.11 Effect Pollution

Most of the examples we have presented so far have showcased single (multi)handlers for a collection of effects. One motivation for choosing effects and handlers as a repre-

sentation for computational effects is their composability. We saw how straightforward it is to compose handlers in Section 3.9 where we composed multiple instances of pipe.

Suppose we want an abstraction for communication which does not raise `Abort` effects when an error is encountered. We could alter our original definition for `pipe` to use the `Maybe` result type:

```
data Maybe X   = nothing | just X
```

but then we are stuck with a single interpretation of failure: `nothing`.

A better approach, more aligned with the ethos of effect handling, is to *interpret* `Abort` effects using `nothing`, as done by the following `Abort` handler:

```
maybe : <Abort>X    -> Maybe X
maybe       x        = just x
maybe  <abort -> _>  = nothing
```

Example 3.21 (Pipe or Nothing)**.** Now, we just compose our original `pipe` multihandler with the above `maybe` handler for `Abort`.

```
maybe (pipe (sendList ["do","be"]) catter!) ⟹ nothing
```

where the above communication omits the terminating empty string from the sent list for the purposes of illustrating the error condition. Recall, `abort!` is invoked if the number of `recv` commands exceeds the number of `send` commands.

Suppose we wish to encapsulate this abstraction into an operator definition which just re-uses the existing handlers `maybe` and `pipe`:

```
nogood : {<Send X>Unit  ->  <Recv X>Y  -> Maybe Y}
nogood        <s>              <r>      = maybe (pipe s! r!)
```

Delegating to `pipe`, we capture the arguments to `nogood` as suspended computations and execute them in the context of the `maybe` and `pipe` handlers. Unfortunately, this definition does not type check for the following reason. The bindings `s` and `r` are suspended nullary computations with abilities `[Send X]` and `[Recv X]`, respectively. Now, `s` and `r` are invoked in a context enclosed by the `maybe` effect handler which extends the ambient ability with `Abort`. So, at the point of invocation, the abilities of `s` and `r` are expected to unify with `[Abort,Send X]` and `[Abort,Recv X]`, respectively, which they do not.

Perhaps an obvious attempt at a solution is to just add `Abort` to the adjustments of the type signature. Then the above definition type checks without further modification:

```
leaky : <Send X,Abort>Unit  ->  <Recv X,Abort>Y  -> Maybe Y
leaky          <s>              <r>     = maybe (pipe s! r!)
```

Unfortunately, there are some issues with the type of `leaky`. The intermediate `Abort` effect, produced by `pipe` and handled by `maybe`, has leaked into the argument types for `leaky`, revealing implementation details which should remain hidden. Perhaps even worse, since an adjustment mentions only the effects which are handled, the above typing leads to the following capturing behaviour:

```
leaky (send "drip"; abort!) recv! ⟹ nothing
```

The first computation raises `abort` which is interpreted by `leaky` — surely not what was intended! Abort effects occurring in the argument computation should be forwarded to the outer context for handling rather than being interpreted by `leaky`. The `leaky` pipe has broken abstraction by revealing its interpretation for a communication mismatch in the type of its arguments. We call this phenomenon *effect pollution*. To prevent such effect pollution we need a way of *hiding* intermediate effects so that external effects with the same name are not accidentally handled.

Biernacki et al. [2018] were the first to describe effect pollution — albeit with a slightly different manifestation — in the context of their effect calculus $\lambda^{H/L}$. They present an abstraction violation for effect polymorphic arguments to higher-order operators. Incidentally, this problem can also occur in FRANK but we present the problem with adjustments instead because it can be expressed in less code [5]. To solve their problem, Biernacki et al. introduce an operation called *lift* which extends the ability of an expression with an additional effect. It turns out, lift suffices to resolve the issues with `leaky` as well.

We extend FRANK with an operation called *mask* which restricts the ambient ability by removing an effect instance. We adopt a different name for our solution because while lift and mask achieve the same outcome, their mechanism should be understood slightly differently. The difference arises due to FRANK being based on a bidirectional type system where effects are pushed inwards, whereas $\lambda^{H/L}$ is based on Hindley-Milner type inference where effects flow outwards. For example, to achieve an ambient ability of `[Send X]` at the point of invoking `s`, we remove (i.e. mask) the `Abort` effect from the ambient ability before invoking `s`. FRANK supports masking through a more general construct called an *adaptor* (see Section 3.12).

[5] The reader interested in the FRANK rendering of Biernacki et al.'s example is encouraged to consult Appendix A.

Example 3.22. To achieve the desired typing for our pipe abstraction, we apply an adaptor to the computation arguments, `s` and `r`, when invoked as arguments to `pipe`:

```
sealed : <Send X>Unit  ->  <Recv X>Y  -> Maybe Y
sealed   <s>  <r> = maybe (pipe (<Abort> s!) (<Abort> r!))
```

Using the "sealed" pipe, any `abort` commands invoked by the arguments are properly forwarded to the outer context.

```
sealed (send "drip"; abort!) recv! ⟹ abort!
```

3.12 General Rewiring Using Adaptors

The last section presented one particular use case for an adaptor, namely masking an effect instance occurring in the ambient. In fact, an adaptor is a map acting on effect instances, providing more general rewiring for an ability. An adaptor has the form $< I_i(s\ x_1 \cdots x_n \to s\ y_1 \cdots y_m) >_i$, a comma separated list of interface names paired with a mapping of their instances in the ambient (the x_i's) to their remapped configuration (the y_i's). The transformations on effect instances admitted by adaptors correspond to the structural rules of logic: weakening, contraction and exchange. We have already seen weakening which corresponds to mask. In this section, we show how adaptors support contraction and exchange. Contraction corresponds to duplication of an effect instance (see Example 3.26), whereas exchange corresponds to swapping effect instances (see Example 3.27). As with mask, which skips the nearest enclosing handler for the specified effect, all adaptors have computational content.

Consider the problem of parsing a natural number from a string and then returning its predecessor, if it exists. The complete solution is given in Figure 3.4 while the next example walks through the definition of the natural number parser.

Example 3.23 (Parsing $\mathbb{N}$)**.** Suppose we aim to parse strings of the form:

```
"z" = zero
"ssz" = suc (suc zero)
```

For simplicity, we assume there are no spaces contained within the strings. Additionally, we must decide what to do if the parsed string does not represent a natural number. Suppose we signal this error state by raising `abort`. The type signature for our number parser is given by:

```
pNat : {String -> [Abort]Nat}
```

```
pNat : {String -> [Abort]Nat}
pNat [] = zero
pNat (x :: []) = if (eqc x 'z') {zero} {abort!}
pNat (x :: xs) = if (eqc x 's') {suc (pNat xs)} {abort!}

pred : {Nat -> [Abort]Nat}
pred zero = abort!
pred (suc n) = n
```

Figure 3.4: Definitions for parsing numbers and computing their predecessors.

Recall that `String` is a synonym for `List Char`. Therefore, we can define `pNat` by pattern matching on a lists of characters. We take the empty list to be synonymous with `zero`:

```
pNat [] = zero
```

For the remaining cases, assume we have an equality operator on characters:

```
eqc : {Char -> Char -> Bool}
```

and that our conditional `if` operator (Example 3.13) is defined in scope. The final character of any string matching a natural number must be 'z'. Thus, we have a separate case for the singleton list:

```
pNat (x :: []) = if (eqc x 'z') {zero} {abort!}
```

Finally, the remaining case parses the successors:

```
pNat (x :: xs) = if (eqc x 's') {suc (pNat xs)} {abort!}
```

Our solution above does not a priori determine what to *do* for a particular error state: that is the job of a handler for `Abort`.

Example 3.24 (Effect Conflation)**.** One approach would be to conflate the two error conditions such that a single handler provides the error-handling for both states:

```
conflate : {String -> [Abort]Nat}
conflate s = pred (pNat s)
```

yielding the following results:

```
maybe (conflate "abc") ⟹ nothing

maybe (conflate "sssz") ⟹ just (suc (suc zero))

maybe (conflate "z") ⟹ nothing
```

Example 3.25 (Effect Distinction)**.** We may also choose to retain distinct instances of `Abort`, one for each error state, and handle them with separate handlers:

```
distinct : {String -> [Abort,Abort]Nat}
distinct s = pred (<Abort> (pNat s))
```

where the type signature specifies two separate instances of the `Abort` effect. Recall from Section 3.4 that abilities may contain multiple instances of an effect. There, we mentioned how the right-most instance is considered the active instance, shadowing any others occurring in the ability. Adaptors provide a mechanism to gain access to the other instances occurring in the ability.

In this example, we use mask to remove an instance of `Abort` from the ambient ability. Note, that the adaptor `<Abort>` is syntactic sugar for:

```
<Abort(s x -> s)>
```

using the more general mapping notation for specifying adaptors. Notice how the map for defining mask reflects the intuition given previously: it is removing an instance of `Abort` from the ambient ability — the instance corresponding to `x` — moving the other instances — represented by `s` — one position to the right. The `distinct` operator has the following behaviours on the example programs:

```
maybe (catch (distinct "abc") {zero}) ⟹ nothing

maybe (catch (distinct "sssz") {zero}) ⟹ just (suc (suc zero))

maybe (catch (distinct "z") {zero}) ⟹ just zero
```

Example 3.26 (Duplicating Effect Instances)**.** Using the general form for an adaptor, we can actually obtain the conflated version from the distinct version by *duplicating* an `Abort` instance:

```
conflate2 : {String -> [Abort]Nat}
conflate2 s = <Abort(s x -> s x x)> (distinct s)
```

Here, we remapped the first occurrence of `Abort` in the ambient to occupy the first and second occurrences in the subterm — `distinct s` — ensuring that, when `conflate2` is handled, any aborts raised will be handled by the same handler. For example,

```
maybe (conflate2 "abc") ⟹ nothing

maybe (conflate2 "sssz") ⟹ just (suc (suc zero))

maybe (conflate2 "z") ⟹ nothing
```

Example 3.27 (Swapping Effect Instances)**.** We may swap two instances of an effect by rearranging the order of their pattern variables in the mapping. Suppose we want to swap the error handlers for the parsing example:

```
distSwp : {String -> [Abort,Abort]Nat}
distSwp s = <Abort(s x y -> s y x)> (distinct s)
```

without changing the order of our `abort` handlers, `distSwp` inverts the error handling for the parsing and predecessor errors:

```
maybe (catch (distSwp "abc") {zero}) ⟹ just zero

maybe (catch (distSwp "sssz") {zero}) ⟹ just (suc (suc zero))

maybe (catch (distSwp "z") {zero}) ⟹ nothing
```

Combinations of mask, copy and swap can be used to build up quite sophisticated adaptors. In practice however, the basic operations seem to suffice for our current examples suite. Thus it would perhaps be beneficial to provide convenient syntactic sugar for these basic structural transformations. We already provide shorthand for masking an effect instance. Perhaps we could introduce keywords, e.g. `Abort(copy)` and `Abort(swap)` for duplicating and swapping occurrences of `Abort`, respectively.

3.13 Extended Example: Hindley-Milner Type Inference

In this section, we implement type inference for a Hindley-Milner type system [Milner, 1978] as an extended example of the effects and handlers paradigm. Specifically, we choose to implement the algorithm presented by Gundry et al. [Gundry, 2013; Gundry

$$
\begin{array}{lll}
\text{(Terms)} & M, N ::= x \mid \lambda x.M \mid MN \\
& \qquad\qquad \mid \ \textbf{let}\ x = M\ \textbf{in}\ N \\
\text{(Monotypes)} & S, T ::= \alpha \mid S \to T \\
\text{(Polytypes)} & P, Q ::= T \mid \forall \alpha.P
\end{array}
$$

Figure 3.5: HM calculus types and terms.

et al., 2010] because of its simplicity and its connection to the FRANK system itself: this algorithm inspired the implementation of FRANK's typechecker. Thus, being able to express the algorithm in Frank shows the applicability of effects and handlers to algorithms used in practice.

We provide a step-by-step description of how the algorithm is represented in FRANK. For brevity, we show only the data representations and most interesting or significant operators. Readers curious about the full implementation may consult the source code in the `examples/gundry-ml` folder in the FRANK repository.

Similarly to the above point, our description of the "variables-in-context" algorithm focuses on the essential points in order to understand our implementation, omitting a formal and language-independent description of it. Furthermore, we do not comment or compare to historical approaches to Hindley-Milner type inference. For a complete formal account of the algorithm we refer the reader to Gundry's thesis [Gundry, 2013].

3.13.1 The Calculus, Contexts, and Computational Effects

Following Gundry, we implement type inference for the Curry-style [6] simply-typed lambda calculus extended with polymorphic **let** expressions. The monotypes, polytypes, and terms of the calculus are presented in Figure 3.5. Variables bound by lambda abstractions have monomorphic type. Variables bound by **let** are assigned a polytype (also known as a *type scheme*), a type enclosed by $\forall$-quantified variables.

Gundry et al.'s algorithm uniformly characterises the problems of type inference and unification within the same constraint problem solving framework in contrast to traditional approaches which typically treat unification as a black box [Pierce, 2004, see Chapter 10]. A unification problem involves assigning values to unknown variables (or *metavariables*) in order to satisfy an equation. Type inference is the process of solving unification problems such that a typing judgement holds. Gundry et al. give

[6] Lambda abstractions are not annotated with the type of their bound variable [Sørensen and Urzyczyn, 2006].

a general account of both problems by tracking progress towards a solution in the context. That is, regular typing contexts are extended to keep track of metavariables as well as ordinary type and term variables. Contexts include both declarations and definitions of metavariables which are generated during type inference and solved for during unification. The full grammar for contexts is given by:

$$(\text{contexts})\ \Gamma ::= \cdot \mid \Gamma, x : P \mid \Gamma, \alpha : \star \mid \Gamma, \alpha := S \mid \Gamma ;$$

Context are dependency-ordered lists, permitting later entries to depend on earlier ones. They admit an *information* ordering between contexts based on simultaneous substitutions of types for metavariables. By substituting types for metavariables, the ordering yields a more informative context which constitutes the progress made towards solving a constraint problem. The type inference and unification algorithms seek to produce "information increases" which are minimal or least constraining. Formally, a substitution θ is a minimal solution if for any other substitution θ' which solves the problem there exists a substitution ζ such that, $\theta' = \zeta \circ \theta$.

A minimal solution is obtained by moving declarations in the context leftwards as little as possible. Given the dependency inherent in contexts, this movement increases the scope of a declaration — making it more global — and constraining its set of solutions. The ; marker acts as a locality separator: the information order may push a declaration left of a ; but such movements are irreversible, indicating the declaration has escaped the local scope delineated by ; and become more globally relevant.

In FRANK, we choose a de Bruijn [1972] representation for the calculus such that type and term variables are represented by natural numbers (or in this case, positive integers [7]).

```
data Ty = meta Int | var Int | arr Ty Ty

data Sch = mono Ty | poly Sch

data Tm = varx Int | lam Tm | app Tm Tm | seq Tm Tm
```

We distinguish between metavariables (`meta`) to be solved for by unification and rigid type variables (`var`) introduced by the $\forall$ quantifier. We implement appropriate bind and unbind operations to mediate between the two; we omit their definition as they are not

[7] Currently, FRANK does not automatically convert positive integers to their `Nat` encoding, so we use `Int` for indices to make the code more readable.

particularly interesting and their only point of use is in the definition of `specialise` (see below for a description of this operator).

Example 3.28 (de Bruijn Type Encoding)**.** As an example of the de Bruijn encoding, the type of the **K** combinator, $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$, is rendered in FRANK as:

```
kk : {Sch}
kk! = poly (poly (mono (arr (var 1) (arr (var 0) (var 1)))))
```

noting that the numbering for a bound variable indicates the position of its binder starting from zero for the innermost binder.

An `Entry` in a context is either a bound metavariable (`bmv`), a bound term variable (`btv`), or a locality marker (`mrk`).

```
data Entry = bmv Int (Decl Ty) | btv Sch | mrk
```

A term variable binding only requires the type scheme assigned to the variable since once added to the context, a term variable may be removed but its position is otherwise fixed. Therefore, the type of the variable (`varx n`) is always the n^{th} `btv` entry in the context. In contrast, a metavariable entry, `bmv`, may be moved during unification, so the data constructor `bmv` contains the metavariable's numeric identifier. Additionally, the metavariable binding contains its declaration information, `Decl Ty`, given by:

```
data Decl X = hole | defn X
```

which captures the two possible metavariable entries that may appear in a context: a declared yet undefined metavariable or a metavariable assigned a (possibly open) monotype.

Finally, we render the grammar for contexts given previously as backwards lists (or "snoc" lists) of *entries*:

```
data Bwd X = emp | snoc (Bwd X) X


type Ctx = Bwd Entry
```

where **type** creates a type synonym. Type synonyms are currently not implemented in FRANK, but they are akin to *interface aliases* [Convent, 2017] which are supported. We use them here simply to abbreviate the more verbose types on the right-hand side; the actual source code uses the more verbose type. We define another type synonym for suffixes:

```
type Suffix = List (Pair Int (Decl Ty))
```

A `Suffix` is a collection of metavariable declarations without any term variables or locality markers occurring between them. A suffix is collected during the type inference of the binding of a **let** expression. The variables contained in the suffix represent the variables which may be generalised over to create a type scheme for the binding.

The algorithm modifies a context so we require an environment for storing and updating the current context. We use our `State` effect to accomplish this:

```
interface CtxSt = [State Ctx]
```

The `CtxSt` effect is an *interface alias* [Convent, 2017; Convent et al., 2020] which assigns a name to a collection of effect interfaces. In this case, we have only one interface: `State` where its parameter has been instantiated to the type of contexts. Thus, the above definition is akin to defining:

```
interface CtxSt = get : Ctx
                | put : Ctx -> Unit
```

For generating new metavariables, we need a fresh supply of integers, which we represent by the following effect interface:

```
interface Names = fresh : Int
```

We reserve the `Abort` effect for unexpected errors signifying a bug in the implementation, and define a `TypeError` effect for expected errors generated by impossible unification problems:

```
interface TypeError = raise X : String -> X
```

where the argument to `raise` may be presented to the user.

Finally, we define two more interface aliases to provide fine-grained specifications of the abilities of subsequent operators:

```
interface NmSt = [CtxSt,Names]

interface Contextual = [Abort,NmSt,TypeError]
```

The `Contextual` effect represents the overall collection of effects for the type inference algorithm. It corresponds to the HASKELL monad of the same name in Gundry et al.'s implementation. However, in some cases we may only need a subcollection of commands from the full `Contextual` effect. The `NmSt` effect is intended for situations which do not trigger an error condition under any circumstances.

We summarise all of the datatypes and effects we have covered in this section in Figure 3.6.

Example 3.29 (Generating Fresh Metavariables)**.** The `freshMeta` operator uses the `fresh` command to generate a fresh positive integer and append a metavariable declaration onto the context:

```
freshMeta : {[NmSt]Int}
freshMeta! = let x = fresh! in
             modify {sx -> snoc sx (bmv x hole)}; x
```

The `modify` operator applies the provided suspended computation to the current value stored in the state cell. Note that by implicit effect polymorphism, if `freshMeta` is invoked in an environment with ability `[Contextual]`, the implicit effect variable will unify with the remaining effects. In contrast, the HASKELL implementation gives the coarser `[Contextual]Int` result type to `freshMeta`. We adopt similar simplifications throughout the implementation, giving the most precise types possible to operators.

3.13.2 The Type Inference Algorithm

Now we consider implementing the type inference algorithm itself. We discuss the definition one language construct at a time. The definition is presented in full in Figure 3.7.

For inferring the type of a variable, we simply have to lookup its type in the context:

```
infer (varx x) = specialise (findSch x)
```

where `findSch` accepts the de Bruijn index of a term variable and traverses the current context, returning its corresponding type scheme. If the type returned by `findSch` turns out to be monomorphic then `specialise` simply returns it. On the other hand, for a polytype, specialisation instantiates all quantified type variables turning them into fresh unknown metavariables.

Example 3.30 (In Search of a Scheme)**.** We briefly describe the implementation of `findSch` because it provides another illustration of fine-grained effect specifications and also demonstrates some error cases. The definition of `findSch` simply delegates the search to another operator which accepts the current context as its second argument:

```
findSch : {Int -> [Abort,CtxSt,TypeError]Sch}
findSch x = help x get!
```

where `help` is defined by the following cases:

```
-- Calculus
data Ty = meta Int | var Int | arr Ty Ty

data Sch = mono Ty | poly Sch

data Tm = varx Int | lam Tm | app Tm Tm | seq Tm Tm

-- Contexts
data Entry = bmv Int (Decl Ty) | btv Sch | mrk

data Decl X = hole | defn X

data Bwd X = emp | snoc (Bwd X) X

type Ctx = Bwd Entry

type Suffix = List (Pair Int (Decl Ty))

-- Computational Effects
interface CtxSt = [State Ctx]

interface Names = fresh : Int

interface TypeError = raise X : String -> X

interface NmSt = [CtxSt,Names]

interface Contextual = [Abort,NmSt,TypeError]
```

Figure 3.6: The complete collection of types and effects used by the algorithm.

```
help : {Int -> Bwd Entry -> [Abort,TypeError]Sch}
help    x       bemp     = raise "Out of scope"
help    0 (bcons ctx (btv t)) = t
help    x (bcons ctx (btv t)) =
  if (x > 0) {help (x-1) ctx} {abort!}
help    x  (bcons ctx _) = help x ctx
```

Notice how `help` does not need its ability to include `CtxSt`, specifying only the error effects it may produce. A type error is raised if the context does not contain any such variable; a programmer error. An `abort` is raised if the number given to `help` is less than zero. Note that if `help` is called with $x \geq 0$, then this condition will not be reached. It is only reached if some other part of the implementation, e.g. `findSch`, calls `help` with $x < 0$; hence, a bug in the implementation.

Note, we could have encapsulated the collection of effects in `findSch` or `help` as an alias. For example,

```
interface Errors = [Abort,TypeError]
```

Doing so is a trade-off between conciseness and transparency. Creating too many aliases may make it more difficult to determine which commands are actually permitted.

For the lambda abstraction case, we generate a fresh metavariable for the type of the argument, and infer the type of the body in a context extended with the argument:

```
infer (lam t) = let a = meta freshMeta! in
                let b = extend (mono a) {infer t} in
                arr a b
```

The `extend` operator temporarily extends the context with the specified term variable and runs the provided suspended computation in this extended context.

Application is interesting because it invokes the unification procedure. First, we infer the type of the function and the argument. Then we generate a fresh metavariable to stand for the result type of the function:

```
infer (app f s) = let a = infer f in
                  let b = infer s in
                  let c = meta freshMeta! in
                  unify a (arr b c); c
```

```
infer : {Tm -> [Contextual]Ty}
infer (varx x) = specialise (findSch x)
infer (lam t) = let a = meta freshMeta! in
                let b = extend (mono a) {infer t} in
                arr a b
infer (app f s) = let a = infer f in
                  let b = infer s in
                  let c = meta freshMeta! in
                  unify a (arr b c); c
infer (seq s t) = let a = generaliseOver {infer s} in
                  extend a {infer t}
```

Figure 3.7: Type inference for HM calculus

We require `f` to have function type where its domain matches the type of `s`. So we pass this problem to the unifier to solve and return the result type of the function, `c`. The implementation of unification exactly mirrors the implementation given by Gundry [2013] so we omit it.

Lastly, polymorphic **let** expressions infer a type for the bound term and then generalise it, quantifying over all the local metavariables. The final step is to infer the type of the **let** body in an extended context:

```
infer (seq s t) = let a = generaliseOver {infer s} in
                  extend a {infer t}
```

Generalisation is defined such that we never generalise variables which escape the scope of `s`. Adding a marker to the context before executing the provided computation suffices to ensure this (Figure 3.8). The `skimCtx` operator constructs a suffix of metavariables from the context by repeatedly popping the right-most entry until the marker is reached. The up operator generalises a suffix of metavariables over a type to produce a type scheme.

3.14 Discussion and Related Work

There is a rapidly growing literature on algebraic effects and handlers [Yallop et al., 2019]. Some work pertains to the theoretical foundations, whilst others are more fo-

```
generaliseOver : {{[CtxSt,TypeError]Ty} -> [CtxSt,TypeError]Sch}
generaliseOver m = modify {sx -> snoc sx mrk};
                   let a = m! in
                   let suffix = skimCtx [] in
                   up suffix a

skimCtx : {Suffix -> [CtxSt,TypeError]Suffix}
skimCtx xs = case popL!
             {(bmv n d) -> skimCtx ((pair n d) :: xs)
             | mrk -> xs}
```

Figure 3.8: Generalisation of type variables.

cussed on the use of effect operations and their handlers as a programming abstraction. In this section we focus on the body of work devoted to programming with effects and handlers. In particular, we emphasise languages with first-class support for effects and handlers. During our discussion, we also highlight FRANK's connection to the broader literature on type systems, suggesting some areas worthy of further investigation.

Frank. A number of extensions to FRANK were introduced by Convent [2017]. Many of these are also described in the JFP article [Convent et al., 2020]. In this chapter, we covered Convent et al.'s introduction of polymorphic commands and interface aliases. We omit discussion of the built-in effect for ML-style references which leverages the polymorphic commands feature. Convent described a direct small-step operational semantics for the FRANK surface language. This semantics was later extended by Convent et al. [2020] to incorporate adaptors. Adaptors were added to provide abstraction and encapsulation of effects; FRANK's proposed solution to the effect pollution problem (see below for alternative solutions). Adaptors can appear in terms, as presented in this chapter, and also within adjustments where they perform a remapping of the ambient ability supplied to an argument computation. The latter case is known as an *adaptor adjustment*. Convent et al. [2020] present an extended concurrency example in FRANK which relies heavily on adaptors.

As mentioned, we have implemented a prototype of FRANK [Convent et al., 2019] using HASKELL. In order to rapidly build a prototype, we consciously decided to take advantage of a number of existing technologies. The backend of FRANK is imple-

mented by first translating to SHONKY [McBride, 2016], which amounts to an untyped version of Frank. The SHONKY implementation executes code directly through an abstract machine. We have modified SHONKY slightly to support features like adaptors and ML-style references. As discussed in Section 3.13, the FRANK typechecking algorithm is inspired by the "type-inference-in-context" technique of Gundry et al. [2010]; the technique has been shown to scale to the dependently typed setting [Gundry, 2013].

Although we do not explicitly spell out the details in this chapter, FRANK is indentation sensitive. In order to implement indentation sensitivity, Adams and Ağacan [2014] introduce an extension to parsing frameworks based on parsing expression grammars. Such grammars provide a formal basis for the Parsec [Leijen and Martini, 2015] and Trifecta [Kmett, 2015] parser combinator HASKELL libraries. In contrast to the ad-hoc methods typically employed by many indentation sensitive languages (including HASKELL and IDRIS [Brady, 2013]), Adams and Ağacan's extension has a formal semantics. FRANK's parser is written using Trifecta with the indentation sensitive extension, which greatly simplifies the handling of indentation by separating it as much as possible from the parsing process.

Eff. Bauer and Pretnar [2015] introduced the EFF programming language, ML-like language providing first-class support for effects and handlers. The full language is based on a fine-grained call-by-value calculus, although the concrete syntax hides this from the programmer. Unlike FRANK, EFF has deep handlers and distinguished constructs for defining and invoking handler. Their EFF calculus was given a denotational semantics yielding a sound induction principle for reasoning about effectful computations [Bauer and Pretnar, 2014]. A distinct early feature of full EFF in contrast to most other languages in this space is the support for *dynamic* creation of instances; which were assumed to be statically known for the purposes of their metatheory. In later revisions to the language, instances were removed in favour of a simpler system.

Other languages. Leijen's [2014] KOKA has also been extended to provide first-class support for effects and handlers. The implementation leverages KOKA's existing row-polymorphic effect system. KOKA's effect system also allows duplicate labels in effect rows.

Hillerström and Lindley [2016] describe extending LINKS with algebraic effects and handlers leveraging its existing row-polymorphic effect system. Unlike FRANK and KOKA, effect rows in LINKS cannot have duplicates. However, their system does

support *presence* polymorphism allowing the *absence* of an effect to be recorded. An absence annotation is akin to the mask adaptor adjustment described by Convent et al. [2020]. Hillerström and Lindley formalise their extension as a core calculus reminiscent of the intermediate form used within the LINKS compiler, proving a correspondence result between its small-step semantics and an abstract machine semantics. Their abstract machine semantics is similar to SHONKY.

Biernacki et al.'s [2019b] HELIUM is a functional language supporting effects and handlers, and an ML-style module system providing two abstraction mechanisms for effects: *local* effects and *existential* effects. Existential effects appear as part of a module's interface in a similar vein to existential types, ensuring clients can never handle them locally and must use built-in handler(s) provided by the module interface. Local effects restrict the scope of their label and operations to a single expression, *e.g.* the body of a function. They provide an alternative solution to the problem identified by Biernacki et al. [2018] and described in Appendix A. Both mechanisms utilise *effect coercions*, a generalisation of *lift*, and similar in expressive power to FRANK's adaptors.

As an extended HELIUM example, Biernacki et al. present a unification algorithm using a union-find based data structure, a key component of implementations of algorithm $\mathcal{W}$ [Milner, 1978] for Hindley-Milner type inference. While, in our example (Section 3.13), effects were used simply to provide more fine-grained specifications, Biernacki et al.'s approach is closer in spirit to Atkey [2015] who decomposes the typechecking process into a series of commands. These commands are combined to build abstract tactics or "scripts". Depending on how the commands are interpreted the scripts may yield typecheckers or elaborators. In a similar vein, it would be interesting to explore whether we can express different type inference algorithms under a common effectful interface.

Multihandlers. Multihandlers are a distinctive feature of FRANK with respect to other languages and calculi for effects and handlers. However, there are a number of programming abstractions which bear similarities with them. *Joinads* are an expressive programming abstraction suitable for concurrent, parallel and event-driven programming [Petricek and Syme, 2011]. Using a computation type-specific pattern matching construct joinads can express join patterns [Fournet and Gonthier, 1996] for communicating processes, futures for parallelism, and event synchronisation. Petricek and Syme [2011] define joinads as a syntactic extension to the *F#* language. That is,

the match construct is not a built-in feature of the language but is translated to a series of function calls. In contrast, FRANK multihandlers are given a direct semantics. The patterns for FRANK multihandlers are slightly more expressive, being able to match on partially complete computations via request patterns. Joinads restrict their pattern matching to binding on the final result only. As a consequence, joinads admit reordering of computation arguments whereas FRANK multihandlers are sensitive to argument order (see Section 7.2).

Effect Pollution. As described in Section 3.11, Biernacki et al. [2018] introduce an operation called *lift* which extends an effect row with an additional effect instance. Leijen [2014] describes a similar operation which he calls *inject*.

Zhang and Myers [2019] present an alternative approach to ensure handlers encapsulate effects. Their key idea is to extend types with *handler variables* and handler polymorphism. The variables act as labels on effects in the ambient context and determine which handlers interpret which effects. Handler polymorphism is resolved by substituting the nearest lexically enclosing handler for the handler variable. A capability region system ensures that computations do not escape their handlers. Following Biernacki et al. [2018], they develop a sound logical relations model and prove their system satisfies an abstraction theorem. While Zhang and Myers do not present an implementation, they claim that the programmer need not deal with handler variables in practice. Rather, they outline a desugaring and rewriting pass which inserts the requisite variable bindings and handler instantiations.

Biernacki et al. [2019b] introduce effect coercions as a generalisation of *lift*. Like adaptors, coercions have computational content so persist at runtime. While they support lifting and swapping within an effect row, they do not support duplication. However, this is not a serious limitation since we can achieve effect duplication using a handler, *e.g.* we can obtain an equivalent definition to `conflate2` in Example 3.26 by wrapping `(distinct s)` in a call to the `catch` handler of Section 3.10, and re-raising the `abort` command in the alternative computation.

Session Types for Communicating Processes Example 3.20 demonstrated communication between two processes. However, we observed that there was a need to `abort` the interaction if the two processes did not agree on the how many messages were to be sent and received. That is, `pipe` does not establish an agreed communication *protocol* between the two processes. To achieve compliance and be capable of

expressing more sophisticated protocols, we could extend FRANK's type system with *session types* [Honda, 1993] which statically enforce communicating parties to adhere to an agreed sequence of interactions.

Introducing session types into the FRANK ecosystem is not without its challenges. The session type of a channel endpoint evolves as communication proceeds; its type is a dynamic resource. Given the FRANK programmer's ability to manipulate delimited continuations using handlers, resources must be carefully managed and deallocated when no longer required. Furthermore, in real-world distributed applications failures are inevitable, and must be accounted for in any practical type-based solution. How to integrate session types and FRANK's effect handlers to support distributed applications is an interesting area to investigate.

Fowler et al. [2019] provide a potential starting point, integrating asynchronous session types with exception handling. The combination permits explicit cancellation of a protocol when a failure mode is encountered. Fowler et al. provide a compelling distributed example application in the form of a web-based chat server.

Chapter 4

Triangulating Context Lemmas

This chapter introduces a distinctive proof method, *triangulation*, for proving context lemmas. We apply the method to a simply-typed fine-grained call-by-value λ-calculus. First, we use the method to show Milner's [1977] classical result holds for our calculus. Second, we demonstrate the extensibility of the technique by proving a CIU result à la Mason and Talcott [1991], formulated with frame stacks [Pitts and Stark, 1998; Pitts, 2005].

Throughout this chapter, our presentation is influenced by our formalisation of the results in the AGDA interactive theorem prover which leverages state-of-the-art technology for dealing with well-typed and well-scoped terms [Allais et al., 2017]. In particular, we show that, contrary to popularly held belief [Lassen, 1998a; Pitts, 2011], handling *concrete* contexts even in the formalised setting is straightforward. That said, we give pen-and-paper renderings of our results in this chapter. For full details, the reader may consult the complete source code of our AGDA formalisation, which is available online at:

`https://www.github.com/cmcl/triangulating-context-lemmas`

Furthermore, we provide a Rosetta Stone in Table 4.1 at the end of the chapter, mediating between informal concepts, LaTeX rendering, and their AGDA formalisation.

While we are not the first to undertake such a formalisation effort [Ambler and Crole, 1999], we believe our *concrete, first-order* representation to be more robust than previous attempts; we compare our development to previous formalisations in Section 4.4.

By formalising results for both applicative and CIU theorems, we hope to ease the extension of our development to a calculus supporting algebraic effects and handlers,

$$
\begin{array}{lll}
\text{(Terms)} & M,N ::= V \mid \textbf{if } V \textbf{ then } M \textbf{ else } N \mid V\,V \\
& \qquad\quad \mid \textbf{ let } x = M \textbf{ in } N \\
\text{(Values)} & V,U ::= x \mid \textbf{tt} \mid \textbf{ff} \mid n \mid () \mid \lambda x : A.M \\
\text{(Types)} & A,B ::= \textbf{int} \mid \textbf{bool} \mid \textbf{unit} \mid A \to B \\
\text{(Sugar)} & M\ V ::= \textbf{let } f = M \textbf{ in } f\,V \text{ (where } f \text{ is fresh)}
\end{array}
$$

Figure 4.1: $\lambda^{\to}_{FG}$ types and terms.

where fine-grained stack-based approaches seem a better fit than big-step semantics. Demonstrating extensibility of our approach is moreover in the spirit of other proposals for mechanised metatheory developments [Aydemir et al., 2005].

This chapter is based on the publication:
Craig McLaughlin, James McKinna, and Ian Stark. Triangulating context lemmas. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pp. 102–114, 2018. doi:10.1145/3167081.

4.1 A Fine-Grained Call-By-Value Calculus

In this chapter, the triangulation proof method is applied to a call-by-value calculus, $\lambda^{\to}_{FG}$, inspired by fine-grained call-by-value (FGCBV) [Levy, 2004] and a normal form for terms used by Pitts [2005]. Figure 4.1 gives an informal context-free grammar for the calculus, with named λ and **let** bindings. The distinguishing features are the separation between the grammars for values and terms (we systematically elide the explicit lifting of values into terms) and the restriction to one term constructor, **let**, to express sequencing. We introduce the syntactic sugar $M\ V$ (also in Figure 4.1) to express application of a term M to a value V. We consider a simple type discipline, where τ ranges over ground types ($\tau \in \{\textbf{bool}, \textbf{int}, \textbf{unit}\}$). For values, x ranges over an infinite set of variables, b ranges over $\{\textbf{tt}, \textbf{ff}\}$, and n over $\mathbb{Z}$. In the definitions and proofs that follow, = and $\triangleq$ denote equalities that hold definitionally, and $\equiv$ denotes equalities which hold propositionally.

4.1.1 Formalising $\lambda^{\to}_{FG}$

The results of this chapter have all been formalised in the AGDA [Norell, 2009] interactive theorem prover. While we give pen-and-paper renderings of our results, our

presentation is influenced by the first-order de Bruijn representation in a dependently-typed meta-language such as our ML_{UTT}. As mentioned in Chapter 2, we follow the discipline advocated by Allais et al. [2017], using their framework to represent the informal syntax of Figure 4.1 using inductive families [Dybjer, 1994], ensuring that our syntax is well-typed and well-scoped by construction. As demonstrated in Example 2.13, the data types describing variables, values and terms of the object language, $\lambda^{\rightarrow}_{FG}$, are indexed by an object-level type and context. Thus we express $\lambda^{\rightarrow}_{FG}$ directly via its typing rules, as in Figure 4.2. A consequence of our encoding is the occasional requirement to introduce a renaming, in particular the special case of *weakening*, to ensure the type-and-scope correctness of terms. For example, in the syntactic sugar for application, the argument is applied in an environment extended with the binding for the function:

$$\frac{\dfrac{\vdots}{\Gamma \vdash M : A \to B} \qquad \dfrac{\Gamma \vdash V : A}{\Gamma, A \to B \vdash \mathit{weak}(V) : A}}{\Gamma \vdash M\ V : B}$$

where *weak* takes Γ-terms to Γ, A-terms for any type A by shifting de Bruijn indices. From now on, for readability, we elide any explicit weakening in terms.

ACMM uses first-order abstract syntax for term representation. However, for operations and proofs defined over terms, ACMM employs weak HOAS [Despeyroux et al., 1995]. The framework defines a *semantics* as an instantiation of a generic Kripke-style traversal over the abstract syntax parametric in the interpretations $\mathcal{V}$, $\mathcal{MV}$, and $\mathcal{MT}$ for variables, values, and terms, respectively. Renaming, capture-avoiding substitution, *etc.* arise as instances of the generic traversal. Semantics are Kripke in the sense that interpretations for binders must hold under all possible environment extensions. For example, the rule for interpreting λ is as follows:

$$\frac{\Box\,(\mathcal{V}\,A\,\Delta \longrightarrow \mathcal{MT}\,B\,\Delta)}{\mathcal{MV}\,(A \Rightarrow B)\,\Gamma}$$

where the modality $\Box$ quantifies over all environment extensions $\Gamma \subseteq \Delta$ by renamings.

Definition 4.1 (Simultaneous Substitution)**.** For typing environments Γ and Δ, let $\theta : \Gamma \vDash \Delta$ denote a simultaneous substitution θ of variables in environment Γ by $\lambda^{\rightarrow}_{FG}$ values in environment Δ. Informally, θ takes Γ-terms to Δ-terms: given $\Gamma \vdash M : A$ we may apply the substitution to M and obtain $\Delta \vdash \theta(M) : A$. In the special case of $\theta : \Gamma, A \vDash \Gamma$ given at $\mathbf{var}_0$ by value $\Gamma \vdash V : A$ and everywhere else the identity we write

$$\begin{array}{ll} \text{(Environments)} & \Gamma,\Delta ::= \cdot \mid \Gamma, A \\ \text{(Indices)} & k ::= 0 \mid \mathbf{succ}\, k \end{array}$$

$$\textsc{zero}\ \frac{}{0 : (A \in \Gamma, A)} \qquad \textsc{succ}\ \frac{k : (B \in \Gamma)}{\mathbf{succ}\, k : (B \in \Gamma, A)} \qquad \textsc{var}\ \frac{k : (A \in \Gamma)}{\Gamma \vdash \mathbf{var}_k : A}$$

$$\textsc{boolean}\ \frac{}{\Gamma \vdash b : \mathbf{bool}}\ (b = \mathbf{tt}, \mathbf{ff}) \qquad \textsc{int}\ \frac{}{\Gamma \vdash n : \mathbf{int}}\ (n \in \mathbb{Z}) \qquad \textsc{unit}\ \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

$$\textsc{fun}\ \frac{\Gamma, A \vdash M : B}{\Gamma \vdash \lambda_A M : A \to B} \qquad \textsc{ifThenElse}\ \frac{\Gamma \vdash V : \mathbf{bool} \qquad \Gamma \vdash N_{\mathbf{tt}} : A \qquad \Gamma \vdash N_{\mathbf{ff}} : A}{\Gamma \vdash \mathbf{if}\ V\ \mathbf{then}\ N_{\mathbf{tt}}\ \mathbf{else}\ N_{\mathbf{ff}} : A}$$

$$\textsc{app}\ \frac{\Gamma \vdash V : A \to B \qquad \Gamma \vdash U : A}{\Gamma \vdash V\ U : B} \qquad \textsc{let}\ \frac{\Gamma \vdash M : A \qquad \Gamma, A \vdash N : B}{\Gamma \vdash \mathbf{let}_A\ M\ N : B}$$

Figure 4.2: $\lambda^{\rightarrow}_{FG}$ typing rules

$$\textsc{ifThenElse}\ \frac{}{\mathbf{if}\ b\ \mathbf{then}\ N_{\mathbf{tt}}\ \mathbf{else}\ N_{\mathbf{ff}} \rightsquigarrow N_b} \qquad \textsc{appBeta}\ \frac{}{(\lambda_A M)\ V \rightsquigarrow M[V]}$$

Figure 4.3: $\lambda^{\rightarrow}_{FG}$ primitive reduction semantics

$\theta(M)$ as $M[V]$ in the usual way. For the special case of $\theta : \Gamma \vDash \cdot$ we say that θ is a Γ-*closing* substitution. If $\theta : \Gamma \vDash \Delta$ and $\theta' : \Delta \vDash \Theta$ then $\theta' \circ \theta : \Gamma \vDash \Theta$ denotes the composition of the simultaneous substitutions.

Definition 4.2. Given $\theta : \Gamma \vDash \Delta$, define θ_k to be the value $\Delta \vdash V : A$ specified for the variable $\Gamma \vdash \mathbf{var}_k : A$. Then, θ_0 denotes the value specified for $\mathbf{var}_0$. For $\theta : \Gamma, A \vDash \Delta$, define the simultaneous substitution $(\mathbf{succ}\,\theta) : \Gamma \vDash \Delta$ by its behaviour on variables, $\Gamma \vdash \mathbf{var}_k : A$, for all indices k:

$$(\mathbf{succ}\,\theta)_k = \theta_{(\mathbf{succ}\, k)}\,.$$

We define a small-step operational semantics for $\lambda^{\rightarrow}_{FG}$ via a primitive reduction on closed terms, in Figure 4.3, which is then contained within the general reductions of

$$\textsc{primRed}\ \frac{M \rightsquigarrow M'}{M \longrightarrow M'} \qquad \textsc{letValue}\ \frac{}{\mathbf{let}_A\ V\ M \longrightarrow M[V]} \qquad \textsc{letRed}\ \frac{M \longrightarrow M'}{\mathbf{let}_A\ M\ N \longrightarrow \mathbf{let}_A\ M'\ N}$$

Figure 4.4: $\lambda^{\rightarrow}_{FG}$ small-step operational semantics

$$\textsc{value}\ \frac{\cdot \vdash V : A}{V \Downarrow_A V} \qquad \textsc{if}\ \frac{N_b \Downarrow_A V}{\mathbf{if}\ b\ \mathbf{then}\ N_{\mathbf{tt}}\ \mathbf{else}\ N_{\mathbf{ff}} \Downarrow_A V}$$

$$\textsc{app}\ \frac{M[V] \Downarrow_B U}{(\lambda_A M)\ V \Downarrow_B U} \qquad \textsc{let}\ \frac{M \Downarrow_A V \qquad N[V] \Downarrow_B U}{\mathbf{let}_A\ M\ N \Downarrow_B U}$$

Figure 4.5: $\lambda^{\rightarrow}_{FG}$ big-step operational semantics

Figure 4.4. The separation of these two relations is inspired by Pitts [2005] and is used when we define frame stack evaluation in Section 4.3. Both relations are type-correct by construction; the proof of type preservation is immediate.

Figure 4.5 presents $\lambda^{\rightarrow}_{FG}$ evaluation semantics as a type-indexed relation between closed terms and values. We omit typing annotations where unambiguous. Evaluation is closed under the reduction relations: the proof is straightforward by induction over derivations.

Lemma 4.3 (Reduction Respects Evaluation)**.**
If $M \rightsquigarrow M'$ or $M \longrightarrow M'$ then $M \Downarrow V$ if and only if $M' \Downarrow V$.

4.1.2 Formalising Contexts

As a step towards formalising observational approximation, we treat contexts as an extension of $\lambda^{\rightarrow}_{FG}$ syntax; and thanks to ACMM the grammar of these concrete contexts is directly expressible as an inductive definition. A *term context* $\mathcal{C}$ is a possibly-open term of arbitrary type containing zero or more occurrences of a well-typed and well-scoped hole. Analogously, a *value context* $\mathcal{V}$ is a possibly-open value of arbitrary type containing zero or more occurrences of such a hole. Naturally enough our formalisation enforces well-typed and well-scoped contexts, by construction, in the same way it

$$\textsc{vccHole}\ \frac{}{\Gamma \vdash \langle\langle - \rangle\rangle : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} A} \qquad \textsc{vccTrm}\ \frac{\Delta \vdash M : B}{\Delta \vdash \mathbf{trm}\ M : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B}$$

$$\textsc{vccFun}\ \frac{\Delta, B \vdash \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} C}{\Delta \vdash \lambda_B\, \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} (B \to C)}$$

$$\textsc{vccIfThenElse}\ \frac{\Delta \vdash \mathcal{V} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} \mathbf{bool} \qquad \Delta \vdash \mathcal{C}_{\mathbf{tt}} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B \qquad \Delta \vdash \mathcal{C}_{\mathbf{ff}} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B}{\Delta \vdash \mathbf{if}\ \mathcal{V}\ \mathbf{then}\ \mathcal{C}_{\mathbf{tt}}\ \mathbf{else}\ \mathcal{C}_{\mathbf{ff}} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B}$$

$$\textsc{vccApp}\ \frac{\Delta \vdash \mathcal{V} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} (B \to C) \qquad \Delta \vdash \mathcal{W} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B}{\Delta \vdash \mathcal{V}\ \mathcal{W} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} C}$$

$$\textsc{vccLet}\ \frac{\Delta \vdash \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B \qquad \Delta, B \vdash \mathcal{E} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} C}{\Delta \vdash \mathbf{let}_B\ \mathcal{D}\ \mathcal{E} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} C}$$

Figure 4.6: $\lambda^{\rightarrow}_{FG}$ typing rules for variable-capturing contexts

does for terms and values.

The usual notion of a *variable-capturing* context (VCC) allows a context to capture variables occurring free in the term that fills the hole. In Figure 4.6 we define VCCs for $\lambda^{\rightarrow}_{FG}$, with $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$ ranging over term contexts and $\mathcal{V}$, $\mathcal{W}$ over value contexts. The form of a well-typed and well-scoped context $\mathcal{C}$ is $\Delta \vdash \mathcal{C} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B$, which says that term context $\mathcal{C}$ has type B in environment Δ and contains zero or more occurrences of a hole of type A in an environment Γ. For such a well-typed term context $\mathcal{C}$, there is an operation, *instantiation*, denoted by $\mathcal{C}[M]^{\mathrm{VCC}}$ which fills the holes in the term context $\mathcal{C}$ with M. Instantiation is (yet) another structural traversal, where the only case of interest is that of the hole constructor, replacing the hole with M:

$$\langle\langle - \rangle\rangle[M]^{\mathrm{VCC}} = M\,.$$

By virtue of our de Bruijn encoding, we avoid traditional concerns (as in [Lassen, 1998a; Pitts, 2011, for example]) associated with α-equivalence of VCCs. However,

VCCs are still a little inconvenient to work with because they force holes to match exactly the enclosing scope of their context, even up to the ordering of the variables captured (see the VCCHOLE rule in Figure 4.6).

To manage such concerns we employ a different notion of *value-substituting* contexts (VSCs), where each occurrence of a hole carries an appropriate (well-typed) substitution, as in rule VSCHOLE below. This ensures that the instantiation operation $\mathcal{C}[M]^{\mathrm{VSC}}$ is itself well-typed and well-scoped, and again definable by a simple traversal. Typing judgements for VSCs are the same as for VCCs except we replace the VCCHOLE rule with the following.

$$\frac{\theta : \Gamma \vDash \Delta}{\Delta \vdash \langle\langle \theta - \rangle\rangle : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VSC}} A} \text{ VSCHOLE}$$

Instantiation for VSCs is as with VCCs except for the base case, where a hole is filled by applying the substitution provided.

$$\langle\langle \theta - \rangle\rangle[M]^{\mathrm{VSC}} = \theta(M)$$

Every VCC is trivially a VSC, by annotating each hole with an identity substitution. For the rest of the paper we use unadorned brackets ('[]') for instantiation when it is clear from the context which version of contexts is being used.

Lassen [1998b] defines a notion of variable-capturing context that is slightly different to ours, equivalent to each hole carrying a *renaming* rather than a general value substitution. This lies strictly between our VCCs and VSCs, as any renaming is naturally a substitution. Later, we shall see that relations based on either VCCs or VSCs coincide, and so also with ones based on Lassen's variant.

4.1.3 Observational Approximation

This section formally defines the notion of observational approximation, alluded to in the introduction, for $\lambda^{\rightarrow}_{FG}$. Its definition can be traced back to Morris [1968] and builds on the account of term contexts and substitutions in the previous section.

We begin with a relation transformer lifting any relation on values to a relation on terms.

Definition 4.4. For closed terms M, N of type A and $\mathcal{R}$ a binary relation on closed values of type A,

$$M \downarrow\mathcal{R}\downarrow^{T} N \triangleq \forall V.\, M \Downarrow_A V \implies \exists U.\, N \Downarrow_A U \wedge V\ \mathcal{R}\ U$$

This relation transformer is closed under primitive reduction and expansion. We state and sketch a proof for left-closure under $\rightsquigarrow$.

Lemma 4.5 ($\lfloor - \rfloor^T$ left-closed under $\rightsquigarrow$)**.** *For all closed terms M, N of type A and $\mathcal{R}$ a binary relation on closed values of type A, the following properties hold.*

1. *If $M \rightsquigarrow P$ and $M \lfloor \mathcal{R} \rfloor^T N$ then $P \lfloor \mathcal{R} \rfloor^T N$*
2. *If $M \rightsquigarrow P$ and $P \lfloor \mathcal{R} \rfloor^T N$ then $M \lfloor \mathcal{R} \rfloor^T N$*

Proof. By the definition of $\lfloor - \rfloor^T$ and Lemma 4.3. □

We next define a basic relation that drives all our other notions of approximation. This aims to capture what values can be directly distinguished, without further context or evaluation.

Definition 4.6 (Ground equivalence)**.** Let $\approx_A$ be the equivalence relation defined on closed values of type A by:

- $V \approx_\tau U \triangleq V = U$, for all $V, U : \tau$,
- $\lambda x : A.M \approx_{A \to B} \lambda x : A.N$ for all M, N

Although this particular relation is symmetric, the definitions we build on it are biased to give approximation preorders. One reason to factor this out in our development is the potential for incorporating non-trivial equivalences on ground values (such as for adding primitive operations and their definitions) in a similar fashion to Johann et al. [2010] (see Section 4.4 for more on this). In languages where the only *primitive* effect is nontermination, *e.g.* ELLA in Chapter 6, ground equivalence degenerates to the total relation.

Now we are in a position to define observational approximation in terms of ground equivalence. First, we have a definition of approximation parametric in the notion of contexts we consider (VSCs, VCCs,...). Let $\mathcal{K}$ range over these notions of contexts. A *program* is a closed term of arbitrary type. A *program $\mathcal{K}$-context* is a closed term $\mathcal{K}$-context ranged over by $\mathcal{P}, \mathcal{Q}$, i.e. $\cdot \vdash \mathcal{P} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathcal{K}} B$; where for convenience we usually omit the empty environment.

Definition 4.7 ($\mathcal{K}$ Approximation)**.** If M, N are terms of type A in context Γ then

$$\Gamma \vdash M \lesssim_A^{\mathcal{K}} N$$

asserts that for all program contexts $\mathcal{P} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathcal{K}} B$,

$$\mathcal{P}[M]^{\mathcal{K}} \lfloor \approx_B \rfloor^T \mathcal{P}[N]^{\mathcal{K}}.$$

From all $\mathcal{K}$-contextual approximations, we choose as primary the one with the largest class of contexts — thereby able to make the finest discrimination between terms.

Definition 4.8 (Observational Approximation)**.** Let observational approximation, written $\lesssim^{\mathrm{VSC}}$, be Definition 4.7 instantiated with $\mathcal{K} = \mathrm{VSC}$.

Now we can establish our first inclusion of approximations.

Lemma 4.9. *If* $\Gamma \vdash M \lesssim_A^{VSC} N$ *then* $\Gamma \vdash M \lesssim_A^{VCC} N$.

Proof. Assume given terms $\Gamma \vdash M, N : A$ with $\Gamma \vdash M \lesssim_A^{\mathrm{VSC}} N$ and VCC $\mathcal{P} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{VCC}} B$. From $\mathcal{P}$ we can build matching VSC $\mathcal{Q}$ where each hole carries the identity substitution. The result follows by using our VSC approximation with M and N, and the property that for all terms $\Gamma \vdash M' : A$, $\mathcal{P}\langle\langle M' \rangle\rangle^{\mathrm{VCC}} \equiv \mathcal{Q}\langle\langle M' \rangle\rangle^{\mathrm{VSC}}$ (proved by induction on the structure of $\mathcal{P}$). □

Definition 4.7 can readily be extended to an equivalence, denoted by $\simeq^{\mathcal{K}}$:

$$\Gamma \vdash M \simeq_A^{\mathcal{K}} N \;\triangleq\; \Gamma \vdash M \lesssim_A^{\mathcal{K}} N \wedge \Gamma \vdash N \lesssim_A^{\mathcal{K}} M\,.$$

Throughout this dissertation, results will be stated for the approximations only but can be straightforwardly lifted to equivalences [Stark, 1994; Pitts and Stark, 1998].

4.2 Big Steps for a Context Lemma

In this section we define three approximation relations and prove a triangle of implications which together lead to a context lemma inspired by Milner's [1977] original result for combinatory logic.

As noted in Chapter 1, proving observational approximation directly can be arduous owing to the requirement to consider all possible program contexts. Happily, Milner [1977] showed that for typed combinatory logic it is enough to consider only contexts where the hole occurs in function position applied to some arguments. The following definition captures such contexts in our setting.

Definition 4.10 (Applicative Substituting Contexts)**.** The *applicative substituting contexts* (ASCs) are the restricted class of contexts defined by the following inference rules:

$$\textsc{ascHole}\ \frac{\theta : \Gamma \vDash \Delta}{\Delta \vdash \langle\langle \theta - \rangle\rangle : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{ASC}} A} \qquad \textsc{ascApp}\ \frac{\Delta \vdash \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{ASC}} (B \to C) \qquad \Delta \vdash V : B}{\Delta \vdash \mathcal{D}\, V : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{ASC}} C}$$

$$\textsc{Seq-Refl}\ \frac{}{M \rightsquigarrow^S M} \qquad \textsc{Seq-V}\ \frac{M \rightsquigarrow^S \mathbf{let}_A\ V\ N}{M \rightsquigarrow^S N[V/x]} \qquad \textsc{Seq-Step}\ \frac{M \rightsquigarrow^S M'}{\mathbf{let}_A\ M\ N \rightsquigarrow^S \mathbf{let}_A\ M'\ N}$$

Figure 4.7: $\lambda^{\rightarrow}_{FG}$ iterated reduction with respect to sequencing

Definition 4.11. Let *ASC approximation* be Definition 4.7 with $\mathcal{K} = \text{ASC}$.

Having introduced the required notation we may now formally state the context lemma we aim to prove in this section. Firstly, in our setting, Milner's original context lemma states that for all open $\lambda^{\rightarrow}_{FG}$ terms $\Gamma \vdash M, N : A$,

$$\Gamma \vdash M \lesssim^{\text{ASC}}_A N \iff \Gamma \vdash M \lesssim^{\text{VCC}}_A N.$$

However, we have taken our notion of VSCs as primary, so our context lemma becomes

$$\Gamma \vdash M \lesssim^{\text{ASC}}_A N \iff \Gamma \vdash M \lesssim^{\text{VSC}}_A N.$$

Before proving the above result we define some auxiliary notions.

Definition 4.12 (Let-Redexes from Substitution)**.** Let θ be a Δ-closing substitution. For $\Delta \vdash \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\text{VCC}} B$, define $\mathcal{D}^\theta$ by induction on the size of Δ:

$$\mathcal{D}^\theta = \begin{cases} \mathcal{D} & \Delta = \cdot \\ (\mathbf{let}_C\ \theta_0\ \mathcal{D})^{(\mathbf{succ}\,\theta)} & \Delta = \Delta', C \end{cases}$$

$\mathcal{D}^\theta$ represents a new (closed) VCC obtained from $\mathcal{D}$ by constructing a sequence of let-redexes on the outside of $\mathcal{D}$ from the substitution θ. For $\Delta \vdash M : A$, let M^θ denote the analogous operation for constructing a closed term from M and θ.

Figure 4.7 defines iterated reduction with respect to term sequencing. The relation satisfies analogous properties to Lemmas 4.3 and 4.5. Then we have the obvious result:

Lemma 4.13. *For $\Delta \vdash M : A$, and Δ-closing substitution θ, then we have*

$$M^\theta \rightsquigarrow^S \theta(M)$$

Definition 4.14. Define the operation $\star : \mathit{ASC} \to \mathit{VCC}$, which transforms a closed ASC into a closed VCC, by induction on the structure of the ASC:

$$\star(\langle\langle \theta - \rangle\rangle) = \langle\langle - \rangle\rangle^\theta$$

$$\star(\mathcal{P}\ V) = \star(\mathcal{P})\ V$$

where the ASCAPP case uses our syntactic sugar, extended to $\mathcal{K}$ contexts.

Lemma 4.15. *For all* $\Gamma \vdash M : A$, $\Delta \vdash \mathcal{C} : \langle\langle \Gamma \vdash A \rangle\rangle^{VCC} B$, *and* Δ*-closing substitutions,* θ*, we have*

$$(\mathcal{C}^{\theta})[M] \equiv (\mathcal{C}[M])^{\theta}$$

Proof. By induction on the size of the environment Δ. □

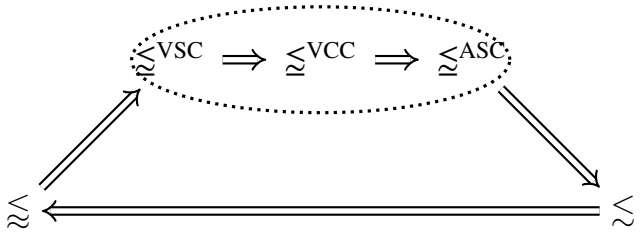
Lemma 4.16. *If* $\Gamma \vdash M \lesssim\!\!\gtrsim_A^{VCC} N$ *then* $\Gamma \vdash M \lesssim\!\!\gtrsim_A^{ASC} N$.

Proof. Assume given terms $\Gamma \vdash M, N : A$, $\Gamma \vdash M \lesssim\!\!\gtrsim_A^{\mathrm{VCC}} N$ and ASC $\mathcal{P} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{ASC}} B$. The proof proceeds by induction on the structure of $\mathcal{P}$.

If $\mathcal{P} = \langle\langle \theta - \rangle\rangle$, for some Γ-closing substitution θ, then the result follows from Lemmas 4.15 and 4.13, using $\lfloor - \rfloor^T$'s left- and right-closure under $\rightsquigarrow^S$.

Otherwise, $\mathcal{P} = Q\ V$, for some $Q : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{ASC}}(B \to C)$ and $V : B$. The result follows by applying the approximation assumption to $\star(\mathcal{P})$ (Definition 4.14) and using $\lfloor - \rfloor^T$'s left- and right-closure under $\rightsquigarrow^S$. □

Lemmas 4.9 and 4.16 give one direction of our context lemma for $\lambda_{FG}^{\rightarrow}$. Not surprisingly, the other direction is more challenging. This is where we adopt the triangulation proof method described earlier. That is, we concern ourselves with establishing the following cycle of implications.



The contextual approximations for VSCs, VCCs and ASCs form the apex of the triangle, while $\lesssim$ is *applicative* approximation and $\lesssim\!\!\gtrsim$ is *logical* approximation, both to be defined shortly. We have already shown the implications within the dotted ellipse (Lemmas 4.9 and 4.16), and the next two sections will address those forming the triangle.

4.2.1 Applicative Approximation

The definition of applicative approximation roughly follows Stark [1994]. It is a characterisation of approximation which is inductive on the type structure, and otherwise analogous to Gordon's [1994] *coinductive* typed applicative similarity.

Definition 4.17 (Applicative Approximation)**.** We define applicative approximation $\lesssim$ using two mutually recursive definitions, one for values and one for terms.
Define the relation $V_1 \lesssim_A^{val} V_2$ for closed values V_1, V_2 inductively on the structure of type A:

$$\frac{V_1 \cong_\tau V_2}{V_1 \lesssim_\tau^{val} V_2}\ \textsc{AppGndApx} \qquad \frac{\forall\, V : A.\ M_1[V] \lesssim_B^{trm} M_2[V]}{\lambda_A M_1 \lesssim_{A\to B}^{val} \lambda_A M_2}\ \textsc{AppAbsApx}$$

Define the relation $M_1 \lesssim_A^{trm} M_2$ for closed terms M_1, M_2 of type A by the following definition:

$$M_1 \lesssim_A^{trm} M_2 \triangleq M_1 \downarrow \lesssim_A^{val} \downarrow^T M_2$$

Just like for observational approximation, applicative approximation can be stated for open terms. We employ simultaneous substitutions to close over the environment.

Definition 4.18 (Open Applicative Approximation)**.** Given $\Gamma \vdash M_1 : A$ and $\Gamma \vdash M_2 : A$, we say that M_1 *applicatively approximates* M_2, written $\Gamma \vdash M_1 \lesssim_A M_2$, if and only if for all Γ-closing substitutions θ we have $\theta(M_1) \lesssim_A \theta(M_2)$.

A relationship between ASC approximation and applicative approximation can be established from which we obtain one side of the triangle.

Theorem 4.19. $\Gamma \vdash M \lesssim_A^{ASC} N \implies \Gamma \vdash M \lesssim_A^{trm} N$

Proof. By induction on the type A. □

Lemma 4.20. *Observational approximation implies applicative approximation:*

$$\Gamma \vdash M \lesssim_A^{VSC} N \implies \Gamma \vdash M \lesssim_A^{trm} N$$

Proof. By Lemmas 4.9 and 4.16 with Theorem 4.19. □

4.2.2 Logical Approximation

Logical approximation differs with respect to applicative approximation only in its handling of functions.

$$\textsc{LogApxIfThenElse}\ \frac{B \lesssim^{val}_{bool} B' \qquad L \lesssim^{trm}_{A} L' \qquad R \lesssim^{trm}_{A} R'}{\textbf{if } B \textbf{ then } L \textbf{ else } R \lesssim^{trm}_{A} \textbf{if } B' \textbf{ then } L' \textbf{ else } R'} \qquad \textsc{LogApxApp}\ \frac{F \lesssim^{val}_{A\to B} G \qquad U \lesssim^{val}_{A} V}{F\ U \lesssim^{trm}_{B} G\ V}$$

$$\textsc{LogApxLet}\ \frac{M \lesssim^{trm}_{A} M' \qquad A \vdash N \lesssim^{trm}_{B} N'}{\textbf{let}_A\ M\ N \lesssim^{trm}_{B} \textbf{let}_A\ M'\ N'}$$

Figure 4.8: $\lesssim$ is closed under the compound term formers

Definition 4.21 (Logical Approximation)**.** Define logical approximation by the mutually recursive relations $\lesssim^{val}$ and $\lesssim^{trm}$. The relation $V_1 \lesssim^{val}_{A} V_2$ for closed values V_1, V_2 is defined recursively on the structure of type A:

$$\textsc{LogGndApx}\ \frac{V_1 \cong_\tau V_2}{V_1 \lesssim^{val}_{\tau} V_2} \qquad \textsc{LogAbsApx}\ \frac{\forall\, V_1, V_2.\ V_1 \lesssim^{val}_{A} V_2 \implies M_1[V_1] \lesssim^{trm}_{B} M_2[V_2]}{\lambda_A M_1 \lesssim^{val}_{A\to B} \lambda_A M_2}$$

The relation $M_1 \lesssim^{trm}_{A} M_2$ for closed terms M_1, M_2 of type A is defined using $\downarrow - \downarrow^T$ and $\lesssim^{val}$:

$$M_1 \lesssim^{trm}_{A} M_2 \triangleq M_1 \downarrow \lesssim^{val}_{A} \downarrow^T M_2$$

To extend logical approximation to open terms we define the notion of logical approximation of substitutions.

Definition 4.22 (Logical Approximation of Substitutions)**.** For Γ-closing substitutions, θ, θ', $\theta \lesssim_\Gamma \theta'$ holds if and only if, for all $k : (A \in \Gamma)$, $\theta_k \lesssim_A \theta'_k$. For $\theta, \theta' : \Gamma \vDash \Delta$, $\Delta \vdash \theta \lesssim_\Gamma \theta'$ holds if and only if, for all Δ-closing substitutions, θ_1, θ_2 such that $\theta_1 \lesssim_\Delta \theta_2$ holds then $\theta_1 \circ \theta \lesssim_\Gamma \theta_2 \circ \theta'$ holds.

Definition 4.23 (Open Logical Approximation)**.** Assume given $\Gamma \vdash M_1 : A$ and $\Gamma \vdash M_2 : A$, then M_1 logically approximates M_2, written $\Gamma \vdash M_1 \lesssim_A M_2$, if and only if, for all Γ-closing substitutions θ_1, θ_2 if $\theta_1 \lesssim_\Gamma \theta_2$ then $\theta_1(M_1) \lesssim_A \theta_2(M_2)$.

Logical approximation is closed under primitive reduction, and expansion.

Lemma 4.24 ($\lesssim^{trm}$ left-closed under $\rightsquigarrow$)**.** *If $M \rightsquigarrow P$ and $M \lesssim^{trm}_{A} N$ then $P \lesssim^{trm}_{A} N$, and if $M \rightsquigarrow P$ and $P \lesssim^{trm}_{A} N$ then $M \lesssim^{trm}_{A} N$.*

Lemma 4.25 ($\lessapprox^{trm}$ right-closed under $\rightsquigarrow$)**.** *If $N \rightsquigarrow P$ and $M \lessapprox_A^{trm} N$ then $M \lessapprox_A^{trm} P$, and if $N \rightsquigarrow P$ and $M \lessapprox_A^{trm} P$ then $M \lessapprox_A^{trm} N$*

The following lemma follows directly from the above definition of open logical approximation.

Lemma 4.26. *If $\Gamma \vdash M_1 \lessapprox_A^{trm} M_2$ and $\Delta \vdash \theta_1 \lessapprox_\Gamma \theta_2$ then $\Delta \vdash \theta_1(M_1) \lessapprox_A^{trm} \theta_2(M_2)$.*

Lemma 4.27 (Logical Apx. Term Closure)**.** *Logical approximation is closed under the compound term formers (Figure 4.8).*

Proof. By Lemmas 4.24 and 4.25, and the LET rule for big-step evaluation. □

Lemma 4.28 (Fundamental Theorem of Logical Relations)**.** *Logical approximation is reflexive:*

1. $\Gamma \vdash V \lessapprox_A^{val} V$
2. $\Gamma \vdash M \lessapprox_A^{trm} M$

Proof. Perform simultaneous induction on the typing derivations $\Gamma \vdash V : A$ and $\Gamma \vdash M : A$ using Lemma 4.27. □

4.2.3 Completing the Triangle

It remains to establish the other two implications of the triangle. First, we link applicative and logical approximation with the following generalised transitivity property [Pitts, 2005; Pitts and Stark, 1998]:

Lemma 4.29. *The following transitivity properties hold:*

1. *If $U \lessapprox_A^{val} V$ and $V \lessapprox_A^{val} W$ then $U \lessapprox_A^{val} W$*
2. *If $M \lessapprox_A^{trm} N$ and $N \lessapprox_A^{trm} P$ then $M \lessapprox_A^{trm} P$*

Proof. By simultaneous induction on the structure of type A. □

The above transitivity property is similar to the same property for Howe's [1989] *precongruence candidate*, $\mathcal{R}^H$, given a transitive relation $\mathcal{R}$, we compose with $\mathcal{R}$ *on the right*:

$$U \mathcal{R}^H V \wedge V \mathcal{R} W \implies U \mathcal{R}^H W \quad \text{(Howe)}$$

whereby setting $\mathcal{R}^H \triangleq \lessapprox^{val}$ and $\mathcal{R} \triangleq \lesssim^{val}$, we see the connection to our Lemma 4.29. Note, however, our property is also provable if we switch the order of $\lessapprox^{val}$ and $\lesssim^{val}$, whereas, to our knowledge, the precongruence candidate is always defined such that (Howe) is the only valid formulation. We discuss Howe's method further in Section 4.4.2.

Using the Fundamental Property of Logical Relations and Lemma 4.29, we obtain the base implication of our triangle.

Lemma 4.30. *Applicative approximation implies logical approximation:*

$$\Gamma \vdash M \lesssim_A^{trm} N \implies \Gamma \vdash M \lessapprox_A^{trm} N$$

Now we establish a relationship between observational and logical approximations by first showing that logical approximation is closed under all VSCs.

Lemma 4.31. *If* $\Gamma \vdash M_1 \lessapprox_A^{trm} M_2$ *and* $\Delta \vdash \mathcal{C} : \langle\langle \Gamma \vdash A \rangle\rangle B$ *then,*

$$\Delta \vdash \mathcal{C}[M_1] \lessapprox_B^{trm} \mathcal{C}[M_2]$$

Proof. By induction on the derivation of $\Delta \vdash \mathcal{C} : \langle\langle \Gamma \vdash A \rangle\rangle B$ using Lemma 4.27 for the compound term formers, Lemma 4.26 for the VSCHOLE case, and Lemma 4.28 for the case where there is no hole in $\mathcal{C}$. □

Lemma 4.32. *Logical approximation implies observational approximation:*

$$\Gamma \vdash M_1 \lessapprox_A^{trm} M_2 \implies \Gamma \vdash M_1 \lesssim_A^{VSC} M_2$$

Proof. Instantiate the context $\mathcal{C}$ of Lemma 4.31 with program context $\cdot \vdash \mathcal{P} : \langle\langle \Gamma \vdash A \rangle\rangle B$ from the definition of VSC approximation. □

We can now complete the cycle, which in turn yields the desired context lemma: that applicative contexts are sufficient to characterise observational approximation.

Theorem 4.33 (Big-Step Triangulation)**.** *Observational, applicative, and logical approximation coincide.*

Proof. Using Lemmas 4.20, 4.30 and 4.32. □

Corollary 4.34 (Context Lemma)**.** *For all open* $\lambda_{FG}^{\rightarrow}$ *terms* $\Gamma \vdash M, N : A$*,*

$$\Gamma \vdash M \lesssim_A^{ASC} N \text{ if and only if } \Gamma \vdash M \lesssim_A^{VSC} N.$$

4.3 Extension to Frame Stacks

Since Milner's original context lemma, a number of similar results have appeared for more complex calculi. In particular, for call-by-value calculi with effects (such as state) applicative contexts are not sufficient: such contexts to do not account for manipulation of state cells during reduction to a value. Instead, one requires a more powerful notion known as *Closed Instantiations of Uses* (CIU) approximation due to Mason and Talcott [1991].

In this section we consider a different triangle of approximations involving *frame stacks* [Pitts, 2005]. We parallel the ReFS proofs of Pitts and Stark [1998] recasting the applicative and logical approximations described in Section 4.2 in terms of frame stack evaluation. First though, we define CIU contexts and relate them to VCCs just like that for ASCs. We omit some proofs since they follow similar reasoning. For an alternative account of a similar development, the reader may wish to consult the chapter by Pitts [2005] in Pierce [2004].

Definition 4.35 (CIU Contexts)**.** Let *CIU contexts* be the restricted class of contexts defined by the following inference rules:

$$\textsc{ciuHole}\ \frac{\theta : \Gamma \vDash \Delta}{\Delta \vdash \langle\langle \theta - \rangle\rangle : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{CIU}} A} \qquad \textsc{ciuApp}\ \frac{\Delta \vdash \mathcal{D} : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{CIU}} B \qquad \Delta, B \vdash N : C}{\Delta \vdash \mathbf{let}_B\ \mathcal{D}\ N : \langle\langle \Gamma \vdash A \rangle\rangle^{\mathrm{CIU}} C}$$

It is worth stressing the differences between ASCs and CIU contexts. ASCs consist of a well-typed and well-scoped hole applied to a sequence of closed values. CIU contexts form a nested sequence of let bindings (culminating in the hole) with a term body.

Definition 4.36 (CIU Approximation)**.** Let CIU approximation be Definition 4.7 with $\mathcal{K} = \mathrm{CIU}$.

Definition 4.37. Define the operation $\blacklozenge : \mathit{CIU} \to \mathit{VCC}$, which transforms a closed CIU context into a closed VCC, by induction on the structure of the CIU context:

$$\blacklozenge(\langle\langle \theta - \rangle\rangle) = \langle\langle - \rangle\rangle^{\theta}$$

$$\blacklozenge(\mathbf{let}\ \mathcal{P}\ N) = \mathbf{let}\ \blacklozenge(\mathcal{P})\ N$$

Lemma 4.38. *If* $\Gamma \vdash M \lesssim_A^{\mathit{VCC}} N$ *then* $\Gamma \vdash M \lesssim_A^{\mathit{CIU}} N$.

Well-Typed Syntax

$$\frac{}{\cdot \vdash Id : A \multimap A}\ \textsc{NilFrmTy} \qquad \frac{\cdot \vdash S : B \multimap C \qquad A \vdash N : B}{\cdot \vdash S \circ_A N : A \multimap C}\ \textsc{ConsFrmTy}$$

Semantics

$$\frac{\cdot \vdash V : A}{\langle Id, V\rangle \downarrow_A V}\ \textsc{nilValue} \qquad \frac{\langle S, N[U]\rangle \downarrow_B V}{\langle S \circ_A N, U\rangle \downarrow_B V}\ \textsc{consValue} \qquad \frac{M \rightsquigarrow M' \qquad \langle S, M'\rangle \downarrow_A V}{\langle S, M\rangle \downarrow_A V}\ \textsc{primRed}$$

$$\frac{\langle S \circ_A N, M\rangle \downarrow_B V}{\langle S, \mathbf{let}_A\ M\ N\rangle \downarrow_B V}\ \textsc{seq}$$

Figure 4.9: $\lambda^{\rightarrow}_{FG}$ frame stack syntax and semantics

4.3.1 Frame Stack Properties

In this section, we establish some properties of frame stacks, a well-typed construct satisfying the inference rules in Figure 4.9. A frame stack is a stack consisting of open terms each containing exactly one free variable. The frame stack type $A \multimap B$ denotes a stack with *argument* type A and return type B. For example, ConsFrmTy says that if we have a frame stack S of type $B \multimap C$, expecting an argument of type B, and a term N of type B with a free variable of type A then we can form the frame stack $S \circ_A N$ of type $A \multimap C$, expecting an argument of type A. The argument to a frame stack mimics that of a hole in a closed VCC.

Evaluation of a frame stack is with respect to a *focussed* term whose type corresponds to the argument type of the frame stack. For readability we mostly suppress typing annotations on frame stack evaluation, just as for big-step evaluation.

Define the (left) action over a frame stack, @, to be the operation that produces a term given a frame and a closed term to fill its hole:

$$Id@M = M$$

$$(S \circ_A N)@M = S@(\mathbf{let}_A\ M\ N)$$

We may extend the @ action to operate on closed CIU contexts instead of terms, denoted by $- \langle\langle @\rangle\rangle -$, replacing the case for ConsFrmTy above with:

$$(S \circ_A N)\langle\langle @\rangle\rangle \mathcal{P} = S\langle\langle @\rangle\rangle(\mathbf{let}_A\ \mathcal{P}\ N)$$

The type of the holes in the context are left unchanged by the operation. The following lemma establishes that the action over a frame stack commutes with context instantiation.

Lemma 4.39. $(S\langle\langle @\rangle\rangle\mathcal{P})[M]^{CIU} = S@(\mathcal{P}[M]^{CIU})$

Proof. By induction on the frame stack S. □

The following lemma relates frame stack and big-step evaluation using the action over a frame stack.

Lemma 4.40. $\langle S,M\rangle \downarrow V \iff S@M \Downarrow V$.

Proof. For $\Longleftarrow$, the proof follows by the properties:

1. $M \Downarrow V \implies \langle Id,M\rangle \downarrow V$

2. $\langle Id,S@M\rangle \downarrow U \implies \langle S,M\rangle \downarrow U$

where (1) is by induction on the derivation of $M \Downarrow V$ and (2) is by induction on S.

For $\Longrightarrow$, the proof follows from a standardisation argument [Takahashi, 1995]: evaluation can be decomposed into evaluation of a *focussed* term M to value W and evaluation of the surrounding context S filled with W.

3. $S@M \Downarrow V \implies \exists\, W.M \Downarrow W \wedge S@W \Downarrow V$

4. $M \Downarrow W \wedge S@W \Downarrow V \implies S@M \Downarrow V$

Both are proved simultaneously by induction on S. □

We define a relation transformer that lifts a relation on values to a relation between frame stack configurations $\langle -,-\rangle$.

Definition 4.41. For frame stacks $S_1,S_2 : A \multimap B$, closed terms $M_1,M_2 : A$, and binary relation $\mathcal{R}$ on closed values of type B, relation $S_1,M_1 \ \lfloor\mathcal{R}\rfloor^F \ S_2,M_2$ holds if and only if

$$\forall\, V.\ \langle S_1,M_1\rangle \downarrow V \implies \exists U.\ \langle S_2,M_2\rangle \downarrow U \wedge V\ \mathcal{R}\ U\,.$$

4.3.2 Applicative Frame Approximation

Having set up frame-stack machinery, we move on to the associated approximation relations. In Sections 4.2.1 and 4.2.2 we defined value and term relations simultaneously, using $\downarrow - \downarrow^T$ to pass from values to terms. In a CIU setting, frame stacks now provide a bridge between value relations and term relations. This is most obvious with logical approximations, where all three are defined simultaneously. For applicative approximation, our results need only the term relation.

Definition 4.42 (Applicative Frame Approximation)**.** Term M_1 applicatively frame approximates M_2 at type A if evaluation of M_1 in any appropriately-typed frame stack S approximates the evaluation of M_2 in the same S.

$$M_1 \ \langle\lesssim\rangle_A^{trm} \ M_2 \triangleq \forall S : A \multimap B.\ S, M_1 \ \downarrow \approx_B \downarrow^F \ S, M_2$$

Approximation for values mirrors that of Definition 4.17, building on this term approximation; while two frame stacks are related based on their evaluation at all values. We omit the formal descriptions as we do not need them further.

Remark 4.43. We have retained the terminology of Section 4.2 for our extension to frame stacks, uniformly referring to the relation occupying the lower-right corner of (both instances of) the triangle as the "applicative" relation. Our terminology is inspired by Milner's original applicative relation which relates functions by applying them to an identical sequence of arguments. However, our usage of the term is more abstract than his technical definition. In particular, we include Definition 4.42 under the term "applicative" which is not considered an applicative relation in Milner's sense due to the ability to test functions using a larger collection of contexts, *e.g.* contexts where the functions being related may themselves be arguments. Whilst for $\lambda^{\rightarrow}_{FG}$, this increase in testing contexts does not affect our discriminatory power, the introduction of effects and handlers is enough to render Milner's applicative contexts unsuitable for reasoning about observational approximation, and justifies the use of the more powerful frame stacks formulation. See Section 6.5.4 for an extended discussion in the setting of our effectful ELLA calculus. Despite these differences, we opt to retain the *logical/applicative* terminology for describing the situation in an abstract sense. The logical notion tests functions with *related* frames and values. Whereas, the applicative notion tests functions with *identical* frames and values (cf. Section 2.3). That is, our usage of the term "applicative" is primarily to distinguish it from the logical notion which recursively appeals to itself when relating terms and values.

Lemma 4.44. *If* $\Gamma \vdash M \lesssim_A^{CIU} N$ *then* $\Gamma \vdash M \langle\lesssim\rangle_A^{trm} N$

Proof. Assume a Γ-closing substitution θ with $\langle S, \theta(M)\rangle \downarrow V$ for some S and V. By Lemma 4.40, $S@\theta(M) \Downarrow V$ holds. In the CIU approximation assumption, set the context $\mathcal{P}$ to be $S\langle\langle@\rangle\rangle\langle\langle\theta-\rangle\rangle$. It follows that there exists a value W such that

$$(S\langle\langle@\rangle\rangle\langle\langle\theta-\rangle\rangle)[N]^{\mathrm{CIU}} \Downarrow W \text{ and } V \cong W$$

The result follows by Lemmas 4.39 and 4.40. □

Lemma 4.45. *Observational approximation implies applicative frame approximation.*

Proof. By Lemmas 4.9, 4.38 and 4.44. □

4.3.3 Logical Frame Approximation

We now define the frame stack analogue to logical approximation from Section 4.2.2, using $\top\top$-lifting / biorthogonality for logical relations as demonstrated by Pitts and Stark [1998].

Definition 4.46 (Logical Frame Approximation)**.** Logical frame approximation is defined by three mutually recursive relations on pairs of (closed) values, terms and stacks. Define $V_1 \langle\lessapprox\rangle_A^{val} V_2$ for closed values V_1, V_2 inductively by the structure of type A:

$$\frac{V_1 \cong_\tau V_2}{V_1 \langle\lessapprox\rangle_\tau^{val} V_2}\ \textsc{LogFrmGndApx} \qquad \frac{\forall\, V_1, V_2.\ V_1 \langle\lessapprox\rangle_A^{val} V_2 \implies M_1[V_1] \langle\lessapprox\rangle_B^{trm} M_2[V_2]}{\lambda_A M_1 \langle\lessapprox\rangle_{A\to B}^{val} \lambda_A M_2}\ \textsc{LogFrmAbsApx}$$

The relation $\langle\lessapprox\rangle^{trm}$ is defined by the $\top\top$-lifting of $\langle\lessapprox\rangle^{val}$ over $\langle\lessapprox\rangle^{stk}$: for closed terms M_1 and M_2, approximation $M_1 \langle\lessapprox\rangle_A^{trm} M_2$ is defined to be

$$\forall S_1, S_2.\ S_1 \langle\lessapprox\rangle_{A\multimap B}^{stk} S_2 \implies S_1, M_1 \ \lfloor\cong_B\rfloor^F\ S_2, M_2$$

where the approximation $S_1 \langle\lessapprox\rangle_{A\multimap B}^{stk} S_2$ for frame stacks S_1 and S_2 is defined as

$$\forall\, V_1, V_2.\ V_1 \langle\lessapprox\rangle_A^{val} V_2 \implies S_1, V_1 \ \lfloor\cong_B\rfloor^F\ S_2, V_2\,.$$

As with applicative frame approximation, $\langle\lessapprox\rangle^{trm}$ uses the $\cong$ relation to relate the final values. In the definition of $\langle\lessapprox\rangle^{stk}$, we implicitly lift the value arguments to terms.

As in Lemmas 4.24, 4.25, and 4.31, logical frame approximation is closed under primitive reduction and expansion, and closed under all VSCs.

Lemma 4.47 (Logical Frame Apx. Lifts)**.** *Logical frame approximation is closed under the compound term formers.*

Proof. The only deviation from the proof of Lemma 4.27 is for **let**, where we extend the approximation of stacks from $S_1 \mathrel{\langle\!\lessapprox\rangle^{stk}_{B\multimap C}} S_2$ to $S_1 \circ_A N \mathrel{\langle\!\lessapprox\rangle^{stk}_{A\multimap C}} S_2 \circ_A N'$. □

Lemma 4.48. *Logical frame approximation is reflexive:*

1. $\Gamma \vdash V \mathrel{\langle\!\lessapprox\rangle^{val}_{A}} V$
2. $\Gamma \vdash M \mathrel{\langle\!\lessapprox\rangle^{trm}_{A}} M$
3. $S \mathrel{\langle\!\lessapprox\rangle^{stk}_{A}} S$

Proof. For (1) and (2) the proof is similar to Lemma 4.28 using Lemma 4.47. For (3), perform induction on the well-typed derivation of S using (2) for CONSVALUE. □

Lemma 4.49. *If* $\Gamma \vdash M_1 \mathrel{\langle\!\lessapprox\rangle_A} M_2$ *and* $\Delta \vdash \mathcal{C} : \langle\!\langle \Gamma \vdash A \rangle\!\rangle B$ *then*

$$\Delta \vdash \mathcal{C}\langle\!\langle M_1 \rangle\!\rangle \mathrel{\langle\!\lessapprox\rangle^{trm}_{B}} \mathcal{C}\langle\!\langle M_2 \rangle\!\rangle$$

4.3.4 Establishing the Triangle

We can now proceed to the following analogue of Lemma 4.32.

Lemma 4.50. *Logical frame approximation implies observational approximation:*

$$\Gamma \vdash M \mathrel{\langle\!\lessapprox\rangle^{trm}_{A}} N \implies \Gamma \vdash M \mathrel{\lesssim^{VSC}_{A}} N\,.$$

Proof. By definition, using Lemma 4.49. □

Lemma 4.51. *Applicative and logical frame approximation satisfy the following transitivity property:*

$$\textit{If } M \mathrel{\langle\!\lessapprox\rangle^{trm}_{A}} N \textit{ and } N \mathrel{\langle\!\lesssim\rangle^{trm}_{A}} P \textit{ then } M \mathrel{\langle\!\lessapprox\rangle^{trm}_{A}} P.$$

Proof. By induction on the structure of type A. □

We establish the final implication of the triangle: that applicative frame approximation implies logical frame approximation.

Lemma 4.52.

$$\Gamma \vdash M \mathrel{\langle\!\lesssim\rangle^{trm}_{A}} N \implies \Gamma \vdash M \mathrel{\langle\!\lessapprox\rangle^{trm}_{A}} N$$

Proof. By Lemmas 4.48 and 4.51. □

Once again, all three notions coincide.

Lemma 4.53 (Frame Stack Triangulation)**.** *Observational, applicative frame and logical frame approximation coincide.*

Proof. Using Lemmas 4.45, 4.52 and 4.50. □

For a simply-typed lambda calculus like $\lambda^{\rightarrow}_{FG}$ it is not in fact essential to consider CIU approximation — we already know from Section 4.2 that applicative contexts are enough to distinguish terms. Differences, though, arise in any language with features like state, exceptions, or other effects that give program contexts greater discriminating power. In these cases we expect the frame-stack approach to be appropriate, and note that the triangulation proofs are reassuringly similar in shape when moving from Milner's original context lemma to a CIU version. In particular, in Chapter 6 we generalise frame stacks to *handler stacks* to extend our development to an effectful calculus supporting algebraic effects and handlers.

4.4 Related Work

There is a considerable body of work concerning program equivalence results. We highlight publications which had a direct influence on our development, and those closely related in terms of pertaining to completeness results, or the formalisation of such results in theorem provers.

4.4.1 Triangulation

Stark [1994] showed a triangle of relations coincide for the ν-calculus and proved a context lemma for a variant of Milner's result which applied the hole to a test function. However, this special form of context lemma is proved by analysing the process of reduction and the particular forms for closed expressions in the calculus. In contrast, our fine-grained calculus is simple enough for consideration of applicative contexts à la Milner [1977] to suffice, and hence the context lemma follows from the triangulation result.

Pitts and Stark [1998] prove a context lemma for ReFS, a functional language with local state. Their use of the triangulation technique inspired this work and our Section 4.3 essentially isolates their approach, free from considerations of state relations.

However, our formal approach gives a more satisfactory account of contexts, their scope and the variable capture involved when instantiating a hole with a term. In particular, there is no requirement for a side-condition in our analogue (Theorem 4.49) of their Theorem 4.9 since by construction every occurrence of a hole (in a VSC) is paired with a well-typed and well-scoped substitution.

Pitts [2005] developed a triangle for an ML-like language, but in contrast to ReFS, it is based on an extension of Howe's relational approach [Gordon, 1994; Lassen, 1998a]. The relational approach for representing contexts is favoured in order to mitigate the difficulties involved with concrete contexts and their potential capture of free variables of a term. The folklore belief is that such issues are especially difficult to handle in the setting of machine checked proofs. On the contrary, a careful choice of representation and appropriate use of state-of-the-art technology makes their representation routine. Moreover, the relational approach relies on an inductive characterisation in terms of the language syntax. Some of these rules involve the use of variables from the context. It is not clear how to formalise such definitions without having to deal with issues of naming and α-convertibility; issues handled automatically by the ACMM framework.

Johann et al. [2010] prove a CIU-like context lemma result in a general operational setting for a polymorphic calculus extended with algebraic effects. Like us, they parameterise their approximations with respect to a notion of 'basic' preorder for making observations on ground type computations. Their expressive setting provides justification for the parameterisation, permitting different choices of basic preorder to give a semantics with respect to different collections of effects. Their calculus does not support handlers for algebraic effects, thus the basic preorders can be seen as providing the sole interpretation for a given effect.

4.4.2 Howe's Method

Howe [1989] developed a general method for proving coincidence of applicative bisimilarity and observational equivalence for a broad class of (untyped) lazy languages and subsequently extended the method to support call-by-value calculi [Howe, 1996]. A key element of the approach is to introduce a relation, called the 'precongruence candidate', which bridges the gap between the applicative and observational notions. The technique has been shown to extend to other languages, including a typed metalanguage with recursive types [Gordon, 1994] and extensions of PCF [Pitts, 1997;

Lassen, 1998a]. Howe's method does not reason explicitly about reduction sequences in contrast to others [Mason and Talcott, 1991; Stark, 1994]. Instead, the precongruence candidate is sufficient to show that the relation is closed under all contexts, treating contexts abstractly. Our approach is a middle ground in that we reason explicitly about the structure of contexts but do not explicitly analyse reduction sequences.

Note, that whilst Gordon [1994] employs Howe's method for showing coincidence of applicative bisimilarity and observational equivalence, he separately proves a context lemma result using a variation on Milner's [1977] careful analysis on reduction sequences and dissection of terms. Pitts [2011] has shown Howe's method can also be used to prove such context lemmas. Based on this observation and our existing results, we conjecture that the triangulation method is a way to understand both inductive and coinductive characterisations of observational equivalence. It is future work to investigate the relationship between the triangulation technique described here and Howe's method.

4.4.3 Formalisation

We contribute a concrete instance of using ACMM in the study of observational approximation and its coincidences. We see our work as a solution to another challenge problem in the broader context of mechanising programming language meta-theory, in the sense of Aydemir et al. [2005], and more recently, Abel et al. [2019]. The initial proposal by Aydemir et al. focussed on basic meta-theory of System F with subtyping, such as type soundness, and its solutions highlight the various representation choices available [Leroy, 2007; Aydemir et al., 2008; Vouillon, 2012]. In this context, ACMM is a particular approach for addressing these issues in a uniform way, allowing the focus to shift to more challenging meta-theoretic results, *e.g.* the strong normalisation proofs studied by Abel et al. [2019], or context lemma results presented in this chapter. Moreover, the extension of ACMM by Allais et al. [2018] to a framework operating over *generic descriptions* of syntaxes with binding enables significant re-use of basic 'infrastructure' lemmas; properties regarding weakening and substitution, and their various compositions.

Ambler and Crole [1999] formalise program equivalence for a call-by-name λ-calculus using applicative bisimulation and Howe's method in Isabelle/HOL using a first-order de Bruijn representation. The authors note the difficulty in proving properties of weakening and substitution, particularly for the relations between terms. Our

statement of analogous properties are simplified by virtue of our well-typed and-scoped encoding of terms. Furthermore, the proofs of such properties are simplified by leveraging a weak HOAS representation for defining the semantics of such operations.

Momigliano et al. [2002] replay Ambler and Crole's [1999] development as an elaborate case study for the Hybrid system [Ambler et al., 2002], a meta-language defined on top of Isabelle/HOL. The development does not include a proof of substitutivity for Howe's candidate relation due to difficulties in the encoding. Hybrid allows the representation of object languages by HOAS such that they admit principles of (co)induction. The underlying encoding of Hybrid within Isabelle/HOL carves out acceptable object language HOAS terms by translation to a subset of de Bruijn expressions. This representation yields proofs of the desired induction principles for the object language(s).

In BELUGA, Momigliano et al. [2018] formalise a proof of pre-congruence for similarity using Howe's method in the setting of a call-by-name λ-calculus. The authors take full advantage of an extension of LF [Harper et al., 1993] types, which support first-class contexts and simultaneous substitutions, provided by BELUGA's underlying contextual modal type theory [Pientka, 2008]. Their development does not include a context lemma result, omitting the proof of coincidence between observational approximation and similarity. Indeed, their HOAS approach does not support any kind of variable-capturing contexts, while the authors note Beluga requires a non-trivial extension to encode observational approximation using the relational approach [Lassen, 1998b; Pitts, 2011].

Table 4.1: Selected notation used in the chapter and corresponding AGDA definitions

Concept	Notation	Reference	Agda Definition
Simultaneous substitution	$\theta(-)$	Definition 4.1	`*-Val`
Lookup in substitution	θ_k	Definition 4.2	`var`
Successor substitution	$\mathbf{succ}-$	Definition 4.2	`succ`
Lift value relation to terms by evaluation	$\lfloor - \rfloor^T$	Definition 4.4	`_[_]^T_`
Ground equivalence	$\approx$	Definition 4.6	`gnd-eqv`
Variable-capturing contexts	$\langle\!\langle - \vdash - \rangle\!\rangle^{\mathrm{VCC}}$	Figure 4.6	$\mathtt{VCC}\langle\langle \_ \vdash \_ \rangle\rangle$
Observational approximation	$\lesssim^{\mathrm{VSC}}$	Definition 4.8	`vsc-apx`
Applicative substituting contexts	$\langle\!\langle - \vdash - \rangle\!\rangle^{\mathrm{ASC}}$	Definition 4.10	$\mathtt{ASC}\langle\langle \_ \vdash \_ \rangle\rangle$
ASC approximation	$\lesssim^{\mathrm{ASC}}$	Definition 4.11	`asc-apx`
Let-redexes from substitution	$\mathcal{D}^{\theta}$	Definition 4.12	`VCC-sub`
Transform ASC to VCC	$\star$	Definition 4.14	`asc-to-vcc`
Applicative approximation	$\lesssim$	Definition 4.17	$\mathtt{app\text{-}apx}_{\emptyset}$
Open applicative approximation	$\Gamma \vdash M \lesssim_A N$	Definition 4.18	`app-apx`
Logical approximation	$\lessapprox$	Definition 4.21	$\mathtt{log\text{-}apx}_{\emptyset}$
Open logical approximation	$\Gamma \vdash M \lessapprox_A N$	Definition 4.23	`log-apx`
CIU contexts	$\langle\!\langle - \vdash - \rangle\!\rangle^{\mathrm{CIU}}$	Definition 4.35	$\mathtt{CIU}\langle\langle \_ \vdash \_ \rangle\rangle$
CIU Approximation	$\lesssim^{\mathrm{CIU}}$	Definition 4.36	`ciu-apx`
Transform CIU Contexts to VCC	$\blacklozenge$	Definition 4.37	`ciu-to-vcc`
Frame stacks relation transformer	$\lfloor - \rfloor^F$	Definition 4.41	`_[_]^F_`
Applicative Frame Approximation	$\langle\lesssim\rangle$	Definition 4.42	$\mathtt{app\text{-}frm\text{-}apx}_{\emptyset}$
Logical Frame Approximation	$\langle\lessapprox\rangle$	Definition 4.46	$\mathtt{log\text{-}frm\text{-}apx}_{\emptyset}$

Chapter 5

A Logic for Relational Reasoning

The previous chapter demonstrated the triangulation proof method for $\lambda^{\rightarrow}_{FG}$, a simply-typed λ-calculus without effects. Since we claim triangulation is a general technique, we want to extend our results to a calculus supporting computational effects. In particular, Chapter 6 presents ELLA, a FRANK-inspired calculus with first-class support for effect handlers.

Recall that handlers can be described as either deep or shallow. Deep handlers are automatically reinvoked on the continuations of handled computations, whereas shallow handlers are not. FRANK (hence also ELLA) handlers are shallow so the handling of subsequent effects is expressed using general recursion.

To extend triangulation to such a calculus, we might think to follow the recipe laid out in Chapter 4, defining the relations recursively on the structure of types. However, in the presence of general recursion we need to be a bit more careful. Ultimately, we want to show both structural relations are compatible with the term formers of the calculus, especially the logical relation [1]. In particular, given the recursive function rule:

$$\frac{\Gamma, A \to B, A \vdash M : B}{\Gamma \vdash \mathbf{rec}_A\, M : A \to B}\ \text{REC}$$

compatibility amounts to showing it is preserved by the logical relation $\lessapprox$:

$$\frac{\Gamma, A \to B, A \vdash M_1 \lessapprox^{trm}_B M_2}{\Gamma \vdash \mathbf{rec}_A\, M_1 \lessapprox^{val}_{A \to B} \mathbf{rec}_A\, M_2} \quad (\star)$$

[1] Compatibility of the applicative notion follows from its connection to the logical notion; the base implication of the triangle. Compatibility of the logical relation implies soundness w.r.t. observational approximation.

The antecedent of $(\star)$ substitutes the recursive function into the respective bodies. Naïvely appealing to the definition of $\lessapprox^{val}$ at the type $A \to B$ does not get us anywhere: we would need to show the substitutions are approximate; the exact approximation we are trying to show!

Instead, we would first prove an unwinding theorem [Pitts and Stark, 1998], a syntactic analogue to the denotational proof principle of fixed point induction: that the observable behaviour of a recursive function is completely captured by its finite unwindings. Second, we prove $(\star)$ holds for all finite unwindings by induction on the number of unwindings; the base case of a non-recursive function follows from the definition of $\lessapprox^{val}$ at function type. Hence, we may deduce $(\star)$ for recursive functions using the unwinding theorem. Pitts and Stark [1998] have proven an unwinding theorem for the ReFS calculus, which is subsequently used to show the compatibility property for their operational logical relation. See also Pitts [2005] for a similar account.

While we have strong reason to believe an unwinding theorem would suffice for ELLA, we pursue an alternative approach known as *step-indexing*. This approach involves augmenting the structural approximation relations with a natural number counter (the eponymous step-index) representing for how many steps of computation the approximation holds. Now our relations are defined by a double recursion: first on the structure of types then on the step-index. Using explicit steps, we can resolve the dilemma introduced by general recursion quite simply, by decrementing the step-index in the premiss of the rule for function types. We choose the step-indexing approach because it is mathematically straightforward yet scales remarkably well to support reasoning about other recursive features, including recursive types and general references [Ahmed, 2004, 2006]. So the literature on step-indexing suggests how we might extend the forthcoming formalism for ELLA to the full Frank language, which has recursive types via algebraic datatypes, and ML-style references. In contrast, unwinding does not scale to support these features; further syntactic analogues of denotational methods are required and these techniques can become rather complex, e.g. *syntactic minimal invariance* for supporting recursive types [Birkedal and Harper, 1999]. Furthermore, there are several formalised λ-calculus developments based on step-indexing [Appel et al., 2007; Jung et al., 2018b; Polesiuk, 2017] whilst, to our knowledge, no such developments based on unwinding theorems. Without a point of comparison, we cannot claim step-indexed relations are more amenable to formalisation than proving an unwinding theorem but their scalability is certainly advantageous given the desire for mechanised metatheory to be extensible.

Step-indexed propositions satisfy a monotonicity condition: if a property holds for n steps of computation then it holds for any $m < n$. For a good intuition as to why this makes sense, imagine we are trying to distinguish terms by their behaviour. If two terms cannot be distinguished by n steps of computation then they certainly will not be distinguished by any $m < n$. While step-indexing is mathematically quite simple, proofs involving step-indexed relations can become marred in tedious step-index arithmetic.

To alleviate the theorist of this burden, there has been a lot of work recently on modal logics to hide the index manipulation within proofs [Appel et al., 2007; Dreyer et al., 2011; Jung et al., 2018b]. These logics deal with implicitly step-indexed propositions and silently thread the step-index through formulae. In places where the index must be decremented, the *later* modal operator $\triangleright$ is used with the semantics that if a proposition P is true at index n then $\triangleright P$ is true at index $n+1$. The later modality satisfies the following induction principle, called *Löb induction* due to its connection to the Gödel-Löb logic of provability [Appel et al., 2007; Nakano, 2001]:

$$\frac{\textsc{Löb} \quad \triangleright P \vdash P}{\vdash P}$$

which says that to prove a proposition holds at some index $n > 0$ we may assume it holds at $n-1$. The induction principle has no base case because all propositions are trivially true at index 0; being the behaviour one can observe in zero computation steps. Coupled with the monotonicity condition, we obtain an induction principle which allows one to assume a formula holds at all steps less than the current one in order to prove the formula holds 'now', *i.e.* at the current index. The induction principle has a coinductive flavour to it: in order to prove something is true 'now' it suffices to prove it holds 'now' under the assumption it holds 'later'.

A number of useful high-level reasoning properties can be derived using Löb induction. In contrast, the unwinding theorem does not provide such a high-level induction principle. Typically you are forced to reason directly on termination derivations which are subtly different for particular example approximations.

To abstract away the threading of step-indices through our proofs, we define a logic to hide step-indices, taking inspiration primarily from the LSLR logic by Dreyer et al. [2011] and *complete ordered families of equivalences* (COFEs) by Di Gianantonio and Miculan [2003]. Another noteworthy logic in this area is Iris [Jung et al., 2018b] which has been used to verify a variety of properties for several higher-order imperative

languages, *e.g.* studies by Jung et al. [2018a] and Timany and Birkedal [2019].

Since ELLA is a monomorphic calculus, we do not require quantification over the interpretation of types, as would be required in calculi supporting polymorphism, *e.g.* System F, or even FRANK which, in contrast to System F, only supports prenex polymorphism. We do, however, require fixed-point predicates to be able to define the applicative and logical approximations which have recursive definitions. Thus, we may restrict our logic to first-order intuitionistic logic extended with fixed-point predicates and the $\triangleright$ modality for manipulating the step-index. Both IRIS and LSLR are more sophisticated logics than the fragment we consider, being based on higher-order and second-order logic, respectively.

5.1 A Modal Logic for Step-Indexed Propositions

In the previous chapter, our reasoning for $\lambda^{\rightarrow}_{FG}$ was conducted within a Martin-Löf logical framework. Using step-indexing to extend the approach to ELLA, we define a modal logic called $\texttt{FOL}^{\mu\triangleright}$ which implicitly carries the step-indexing through formulae. Our logical and applicative notions, and our reasoning principles are embedded within $\texttt{FOL}^{\mu\triangleright}$. $\texttt{FOL}^{\mu\triangleright}$ is defined with respect to a multi-sorted signature Ω. Therefore, $\texttt{FOL}^{\mu\triangleright}$ is **independent of ELLA or any other calculus we choose to study**. In Chapter 6, we instantiate the signature to ELLA types, terms and stacks, and functions and relations over them.

Definition 5.1. A multi-sorted signature Ω consists of:

- a collection $\mathbb{S}$ of sorts $\mathcal{S}$;
- a collection of typed function symbols $f : \overline{\mathcal{S}} \to \mathcal{S}$;
- a collection of typed relation symbols $\mathcal{A} : \overline{\mathcal{S}}$.

where $\overline{\mathcal{S}}$ denotes a sequence of zero or more sorts from $\mathbb{S}$.

The relation symbols occurring in the signature are *atomic* (hence the choice of metavariable) in the sense that they are independent of step-indexing.

We present the grammar for $\texttt{FOL}^{\mu\triangleright}$ in Figure 5.1. Note that we implicitly inject applications $\mathcal{A}\ \bar{t}$ of atomic relations $\mathcal{A}$ to terms $\bar{t}$ into step-indexed propositions ϕ. Additionally, we may define (implicitly) step-indexed predicates π by abstraction over a proposition, or by the fixed-point construction, μ. Later we introduce constraints

$$\begin{aligned}
&(\text{propositions})\ \phi, \psi ::= \pi\, \bar{t} \mid \mathcal{A}\, \bar{t} \mid \phi \vee \psi \mid \phi \wedge \psi \mid \triangleright\, \phi \\
&\qquad\qquad\qquad\qquad \mid\ \phi \implies \psi \mid \forall x : \mathcal{S}.\ \phi \mid \exists x : \mathcal{S}.\ \phi \\
&(\text{predicates})\qquad \pi ::= P \mid (\bar{x} : \overline{\mathcal{S}}).\phi \mid \mu P : \overline{\mathcal{S}}.\pi
\end{aligned}$$

Figure 5.1: Syntax for $\mathtt{FOL}^{\mu\triangleright}$

on the occurrence of predicate variables P in fixed-point predicates to ensure such predicates are well-founded.

We define $\mathtt{FOL}^{\mu\triangleright}$ directly by the semantics of a shallow embedding in our meta-language $ML_{\mathtt{UTT}}$ introduced in Chapter 2. Recall earlier we stated that step-indexed propositions must satisfy a monotonicity condition. We can leverage our expressive meta-language to enforce that all step-indexed propositions are accompanied by a proof that they are monotone. We denote step-indexed propositions using ϕ and ψ, as in our informal grammar. The collection of such propositions which are monotone is defined in $ML_{\mathtt{UTT}}$ to be `IProp` using a Σ-type as follows.

$$\mathtt{IProp} \triangleq \Sigma \phi : \mathbb{N} \to \mathtt{Set}.\mathtt{monotone}(\phi)$$

where the predicate `monotone` expresses the monotonicity property:

$$\mathtt{monotone}(\phi) \triangleq \Pi n : \mathbb{N}.\, \phi(\mathbf{succ}\, n) \to \phi(n)$$

From now on, we only consider monotone step-indexed propositions, and abusing notation, denote such propositions using ϕ or ψ. Thus, each proposition occurring in the standard connectives of Figure 5.1 is accompanied with a proof that it is monotone.

Our shallow embedding is parameterised by the signature Ω. In $ML_{\mathtt{UTT}}$, we may represent the signature as an *inductive-recursive* definition [Dybjer and Setzer, 1999] consisting of a *universe* of *codes* $\mathcal{U}$ and a *decoding* function, $[\![-]\!] : \mathcal{U} \to \mathtt{Set}$. Given this representation, each sort $\mathcal{S} \in \mathbb{S}$ of our signature Ω arises as the interpretation $[\![u]\!]$ of some universe code $u : \mathcal{U}$. In giving the semantics of $\mathtt{FOL}^{\mu\triangleright}$, we identify the collection of sorts $\mathbb{S}$ as the universe of codes and assume given an appropriate decoding function $[\![-]\!] : \mathbb{S} \to \mathtt{Set}$.

Define the judgement $n \vDash \phi$ as the truth value of the proposition $\phi : \mathtt{IProp}$ at $n : \mathbb{N}$:

$$\begin{aligned}
_ \vDash _ \quad &: \quad \mathbb{N} \to \mathtt{IProp} \to \mathtt{Set} \\
n \vDash \phi \quad &\triangleq \quad \mathsf{proj}_1(\phi)\ n
\end{aligned}$$

We now define our shallow embedding of $\mathtt{FOL}^{\mu\triangleright}$ using `IProp`. We show only the construction of the step-indexed proposition $\mathbb{N} \to \mathtt{Set}$, omitting the accompanying proofs of `monotone` which follow trivially from the components of each connective.

Given $\phi, \psi : \texttt{IProp}$, define conjunction and disjunction as follows.

$$\phi \wedge \psi \triangleq \lambda n.n \vDash \phi \times n \vDash \psi$$

$$\phi \vee \psi \triangleq \lambda n.n \vDash \phi + n \vDash \psi$$

where disjunction uses a meta-level sum type, $+$ with constructors inj_1 and inj_2 which we omit but can be easily defined as an inductive type.

Assume given a map $\phi : [\![\mathcal{S}]\!] \to \texttt{IProp}$, define $\forall$ and $\exists$ quantifiers as follows.

$$\forall x : \mathcal{S}.\ \phi \triangleq \lambda n.\Pi x : [\![\mathcal{S}]\!].\, n \vDash \phi\ x$$

$$\exists x : \mathcal{S}.\ \phi \triangleq \lambda n.\Sigma x : [\![\mathcal{S}]\!].\, n \vDash \phi\ x$$

An interesting case is implication. The semantics we give must satisfy monotonicity, and doing so requires a Kripke-style semantics for implication, ensuring that the implication holds for all 'future worlds', *i.e* all indices strictly less than the current one, as well as the current one. Assume given $\phi, \psi : \texttt{IProp}$,

$$\phi \implies \psi \triangleq \lambda n.\Pi k : \mathbb{N}.\, k \leq n \to k \vDash \phi \to k \vDash \psi$$

Equivalence of two step-indexed propositions $\phi, \psi : \texttt{IProp}$ is built from existing operators:

$$\phi \iff \psi \triangleq (\phi \implies \psi) \wedge (\psi \implies \psi)$$

Assuming $\phi : \texttt{IProp}$, the $\triangleright$ modality is defined by cases on the step-index.

$$\begin{array}{llll} \triangleright & \phi & 0 & \triangleq \top \\ \triangleright & \phi & (\mathbf{succ}\, n) & \triangleq n \vDash \phi \end{array}$$

where we omit the obvious injection of $\top$ into an $\texttt{IProp}$.

Application of relation symbols $\mathcal{A} : \overline{[\![\mathcal{S}]\!]} \to \texttt{Set}$ to terms $\overline{t : [\![\mathcal{S}]\!]}$ gives rise to a trivial step-indexed proposition in which the step-index is ignored. Such an interpretation is trivially monotone. Since the injection of such relations into $\texttt{FOL}^{\mu\triangleright}$ is implicit, we identify the $ML_{\texttt{UTT}}$ application $\mathcal{A}\,\bar{t}$ yielding a $\texttt{Set}$ with the corresponding $\texttt{FOL}^{\mu\triangleright}$ application construct yielding an $\texttt{IProp}$.

5.1.1 Predicates and Contractivity

We now turn to the semantics for $\texttt{FOL}^{\mu\triangleright}$ predicates for which the fixed-point construct is of central importance, being the basis for the definition of ELLA applicative and

logical approximations in Chapter 6. In giving the semantics, we restrict attention to unary predicates $(x : \mathcal{S}).\phi$ and $\mu P : \mathcal{S}.\pi$. The results extend to predicates of arbitrary arity by straightforward (un)currying. A unary predicate π is represented in ML_{UTT} using the following type.

$$\begin{array}{lcll} \mathsf{Pred} : & \mathbb{S} & \to & \mathtt{Set}_1 \\ \mathsf{Pred} & \mathcal{S} & \triangleq & [\![\mathcal{S}]\!] \to \mathtt{IProp} \end{array}$$

We define the trivial predicate which is true for all arguments and steps:

$$\begin{array}{lcl} \mathsf{True}_{\mathcal{S}} & : & \mathsf{Pred}\ \mathcal{S} \\ \mathsf{True}_{\mathcal{S}} & \triangleq & \lambda\_.\top \end{array}$$

where we typically omit the $\mathcal{S}$ subscript and just write True.

The encoding for $(x : \mathcal{S}).\phi$ with $\phi : \mathsf{Pred}\ \mathcal{S}$ is defined as follows.

$$(x : \mathcal{S}).\phi \triangleq \lambda x.\lambda n.n \vDash \phi\ x$$

Earlier, we briefly alluded to an important restriction placed on the occurrences of predicate variables within a fixed-point predicate to ensure the predicate is well-founded. Roughly speaking, this restriction, known as *contractivity*, requires all uses of the predicate variable to occur under a $\triangleright$ operator. The contractivity constraint is sufficient to establish the existence of a fixed-point for a recursive predicate.

Dreyer et al. [2011] give a Kripke model for their logic LSLR which relies on contractivity for interpreting recursive relations. However, in their case contractivity is used to ensure their chosen induction metric is well-founded. While $\mathtt{FOL}^{\mu\triangleright}$ is highly inspired by LSLR, our use of contractivity is much closer to its application in establishing a fixed-point for COFEs by Di Gianantonio and Miculan [2003]. We illuminate how the semantics for $\mathtt{FOL}^{\mu\triangleright}$ predicates arises as a particular instance of the general COFE framework. Our presentation is motivated by a desire to establish a clear connection between (a shallow embedding of) an LSLR-like logic, and the exact structures involved in the construction of a COFE. Indeed, recent work by Biernacki et al. [2018] assume such a connection exists, writing:

> In order to hide indices and give a concise, readable definition we work in the category COFE of complete ordered families of equivalences ... This means that throughout the paper formulas (and relations) are implicitly indexed...

yet their formalisation in Coq is based on the LSLR-inspired IxFree library by Polesiuk [2017], and they do not make the connection explicit. Similar to our presentation

below, Sieczkowski et al. [2015] report on formalising COFEs in Coq but do not show the construction of the unique fixed-point, as we do in Definition 5.11.

We recall the definitions and properties of *ordered families of equivalences* (OFEs) and COFEs due to Di Gianantonio and Miculan [2003] given in standard first-order logic. $\texttt{FOL}^{\mu\triangleright}$ predicates serve as a running illustrative example.

Definition 5.2. An *ordered family of equivalences (OFE)* is a tuple $O = \langle A, <, X, \sim \rangle$ where A (the *carrier*) and X (the *domain*) are sets, $<$ is a well-founded order on A, and $\sim$ is an A-indexed family of equivalence relations $\langle \sim_a \rangle_{a \in A}$ on X.

Example 5.3 ($\texttt{FOL}^{\mu\triangleright}$ predicates form an OFE)**.** The ordered family of equivalences for the set of all $\texttt{FOL}^{\mu\triangleright}$ predicates $\mathsf{Pred}\ \mathcal{S}$ for some $\mathcal{S} : \mathbb{S}$ is:

$$O_{\mathcal{S}} = \langle \mathbb{N}, <, \mathsf{Pred}\ \mathcal{S}, \sim \rangle$$

induced by

$$\pi \sim \pi' \triangleq \forall x : \mathcal{S}. \pi\ x \iff \pi'\ x$$

and

$$\pi \sim_n \pi' \triangleq n \vDash \pi \sim \pi'$$

Our OFE definition above is analogous to an exposition given by Jung et al. [2018b] for step-indexing in an Iris model. Not surprisingly, both accounts agree on the definition of n-equivalence; it must be monotone with respect to step-indices. The accounts differ in regards to the metatheory: we describe an OFE within our meta-language $ML_{\texttt{UTT}}$, whereas Jung et al. opt for a more familiar set theoretic setting.

Definition 5.4. Let $O = \langle A, <, X, \sim \rangle$ be an OFE. Let $I \subseteq A$, and $(x_a)_{a \in I}$, an I-indexed family of elements in X. Such a family $(x_a)_{a \in I}$ is *coherent* if

$$\forall a, b \in I. a < b \implies x_a \sim_a x_b$$

Moreover, $(x_a)_{a \in I}$ has a *limit* y if

$$\forall a \in I \implies x_a \sim_a y$$

A family of elements $(x_a)_{a \in I}$ in X indexed by $I \subseteq A$ can be understood as the following Π-type in $ML_{\texttt{UTT}}$.

$$\mathsf{fam} : \Pi a : A\,.\, a \in I \to X$$

which degenerates to $\mathsf{fam} : A \to X$ when $I = A$.

Example 5.5. By the definition of $\sim$ in Example 5.3, a family of predicates given by $\pi_{\_} : \mathbb{N} \to \mathsf{Pred}\ \mathcal{S}$ is coherent if and only if

$$\forall n, m \in \mathbb{N}, x : \mathcal{S}.\, n < m \to n \vDash \pi_n\ x \iff \pi_m\ x$$

Predicates occurring later in the sequence (*i.e.* at larger step-index) are required to agree with all earlier predicates for an increasing number of indices. Thus, most agreement occurs at lower step-index and suggests that the sequence is decreasing in nature, carving out an ever smaller collection of $\mathcal{S}$ objects which satisfy the next predicate in the sequence. In other words, a coherent family is a sequence, or chain, of predicates which is converging with respect to the equivalence relation of the OFE. These observations accord with the view that increasing the step counter increases our discriminatory power.

Definition 5.6. A *complete ordered family of equivalences* (COFE) is a tuple

$$O = \langle A, <, X, \sim, \lim_{\cdot \in A}, \lim_{\cdot <_{\cdot}} \rangle$$

such that

- $\langle A, <, X, \sim \rangle$ is an OFE;
- $\lim_{\cdot \in A}$ is a function such that for all coherent families $(x_a)_{a \in A}$, $\lim\limits_{a \in A} x_a$ is a limit for $(x_a)_{a \in A}$;
- $\lim_{\cdot <_{\cdot}}$ is a function such that for all $a \in A$ and coherent families $(x_b)_{b \in \downarrow a}$, $\lim\limits_{b < a} x_b$ is a limit for $(x_b)_{b \in \downarrow a}$;

where $\downarrow a \triangleq \{b \in A \mid b < a\}$.

As remarked by Di Gianantonio and Miculan [2003], and seen earlier for fam, the limit constructions can be given a typing in $ML_{\mathtt{UTT}}$:

$$\lim_{\cdot \in A} : (A \to X) \to X \qquad\qquad \lim_{\cdot <_{\cdot}} : \Pi a : A\,.\, (\downarrow a \to X) \to X$$

Example 5.7. Extending the OFE from Example 5.3, define the following limit constructions for coherent families of unary $\mathtt{FOL}^{\mu \triangleright}$ predicates:

$$k \vDash \left(\lim_{n \in \mathbb{N}} \pi_n \right)\ x \triangleq k \vDash \pi_k\ x$$

$$\lim_{n < 0} \pi_n \triangleq \mathsf{True} \qquad\qquad \lim_{n < m+1} \pi_n \triangleq \pi_m$$

Note that for $\lim_{n\in\mathbb{N}} \pi_n$ we have only defined the first component $\mathbb{N} \to \texttt{Set}$ of $\texttt{IProp}$. Monotonicity of the limit follows from coherence and monotonicity of the predicates in the family; we omit the details. We can prove these constructions satisfy the conditions required of a limit. For $\lim_{n\in\mathbb{N}} \pi_n$, we require to show

$$\forall m \in \mathbb{N}. \pi_m \sim_m \left(\lim_{n\in\mathbb{N}} \pi_n\right)$$

By definition, given $k \leq m$ and $x : \mathcal{S}$, it suffices to show

$$k \vDash \pi_m\ x \leftrightarrow k \vDash \pi_k\ x$$

where $P \leftrightarrow Q \triangleq P \to Q \times Q \to P$ in $ML_{\texttt{UTT}}$.

By cases, $k = m$ is immediate and $k < m$ follows from coherence of the family. The check for the other limit construction is similar.

Definition 5.8. Given an OFE $\langle A, <, X, \sim\rangle$ a function $f : X \to X$ is *contractive* if for every $x, y \in X$ and $a \in A$ we have

$$\forall b < a. x \sim_b y \implies f(x) \sim_a f(y)$$

Given approximate inputs, a contractive function produces more approximate outputs. The formal definition essentially characterises the function of the $\triangleright$ modality.

Example 5.9 (Contractivity in $\texttt{FOL}^{\mu\triangleright}$)**.** By definition of the OFE in Example 5.3, a function $f : \mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}$ is contractive, written $\mathsf{contractive}(f)$ if, given predicates $\pi, \pi' : \mathsf{Pred}\ \mathcal{S}$, the following inference is provable for all $n : \mathbb{N}$:

$$\forall k. k < n \to k \vDash \pi \sim \pi' \to n \vDash f\ \pi \sim f\ \pi'$$

using the definition of $\sim$ for the aforementioned OFE.

Expressing the above using the Kripke semantics for $\texttt{FOL}^{\mu\triangleright}$ implication and the $\triangleright$ modality we obtain:

$$\Pi n : \mathbb{N}. n \vDash \triangleright(\pi \sim \pi') \implies (f\ \pi \sim f\ \pi')$$

which embodies the stipulation that occurrences of the predicate variable, standing for π and π' in the left- and right-hand side, respectively, must occur under a later operator.

We now have introduced enough machinery to state Di Gianantonio and Miculan's [2003] main result.

Theorem 5.10. *Let $O = \langle A, <, X, \sim, \lim_{\cdot\in A}, \lim_{\cdot<\cdot}\rangle$ be a COFE and f a contractive function on X. Then, there exists a* generalised fixed point *$x \in X$ with the property that $x \sim f(x)$. For every other generalised fixed point y, we have $x \sim y$.*

We can prove a fixed point result for our COFE instance of $\mathtt{FOL}^{\mu\triangleright}$ predicates. The proof of Theorem 5.10 relies on an A-indexed family of X elements defined by induction on the well-founded order relation $<$ using a limit construction and the contractive function f. Di Gianantonio and Miculan [2003] give this construction as:

$$x_a \triangleq f\left(\lim_{b<a} x_b\right) \qquad \text{(POWLIMIT)}$$

Since, we are working with an explicit A, namely $\mathbb{N}$, this construction essentially amounts to defining a family by induction on $\mathbb{N}$. The inductive case applies the contractive function to a structurally smaller argument. The base case simply returns $\lim_{m<0} \pi_m$ which by Example 5.7 is defined to be True.

Definition 5.11. Define $\mathsf{pow} : (\mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}) \to \mathbb{N} \to \mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}$ to be the $\mathbb{N}$-indexed family of $\mathtt{FOL}^{\mu\triangleright}$ predicates defined by primitive recursion on $\mathbb{N}$:

$$\begin{array}{llll} \mathsf{pow} & f & 0 & P_0 \triangleq P_0 \\ \mathsf{pow} & f & (\mathbf{succ}\, n) & P_0 \triangleq f\ (\mathsf{pow}\ f\ n\ P_0) \end{array}$$

To arrive at a fixed point for the above family, we may utilise the limit construction $\lim_{\cdot\in A}$ provided by the COFE for $\mathtt{FOL}^{\mu\triangleright}$ predicates. First, the essential property we must show is that the family generated by Definition 5.11 is coherent. Abstractly, that is:

$$\forall c < b \in{\downarrow} a.\, f\left(\lim_{e<c} x_e\right) \sim_c f\left(\lim_{e<b} x_e\right)$$

by induction on $<$ for each family $(x_b)_{b\in\downarrow a}$, for all $a \in A$.

Concretely, we require a proof of coherence for the family generated by pow with $P_0 = \mathsf{True}$.

Lemma 5.12. *Given $k \leq n$, $x : \mathcal{S}$, and $f : \mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}$,*

$$k \vDash f\ (\mathsf{pow}\ f\ k\ \mathsf{True})\ x \iff f\ (\mathsf{pow}\ f\ n\ \mathsf{True})\ x$$

Proof. By induction on k. □

Note that the construction of Equation POWLIMIT constructs the sequence:

$$\top, f(\top), f(f(\top)), \ldots, f^n(\top), \ldots$$

which converges to a fixed point by Theorem 5.10. We shall prove the fixed point constructed is a *greatest* fixed point, thus justifying the view of Löb induction as a coinductive proof principle.

We pave the way towards our result by first considering the following theorem for an endofunction on a complete lattice.

Theorem 5.13. *[Sangiorgi, 2011, § 2.8]. Let F be an endofunction on a complete lattice, in which $\bot$ and $\top$ are the bottom and top elements. If F is* cocontinuous*, then*

$$\mathtt{gfp}(F) = \bigsqcap_{n \geq 0} F^n(\top).$$

Theorem 5.13 constructs exactly the same sequence as Equation POWLIMIT. Thus, it suffices to show that contractive functions are *cocontinuous*.

Definition 5.14. [Sangiorgi, 2011, § 2.8]. An endofunction F on a complete lattice is *cocontinuous* if for all sequences $\alpha_0, \alpha_1, \ldots$ of decreasing points in the lattice, we have

$$F(\bigsqcap_i \alpha_i) = \bigsqcap_i F(\alpha_i).$$

From the COFE structure in Example 5.7 we wish to form a complete lattice with respect to some partial order $\leq$ upon which the contractive functions of Example 5.9 are cocontinuous. In particular, fix a contractive function f. We take our lattice to be the elements, $\top, f(\top), f(f(\top)), \ldots$; that is the set $\{f^n(\top) \mid n \in \mathbb{N}\}$ along with the partial order $\leq$ defined by:

$$f^j(\top) \leq f^i(\top) \triangleq i \leq j$$

This lattice is complete: the top element is $\top$, the bottom element is $\lim_{n \in \mathbb{N}} f^n(\top)$, given by the COFE structure, and any two elements have a meet and a join defined by:

$$f^j(\top) \sqcap f^i(\top) \triangleq \begin{cases} f^j(\top) & \text{if } i \leq j \\ f^i(\top) & \text{if } j \leq i \end{cases} \qquad f^j(\top) \sqcup f^i(\top) \triangleq \begin{cases} f^i(\top) & \text{if } i \leq j \\ f^j(\top) & \text{if } j \leq i \end{cases}$$

These definitions compute the meet (resp. join) as the element under the most (resp. least) applications of the contractive function f.

Define $F(\pi) = f(\pi)$ to be an endofunction on the complete lattice above. The final step in concluding Theorem 5.13 is to prove this endofunction is cocontinuous.

Lemma 5.15. *The endofunction F is cocontinuous. That is, for all sequences $\overline{\alpha} = \alpha_0, \alpha_1, \ldots$ of decreasing points in the lattice, we have*

$$F(\bigsqcap_i \alpha_i) = \bigsqcap_i F(\alpha_i).$$

Proof. We prove the result in two parts.

Case $F(\bigsqcap_i \alpha_i) \leq \bigsqcap_i F(\alpha_i)$.

We have $\bigsqcap_i \alpha_i \leq \alpha_i$, for all $\alpha_i \in \overline{\alpha}$, by definition of the meet for a sequence. Therefore, by definition of $\leq$, we have

$$f\left(\bigsqcap_i \alpha_i\right) \leq f(\alpha_i)$$

for all $\alpha_i \in \overline{\alpha}$. By the definition of F, we may conclude

$$F\left(\bigsqcap_i \alpha_i\right) \leq F(\alpha_i)$$

for all $\alpha_i \in \overline{\alpha}$. In other words, $F(\bigsqcap_i \alpha_i)$ is a lower bound for the sequence, $\{F(\alpha_i) \mid \alpha_i \in \overline{\alpha}\}$. By definition of the greatest lower bound, we have

$$F\left(\bigsqcap_i \alpha_i\right) \leq \bigsqcap_i F(\alpha_i)$$

as required.

Case $\bigsqcap_i F(\alpha_i) \leq F(\bigsqcap_i \alpha_i)$.

By definition of the greatest lower bound,

$$\bigsqcap_i F(\alpha_i) \leq F(\alpha_i)$$

for all $\alpha_i \in \overline{\alpha}$. In particular, by the completeness of the lattice and the definition of the meet for a sequence, $\bigsqcap_i \alpha_i$ occurs in the sequence $\overline{\alpha}$. Hence,

$$\bigsqcap_i F(\alpha_i) \leq F\left(\bigsqcap_i \alpha_i\right)$$

as required.

□

Definition 5.16 ($\mathtt{FOL}^{\mu\triangleright}$ Fixed-Point Predicate)**.** Given $\pi : \mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}$ such that $\mathsf{contractive}(\pi)$, define $(\mu P : \mathcal{S}.\pi) : \mathsf{Pred}\ \mathcal{S}$ as follows.

$$\mu P : \mathcal{S}.\pi \triangleq \lambda x.\lambda n.n \vDash \pi\ (\mathsf{pow}\ \pi\ n\ \mathsf{True})\ x.$$

Again, we do not give a proof of monotonicity for this construction, but it follows from contractivity of π and Lemma 5.12.

We complete our description of $\texttt{FOL}^{\mu\triangleright}$ by defining the main reasoning judgement.

Definition 5.17. The main judgement $\Phi \Vdash \psi$ states that under the hypotheses Φ the formula ψ is provable in $\texttt{FOL}^{\mu\triangleright}$. Encoding of this judgement is as follows. First, we define an operation on lists of formulae.

$$\begin{array}{lccl} \bigwedge : & \mathsf{List}\ \texttt{IProp} & \to & \texttt{IProp} \\ \bigwedge & [] & \triangleq & \top \\ \bigwedge & (\phi :: \Phi) & \triangleq & \phi \wedge (\bigwedge \Phi) \end{array}$$

The main judgement unpacks the list of hypotheses Φ into a single $\texttt{FOL}^{\mu\triangleright}$ formula, resulting in the following encoding for the main judgement.

$$\Phi \Vdash \psi \triangleq \Pi n : \mathbb{N} . n \vDash (\bigwedge \Phi) \implies \psi$$

In fact, since we are quantifying over all steps, we can further simplify the definition by using the (non-dependent) function space of $ML_{\mathtt{UTT}}$ for implication:

$$\Phi \Vdash \psi \triangleq \Pi n : \mathbb{N} . n \vDash (\bigwedge \Phi) \to n \vDash \psi$$

5.2 Derivable Inference Rules

We can derive a collection of inference rules which are provable in our model, *i.e.* the shallow embedding. We present a basic set of rules in Figure 5.2 which includes the standard introduction and elimination rules for the logical connectives, familiar from natural deduction presentations of first-order logic. The inference rules can be understood as representing the function space of the meta-language $ML_{\mathtt{UTT}}$. That is, given $X_1, \ldots, X_n, Y : \texttt{Set}$, stating a rule such as:

$$\frac{X_1 \quad X_2 \quad \cdots \quad X_n}{Y}\ \textsc{rule}$$

should be read as witnessing an $ML_{\mathtt{UTT}}$ function $\mathsf{rule} : X_1 \to \cdots \to X_n \to Y$.

A clear advantage of our shallow embedding is the handling of capture-avoiding substitution, α-equivalence, and the treatment of binders and the function space by the meta-language $ML_{\mathtt{UTT}}$. In particular, we do not require inference rules for substitution, *e.g.* in universal quantification elimination, or application to a predicate, because

$$(\Longrightarrow \mathrm{I})\ \frac{\Phi, \phi \Vdash \psi}{\Phi \Vdash \phi \Longrightarrow \psi} \qquad (\Longrightarrow \mathrm{E})\ \frac{\Phi \Vdash \phi \Longrightarrow \psi \quad \Phi \Vdash \phi}{\Phi \Vdash \psi}$$

$$(\wedge \mathrm{I})\ \frac{\Phi \Vdash \phi \quad \Phi \Vdash \psi}{\Phi \Vdash \phi \wedge \psi} \qquad (\wedge_L \mathrm{E})\ \frac{\Phi \Vdash \phi \wedge \psi}{\Phi \Vdash \phi} \qquad (\wedge_R \mathrm{E})\ \frac{\Phi \Vdash \phi \wedge \psi}{\Phi \Vdash \psi}$$

$$(\vee_L \mathrm{I})\ \frac{\Phi \Vdash \phi}{\Phi \Vdash \phi \vee \psi} \qquad (\vee_R \mathrm{I})\ \frac{\Phi \Vdash \psi}{\Phi \Vdash \phi \vee \psi} \qquad (\vee \mathrm{E})\ \frac{\Phi \Vdash \phi_1 \vee \phi_2 \quad \Phi, \phi_1 \Vdash \psi \quad \Phi, \phi_2 \Vdash \psi}{\Phi \Vdash \psi}$$

$$(\triangleright \mathrm{I})\ \frac{\Phi \Vdash \phi}{\Phi \Vdash \triangleright \phi} \qquad (\triangleright \textsc{Löb})\ \frac{\Phi, \triangleright \phi \Vdash \phi}{\Phi \Vdash \phi}$$

$$(\forall \mathrm{I})\ \frac{\Pi x : [\![\mathcal{S}]\!] . \Phi \Vdash \phi\ x}{\Phi \Vdash \forall x : \mathcal{S}.\ \phi}\ (x \text{ a fresh } ML_{\mathrm{UTT}} \text{ variable}) \qquad (\forall \mathrm{E})\ \frac{\Phi \Vdash \forall x : \mathcal{S}.\ \phi}{\Phi \Vdash \phi\ t}\ (t : [\![\mathcal{S}]\!])$$

$$(\exists \mathrm{I})\ \frac{\Phi \Vdash \phi\ t}{\Phi \Vdash \exists x : \mathcal{S}.\phi}\ (t : [\![\mathcal{S}]\!]) \qquad (\exists \mathrm{E})\ \frac{\Phi \Vdash \exists x : \mathcal{S}.\phi}{\Sigma x : [\![\mathcal{S}]\!] . \Phi \Vdash \phi\ x}\ (x \text{ a fresh } ML_{\mathrm{UTT}} \text{ variable})$$

$$\textsc{absPropApp}\ \frac{\Phi \Vdash (\overline{x} : \overline{\mathcal{S}}).\phi}{\Phi \Vdash ((\overline{x} : \overline{\mathcal{S}}).\phi)\ \overline{t}}\ (\overline{t : [\![\mathcal{S}]\!]}) \qquad \textsc{unroll}\ \frac{\Phi \Vdash (\mu P : \overline{S}.\pi)\ \overline{t}}{\Phi \Vdash \pi\ (\mu P : \overline{S}.\pi)\ \overline{t}} \qquad \textsc{roll}\ \frac{\Phi \Vdash \pi\ (\mu P : \overline{S}.\pi)\ \overline{t}}{\Phi \Vdash (\mu P : \overline{S}.\pi)\ \overline{t}}$$

$$(\wedge \mathrm{D})\ \frac{\Phi \Vdash \triangleright(\phi \wedge \psi)}{\Phi \Vdash (\triangleright\phi) \wedge (\triangleright\psi)} \qquad (\vee \mathrm{D})\ \frac{\Phi \Vdash \triangleright(\phi \vee \psi)}{\Phi \Vdash (\triangleright\phi) \vee (\triangleright\psi)} \qquad (\Longrightarrow \mathrm{D})\ \frac{\Phi \Vdash \triangleright(\phi \Longrightarrow \psi)}{\Phi \Vdash (\triangleright\phi) \Longrightarrow (\triangleright\psi)}$$

$$(\forall \mathrm{D})\ \frac{\Phi \Vdash \triangleright(\forall x : \mathcal{S}.\ \phi)}{\Phi \Vdash \forall x : \mathcal{S}. \triangleright \phi}$$

Figure 5.2: A collection of derivable rules for our $\mathtt{FOL}^{\mu\triangleright}$ model

substitution of bound variables for terms is automatic by virtue of our encoding of these constructs as meta-language functions. In contrast, Dreyer et al.'s [2011] inference rules represent a deep embedding of LSLR in an unspecified meta-logic which is subsequently given an interpretation through a step-indexed model.

As mentioned earlier, the $\triangleright$ modality admits the Löb induction principle. Taking Φ to be the empty list in rule ($\triangleright$Löb), we arrive at the rule given in the introduction. Concretely, we can unfold the definition to obtain:

$$\frac{\Pi k : \mathbb{N} \,.\, k \leq n \to k \vDash \triangleright\psi \to k \vDash \phi}{n \vDash \psi}$$

where we see that the Löb rule is indeed an induction principle: well-founded induction over the ordering relation $<$.

The $\triangleright$ modality also abstractly characterises the monotonicity property of step-indexed propositions; the ($\triangleright$I) rule in Figure 5.2. For this reason, when applying this rule, we often refer to it as monotonicity, or simply *mono*.

A key property to establish for our connectives is that they are contractive in the sense of Example 5.9. The majority of these are straightforward, following from the elimination rules of the various connectives. Contractivity of the $\triangleright$ modality relies on its distributivity with respect to implication. More generally, $\triangleright$ distributes over all connectives apart from $\exists$. For example, the rule

$$\frac{\Phi \Vdash \triangleright(\phi \wedge \psi)}{\Phi \Vdash (\triangleright\phi) \wedge (\triangleright\psi)}$$

is derivable in our model, where the double bar indicates the rule is an isomorphism of $ML_{\mathtt{UTT}}$ functions.

Finally, we have rules for predicates: application for abstracted propositions, and the (un)folding rules for our fixed-point predicates. By appealing to our $ML_{\mathtt{UTT}}$ model, these constructs are guaranteed to be well-formed, *e.g.* matching actual and expected argument sequences, in order for the rules to define valid $ML_{\mathtt{UTT}}$ function typings.

The set of rules is not exhaustive; they do not constitute a definition of $\mathtt{FOL}^{\mu\triangleright}$. One benefit of a shallow embedding over a deep embedding is the ease of extending the system with further rules, defined directly by the semantics we presented in the previous section.

5.3 Discussion

Our AGDA formalisation is heavily inspired by the IXFREE library developed by Polesiuk [2017]. IXFREE is based on the LSLR logic and is similar to other libraries, *e.g.* ModuRes [Sieczkowski et al., 2015] and Appel et al.'s [2007] accompanying COQ library. IXFREE differs from ModuRes by fixing the carrier A to be $\mathbb{N}$ whereas ModuRes formalises the more general COFE structures, supporting the modelling of more complex calculi involving higher-order state and concurrency. The development by Appel et al. is similar to IXFREE, stratifying judgements by natural numbers characterising the finite approximations. The former, however, has a slightly different aim: they seek to establish safety properties of programs with respect to a semantics, rather than establishing logical relations for program equivalence.

Porting the library to AGDA was somewhat non-trivial due to the different support AGDA and COQ provide for theorem proving. AGDA theorem proving consists of constructing a dependently-typed function which realises the statement of the theorem. In contrast, COQ supports proof construction by tactic scripts using an untyped tactic language called LTAC [Delahaye, 2000]. User-defined LTAC scripts support pattern matching on hypotheses in the context and on current proof obligations, allowing straightforward discharging of step-index arithmetic without the user of the library ever having to see it.

It is more challenging to achieve similar abstract behaviour in AGDA. The shallow embedding hides the step-indexing explicit in logical connectives but fails to eliminate all step-indexing arithmetic. In particular, reasoning about $\mathtt{FOL}^{\mu\triangleright}$ implications at index n, say, usually requires introduction of the Kripke assumption of a future step-index k such that $k \leq n$. These appear as additional patterns to the AGDA function representing the theorem statement. While some cases would benefit from the more abstract treatment, *e.g.* cases which introduce the constraint only to apply it immediately to a subgoal, we argue that many cases are more readable as a result since the extra assumptions can be included in spelling out intermediate proof obligations.

While we gain much flexibility by our choice of encoding, our shallow embedding has some disadvantages. Firstly, as a well-known drawback of shallow embeddings, formulae cannot be easily transformed into an equivalent form. Secondly, AGDA has trouble inferring the step-index and proposition of a supplied judgement $n \vDash \phi$, making appeals to lemmas extremely verbose for concrete propositions ϕ that are even just a few nested connectives in size. We have found packaging the judgement with AGDA's

record syntax assists the AGDA type checker in inferring the details. Separately, we investigated a form of *reification* and *reflection* to address these issues. We define a deep HOAS embedding of $\texttt{FOL}^{\mu\triangleright}$ which *reifies* the shallow `IProp` representation as a syntax tree. We are able to define recursive functions over the syntax to perform transformations of formulae. These functions can be invoked during proofs to simplify goals, in a certain sense emulating a restricted form of type-safe tactic-based proving. Once the transformation is performed, we *reflect* the syntactic representation back to an `IProp` to complete the proof. A concrete example of such a transformation is a procedure to push the $\triangleright$ modality as far into a formula as possible by repeatedly appealing to the distributivity laws. An analogous LTAC tactic has been defined for the users of the IXFREE library. The procedure is used throughout the development, including in proofs of contractivity to reduce the formula to a form for which the contractivity laws of each connective easily applies. Our particular approach to reification and reflection was inspired by the category theory library developed by McBride [2018] in AGDA.

Currently, we have formalised as far as the fundamental theorem for an extension of $\lambda^{\rightarrow}_{FG}$ with recursive functions. Our development is essentially a partial formalisation of a modal logic version of Pitts's [2010] tutorial which presents a step-indexed account of frame stack termination. The main difference between the two being Pitts' termination relation is augmented with step counter whereas the reduction relation in our AGDA development is an atomic relation provided by the instantiated signature Ω, allowing us to more effectively hide step indices occurring in theorem statements.

5.4 Related Work

We have already mentioned our main influences: LSLR by Dreyer et al. [2011] and COFE by Di Gianantonio and Miculan [2003]. It is worth noting Appel et al. [2007] and Nakano [2001], two papers that popularised the use of Kripke modal logics with $\triangleright$ for approximating self-references. Indeed, these two works were the original inspiration for the LSLR logic. Regarding Nakano [2001], two typing systems are built upon a modal logic for approximating self-references in types, *i.e.* arbitrary recursive types. The semantics is essentially the same as Dreyer et al.'s Kripke model for LSLR. Appel et al. [2007] apply the modal logic to the interpretation of typed assembly languages in the context of proof-carrying code. Using their model they prove semantically that well-typed programs are safe to execute. They provide a formalisation of a shallow embedding of their logic in COQ.

The IxFree library has been used by others for formalising equivalence results for algebraic effects and handlers [Biernacki et al., 2018; Zhang and Myers, 2019]. In particular, Biernacki et al. develop a sound logical relation for a core calculus and prove a number of program equivalences. Building on their formalisation, Zhang and Myers develop a logical relations model for a calculus which ensures effect encapsulation by resolving effect operations according to the lexical scoping of handlers.

Our syntax for our logic splits propositions and predicates into distinct phrase classes. In contrast, Dreyer et al. [2011] define a general "relations" grammar whose well-formedness includes the arity for the relation. Our syntax draws inspiration from the first-order logic by Pretnar [2010], for the a-calculus, supporting reasoning about algebraic effects and handlers. Pretnar's logic interprets (least and greatest) fixed-point predicates using an extension of the Knaster-Tarski theorem.

Basold [2018] develops a similar logic to $\texttt{FOL}^{\mu\triangleright}$, called $\mathbf{FOL}_{\blacktriangleright}$, focussed on observational equivalence for mixed inductive-coinductive programs. Basold presents a Kripke-style model similar to our concrete model in Agda, where Kripke worlds are represented using natural numbers corresponding to approximations. $\mathbf{FOL}_{\blacktriangleright}$ is sound for observational equivalence but not complete; it is unable to validate certain observational equivalences involving inductive arguments over coinductive objects (see [Basold, 2018, Example 5.2.32]). On the other hand, our $\texttt{FOL}^{\mu\triangleright}$ logic is independent of the questions of soundness or completeness for observational equivalence. In Chapter 6, we derive the triangulation result for Ella by *encoding* the relevant relations within $\texttt{FOL}^{\mu\triangleright}$. However, $\texttt{FOL}^{\mu\triangleright}$ is more limited than $\mathbf{FOL}_{\blacktriangleright}$ in the sense that use of the later modality is restricted to the construction of predicates, and cannot be used to construct or reason about coinductive objects, *e.g.* infinite streams.

Chapter 6

Ella

In this chapter, we characterise observational equivalence for ELLA, an effectful programming language, using the *triangulation* proof technique described in Chapter 4.

We present a core monomorphic calculus, ELLA, supporting *binary* handlers which we claim captures the essence of the full FRANK language. Its syntax and type system are naturally inspired by FRANK and our fine-grained CBV development in Chapter 4.

We extend the frame stack development of Section 4.3 to *handler* stacks in order to incorporate effect handling and the forwarding of commands to their nearest dynamically enclosing effect handler.

Following the triangulation method, we define a notion of program approximation based on concrete contexts, as extensions of the term grammar for ELLA. Next, we define logical and applicative approximations within $\mathtt{FOL}^{\mu\triangleright}$ for structural reasoning about behaviour, and prove that they coincide with the observational notion. From these structural notions, we derive useful lemmas for reasoning about concrete program approximations.

The chapter concludes by discussing our main influences from the literature, and situates our work on ELLA in the wider context of program equivalence for effectful calculi.

6.1 Syntax and Semantics

ELLA is a call-by-value monomorphic language with a type- and-effect system. The language has first-class support for effects and their handlers. ELLA captures most of the features supported by FRANK [Lindley et al., 2017] except from parametrised datatypes (and interfaces), and value and effect polymorphism. Adding these features

to ELLA would be straightforward but they would obscure the key ideas presented here so we omit them. In particular, Biernacki et al. [2018] have developed a sound logical relation for a effect-polymorphic core calculus which we discuss further in Secion 6.5. Extensions for value polymorphism and recursive types could follow the development for F^{μ} by Dreyer et al. [2011], due to ELLA's clean separation between value and effect types.

The syntax for ELLA has taken inspiration from the developments of Chapter 4 and differs from FRANK's abstract syntax given by Convent et al. [2020]. In particular, ELLA is expressed in an A-normal/fine-grained form [Levy, 2004; Pitts, 2005]. One could imagine a translation on FRANK programs, turning them into ELLA programs by explicitly binding intermediate terms, and instantiating all polymorphism.

We recall our notational convention from Chapter 4: we denote equalities that hold definitionally by = and $\triangleq$, and propositional equalities by $\equiv$.

6.1.1 Syntax

Figure 6.1 gives the abstract syntax for ELLA types, terms and environments. Unsurprisingly, the types of ELLA are separated into value types and computation types. The value types are the base types (boolean, integer or unit) and suspended computation types, $\{C\}$. An ELLA computation type takes the form:

$$C = \overline{\langle\Delta\rangle A} \to [\Sigma]B \quad \text{where } |\overline{\langle\Delta\rangle A}| = 2$$

where operators have exactly two arguments, $(\langle\Delta_i\rangle A_i)_{i=1,2}$, each specifying a distinct adjustment Δ_i and value type, A_i. The result type specifies the computation may perform effects in the ability Σ — to be handled by the environment — and returns a value of type B.

Restricting our formalism to computation types of two arguments is sufficient to express the example approximations we consider in Chapter 7, including a variant of the `pipe` multihandler first shown in Section 3.9. Of course, we can simulate unary handlers and functions by setting Δ_2 to the identity adjustment, ι, and $A_2 = \mathbf{unit}$. For argument sequences larger than two, we could consider encoding them using binary operators similar to Lindley et al.'s [2017] translation of n-ary operators into unary handlers and case splits, or generalise the formalism presented in this chapter to handle operators of arbitrary finite arity. We do not envisage any complications arising from such an extension of our results.

Types

$$
\begin{array}{lrl}
\text{(value types)} & A, B : \texttt{Ty} & ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} \mid \{C\} \\
\text{(computation types)} & C & ::= T, T \to G \\
\text{(argument types)} & T & ::= \langle \Delta \rangle A \\
\text{(result types)} & G & ::= [\Sigma] A \\
\text{(interfaces)} & I & ::= \{c_i : A_i \to B_i\}_i \\
\text{(abilities)} & \Sigma : \texttt{Ab} & ::= \varnothing \mid \Sigma, I \\
\text{(adjustments)} & \Delta : \texttt{Adj} & ::= \mathfrak{i} \mid \Delta, I \\
\text{(type environments)} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}
$$

Terms

$$
\begin{array}{lrl}
\text{(booleans)} & b & ::= \mathbf{tt} \mid \mathbf{ff} \\
\text{(commands)} & c & \\
\text{(integers)} & k & \in \mathbb{Z} \\
\text{(term variables)} & x, y, z, f & \\
\text{(values)} & v, w : \texttt{Val} & ::= x \mid b \mid k \mid () \mid (\mathbf{rec}\, f\{e\} : \{C\}) \\
\text{(uses)} & m : \texttt{Trm}^{\uparrow} & ::= v \mid v\, n_1\, n_2 \\
\text{(constructions)} & n : \texttt{Trm}^{\downarrow} & ::= \downarrow m \mid H \star c\, v \mid \mathbf{let}\, x : A = n\, \mathbf{in}\, n' \\
\text{(frames)} & F : \texttt{Frm} & ::= (\mathbf{rec}\, f\{e\} : \{C\}, u\, \square) \mid (\mathbf{rec}\, f\{e\} : \{C\}, \square\, n) \mid (x : A.n) \\
\text{(handler stacks)} & H : \texttt{Stk} & ::= Id \mid H \circ F \\
\text{(normal forms)} & u : \texttt{NF} & ::= \downarrow v \mid H \star c\, v \\
\text{(computations)} & e & ::= \overline{\overline{r} \mapsto n} \\
\text{(computation patterns)} & r & ::= p \mid \langle c\, p \to z \rangle \mid \langle x \rangle \\
\text{(value patterns)} & p & ::= b \mid k \mid () \mid x
\end{array}
$$

Figure 6.1: ELLA Syntax

Thus, for the computation type $\overline{\langle\Delta\rangle A} \to [\Sigma]B$, we always have $|\overline{\langle\Delta\rangle A}| = 2$. Generally, for other sequences (*e.g.* pattern clauses) we adopt the notation $\overline{\alpha}$ to specify a sequence of αs of arbitrary finite length.

We briefly describe the various components of ELLA's effect system. We introduced them previously (Section 3.3) in the more general FRANK setting. Commands are specified by a symbol or name, paired with their type signature, *e.g.* $c : A \to B$. For simplicity, commands are unary as opposed to FRANK's n-ary commands. Interfaces, ranged over by I, describe an effect by its primary sources, *i.e.* the collection of commands which give rise to the effect. An ability Σ represents the effects permitted by the context which is either ($\varnothing$) representing purity or a non-empty collection of interfaces. An adjustment Δ is a collection of interfaces.

Definition 6.1. We write $\Delta(\Sigma)$ for the *action of an adjustment* on an ability:

$$\iota(\Sigma) = \Sigma \qquad\qquad (\Delta, I)(\Sigma) = \Delta(\Sigma), I$$

In ELLA, we give a list-like syntax for interpretation of adjustments and abilities. Potential duplicates do not play a significant role in ELLA because we do not have adaptors, parameterised interfaces nor effect polymorphism. So any duplicate instances are in fact identical. That is, `State Int` and `State Bool` are not represented as the same interface (`State`) instantiated to different arguments, `Int` and `Bool`, respectively. Instead, we would represent the two state effects as *different* interfaces, `IState` and `BState`, say, with their own distinct collection of commands. Despite this, we choose to stick with an interpretation which allows duplicates because it is simpler than preventing them with a set-based semantics, and it leaves open the door to subsequent extensions.

Following the design of FRANK, ELLA has a bidirectional type system (Figure 6.2) which distinguishes terms whose types are inferred (*uses*), and terms whose types are checked (*constructions*). Following our $\lambda^{\rightarrow}_{FG}$ formalism, all values and terms are implicitly well-typed and-scoped according to the (bidirectional) type system.

For uses, note that we have an implicit injection of values (whose type may always be inferred). The use *values* consist of the unit value $()$, booleans $b \in \{\mathbf{tt}, \mathbf{ff}\}$, integers $k \in \mathbb{Z}$, and x ranging over countably infinite sets of variable names. In Figure 6.2, we give only one representative rule for base types (T-BOOL), the others are completely standard. The final value form is type-annotated recursive computations, $\mathbf{rec}\, f\{e\} : \{C\}$, which embody functions and effect handlers. In bidirectional systems,

$\boxed{\Gamma\,[\Sigma]\text{–}\ m \Rightarrow A}$

T-VAR
$$\frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\text{–}\ x \Rightarrow A}$$

T-BOOL
$$\frac{b \in \{\mathbf{tt}, \mathbf{ff}\}}{\Gamma\,[\Sigma]\text{–}\ b \Rightarrow \mathbf{bool}}$$

T-THUNK
$$\frac{\Gamma, f : \{C\} \vdash e : C}{\Gamma\,[\Sigma]\text{–}\ (\mathbf{rec}\ f\{e\} : \{C\}) \Rightarrow \{C\}}$$

T-APP
$$\frac{\Gamma\,[\Sigma]\text{–}\ v \Rightarrow \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\} \qquad (\Delta_i(\Sigma) = \Sigma'_i)_{i=1,2} \qquad (\Gamma\,[\Sigma'_i]\text{–}\ n_i : A_i)_{i=1,2}}{\Gamma\,[\Sigma]\text{–}\ v\ \overline{n} \Rightarrow B}$$

$\boxed{\Gamma\,[\Sigma]\text{–}\ n : A}$

T-SWITCH
$$\frac{\Gamma\,[\Sigma]\text{–}\ m \Rightarrow A \qquad A = B}{\Gamma\,[\Sigma]\text{–}\ \downarrow m : B}$$

T-COMMAND
$$\frac{c : A \to B \in \Sigma \quad \Gamma\,[\Sigma]\text{–}\ H : [\Sigma']B \multimap B' \quad c\text{-stuck}(H) \quad \Gamma\,[\Sigma']\text{–}\ v \Rightarrow A}{\Gamma\,[\Sigma]\text{–}\ H \star c\ v : B'}$$

T-LET
$$\frac{\Gamma\,[\Sigma]\text{–}\ n : A \qquad \Gamma, x : A\,[\Sigma]\text{–}\ n' : B}{\Gamma\,[\Sigma]\text{–}\ \mathbf{let}\ x : A = n\ \mathbf{in}\ n' : B}$$

$\boxed{\Gamma \vdash e : C}$

T-COMP
$$\frac{\begin{array}{l}(r_{i,j} : T_j \dashv [\Sigma]\ \Gamma'_{i,j})_{i,j} \\ (\Gamma, (\Gamma'_{i,j})_j\,[\Sigma]\text{–}\ n_i : B)_i \\ (r_{i,j})_{i,j}\ \text{covers}\ (T_j)_j\end{array}}{\Gamma \vdash \overline{\overline{r} \mapsto n} : \overline{T} \to [\Sigma]B}$$

$\boxed{p : A \dashv \Gamma}$

P-VAR
$$\frac{}{x : A \dashv x : A}$$

P-BOOL
$$\frac{}{b : \mathbf{bool} \dashv \cdot}$$

$\boxed{r : T \dashv [\Sigma]\ \Gamma}$

P-VALUE
$$\frac{p : A \dashv \Gamma}{p : \langle\Delta\rangle A \dashv [\Sigma]\ \Gamma}$$

P-REQUEST
$$\frac{c : A \to B \in \Delta(\varnothing) \qquad p : A \dashv \Gamma}{\langle c\ p \to z\rangle : \langle\Delta\rangle B' \dashv [\Sigma]\ \Gamma, z : \{\langle\iota\rangle B, \langle\iota\rangle\mathbf{unit} \to [\Delta(\Sigma)]B'\}}$$

P-CATCHALL
$$\frac{}{\langle x\rangle : \langle\Delta\rangle A \dashv [\Sigma]\ x : \{\overline{\langle\iota\rangle\mathbf{unit}} \to [\Delta(\Sigma)]A\}}$$

Figure 6.2: ELLA Bidirectional Typing Rules

a beta reduction is always explicitly annotated with the type of the abstraction. The remaining *use*, or inferrable term, is application $v\ n_1\ n_2$, which we explain shortly.

Constructions (checked terms) are ranged over by n and consist of let bindings for sequencing, an explicit injection $\downarrow m$ of a use to a construction, and *stuck terms* generalising command invocations. Stuck terms along with values form a subset of constructions called *normal forms*, ranged over by u. A stuck term is of the form $H \star c\ w$, for some command $c : A \to B$, value w of type A and *handler stack* H subject to a "stuck" constraint we introduce shortly.

Handler stacks generalise the frame stacks we saw in Chapter 4. Their grammar is presented in Figure 6.1. Id represents the empty stack and $H \circ F$ is concatenation of a single *frame* F onto the stack H.

There are three distinct kinds of frame: a sequencing frame, $(x : A.n)$, an operator frame, $(\mathbf{rec}\ f\{e\} : \{C\}, \square\ n)$, which pairs an operator with its second argument, and another operator frame, $(\mathbf{rec}\ f\{e\} : \{C\}, u\ \square)$, which pairs an operator with its first argument reduced to a normal form. The intuition behind the two operator frames is that during reduction of an application term, we reduce the arguments in order from left-to-right until they reach a normal form. We store the unevaluated second argument in the frame while we reduce the first argument. Once the first argument reaches a normal form, we store it in the frame and focus reduction on the second argument.

The typing rules for handler stacks appear in Figure 6.3. By the T-ID rule, a handler stack may appear in a context with any typing environment or ability. Sequencing (T-SEQ) does not alter the collection of effects supported by a stack. The rules for operator frames modify the ability permitted at the argument type according to the type signature of the operator. For example, T-HANDLE-FST rule constructs a handler stack expecting an argument term of type A_1, which may perform effects occurring in the ability Σ', or effects which are handled by the first argument of the operator e. Note that for T-HANDLE-SND, we enforce that the first argument is a normal form for the operator e by using the Σ-normal() judgement. We usually omit the ability annotation on $\circ$ for brevity.

The "stuck" constraint we alluded to earlier is captured by the judgement c-stuck(H) defined by the inference rules in Figure 6.4. Intuitively, a frame is c-stuck if it does not handle the command c.

ELLA adopts a similar fine-grained approach to syntax witnessed in $\lambda^{\rightarrow}_{FG}$. The one exception is application (T-APP) which is generalised to account for effect handlers. Like FRANK, there is no special syntax for effect handling in ELLA. Instead,

$$\text{T-ID}\ \frac{}{\Gamma\,[\Sigma]\text{–}\ Id_\Sigma : [\Sigma]A \multimap A} \qquad \text{T-SEQ}\ \frac{\Gamma\,[\Sigma]\text{–}\ H : [\Sigma']B \multimap B' \qquad \Gamma, x : A\,[\Sigma']\text{–}\ n : B}{\Gamma\,[\Sigma]\text{–}\ H \circ_{\Sigma'} (x : A.n) : [\Sigma']A \multimap B'}$$

$$\text{T-HANDLE-FST}\ \frac{\Gamma\,[\Sigma]\text{–}\ H : [\Sigma']B \multimap B' \qquad C = \overline{\langle\Delta\rangle A} \to [\Sigma']B \qquad (\Delta_i(\Sigma') = \Sigma'_i)_{i=1,2} \qquad \Gamma, f : \{C\} \vdash e : C \qquad \Gamma\,[\Sigma'_2]\text{–}\ n : A_2}{\Gamma\,[\Sigma]\text{–}\ H \circ_{\Sigma'_1} (\mathbf{rec}\ f\{e\} : \{C\}, \Box\ n) : [\Sigma'_1]A_1 \multimap B'}$$

$$\text{T-HANDLE-SND}\ \frac{\Gamma\,[\Sigma]\text{–}\ H : [\Sigma']B \multimap B' \qquad C = \overline{\langle\Delta\rangle A} \to [\Sigma']B \qquad (\Delta_i(\Sigma') = \Sigma'_i)_{i=1,2} \qquad \Gamma, f : \{C\} \vdash e : C \qquad \Delta_1(\varnothing)\text{-normal}(u) \qquad \Gamma\,[\Sigma'_1]\text{–}\ u : A_1}{\Gamma\,[\Sigma]\text{–}\ H \circ_{\Sigma'_2} (\mathbf{rec}\ f\{e\} : \{C\}, u\ \Box) : [\Sigma'_2]A_2 \multimap B'}$$

$$\frac{}{\Sigma\text{-normal}(\downarrow w)} \qquad \frac{c : A \to B \in \Sigma}{\Sigma\text{-normal}(H \star c\ w)}$$

Figure 6.3: ELLA handler stack typing rules

$$\frac{c : A \to B \notin \Delta_1(\varnothing)}{c\text{-stuck}((\mathbf{rec}\ f\{e\} : \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\}, \Box\ n))}$$

$$\frac{c : A \to B \notin \Delta_2(\varnothing)}{c\text{-stuck}((\mathbf{rec}\ f\{e\} : \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\}, u\ \Box))} \qquad \frac{}{c\text{-stuck}((x : A.n))}$$

$$\frac{}{c\text{-stuck}(Id)} \qquad \frac{c\text{-stuck}(H) \quad c\text{-stuck}(F)}{c\text{-stuck}(H \circ F)}$$

Figure 6.4: ELLA stuck frames and handler stacks

Given $f : \{C\} \vdash e : C$ and $\Gamma \,[\Sigma]\!\!-\, v \Rightarrow \{\langle\Delta\rangle A, \langle\iota\rangle\mathbf{unit} \to [\Sigma]B\}$,

$$\mathbf{unfold}(e) \triangleq e[(\mathbf{rec}\ f\{e\} : \{C\})/f]$$

$$\uparrow^C e \triangleq \mathbf{rec}\ f\{e\} : \{C\}$$

$$c\ v \triangleq Id \star c\ v$$

$$v\ n_1 \triangleq v\ n_1\ ()$$

Figure 6.5: ELLA syntactic sugar

operators generalise functions, capturing both pure functions and effect handlers. A corresponding generalisation of function application means operators are applied to arbitrary terms which evaluate to normal forms. Consider the typing rule for application in Figure 6.2. Effects are pushed inward at application sites: arguments may perform all the effects allowed by the ambient ability, Σ, plus all the effects, Δ, handled by the operator being applied.

In Figure 6.5, we present some convenient syntactic sugar which we employ throughout this chapter. For $\uparrow^C e$, we sometimes omit the type ascription C when it is clear from the context. For operators where the second argument type is trivial, *i.e.* $\langle\iota\rangle\mathbf{unit}$, we permit omission of the second argument during application which is syntactic sugar for application by the unit value, $()$.

Operators are defined by computations (rule T-COMP) which are a list of clauses each containing patterns and a term body. For a computation to be well-typed it is necessary that the collection of patterns completely covers the argument type(s). We do not present a coverage checking algorithm in this dissertation. We refer the reader to Lindley et al. [2017] for a pattern matching compilation algorithm for FRANK which can be used to check coverage. We assume a fixed collection of built-in operators representing arithmetic operations: $\{+, -, \geq, \leq, ==, \ldots\}$. We omit presenting their completely standard semantics.

Pattern matching has two judgement forms: value pattern matching, $p : A \dashv \Gamma$ and computation pattern matching $r : T \dashv[\Sigma]\ \Gamma$. For value pattern matching, we match on some identifier x or a concrete base value, e.g. a boolean $b \in \{\mathbf{tt}, \mathbf{ff}\}$, (here again, we show only one rule for base types). For computation pattern matching, we either match against a value pattern, a catch-all pattern or a *request* pattern. A request pattern specifies the handling of a command c provided it occurs in the adjustment Δ for that

argument and the value pattern p matches the argument type of c. The variable z is bound to the rest of the computation, the *continuation*.

We define a notion of simultaneous substitution inspired by Definition 4.1.

Definition 6.2 (Simultaneous substitutions)**.** For typing environments Γ_1 and Γ_2 let $\theta : \Gamma_1 \vDash \Gamma_2$ denote a simultaneous substitution θ of variables in environment Γ_1 by values in environment Γ_2. Informally, θ takes Γ_1-terms to Γ_2-terms: given $\Gamma\,[\Sigma]\text{–}\; n : A$ we may apply the substitution to n and obtain $\Gamma_2\,[\Sigma]\text{–}\; \theta(n) : A$. The substitution of a value v for a variable x is denoted $[v/x]$. We write $[\overline{v}/\overline{x}]$ for the simultaneous substitution, substituting each v_i for x_i. For the special case of $\theta : \Gamma \vDash \cdot$ we say that θ is a Γ-*closing* substitution.

Define the left action over a handler stack, @, to be the operation that produces a term given a handler stack and a closed term to fill its hole:

$$\begin{array}{rcl} Id@n & = & n \\ (H \circ (x : A.n'))@n & = & H@(\mathbf{let}\, x : A = n\, \mathbf{in}\, n') \\ (H \circ (\mathbf{rec}\, f\{e\} : \{C\}, \Box\, n'))@n & = & H@{\downarrow}(\uparrow e\; n\; n') \\ (H \circ (\mathbf{rec}\, f\{e\} : \{C\}, u\, \Box))@n & = & H@{\downarrow}(\uparrow e\; u\; n) \end{array}$$

Given a frame F and a handler stack H, define the operation $F \bullet H$ by primitive recursion on H, and a further case analysis on F when $H = Id$:

$$\begin{array}{rcll} (x : A.n) \bullet Id_\Sigma & \triangleq & Id_\Sigma \circ (x : A.n) & \\ (\uparrow^C e, \Box n) \bullet Id_{\Delta_1(\Sigma)} & \triangleq & Id_\Sigma \circ (\uparrow^C e, \Box n) & \text{if } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B \\ (\uparrow^C e, u\Box) \bullet Id_{\Delta_2(\Sigma)} & \triangleq & Id_\Sigma \circ (\uparrow^C e, u\Box) & \text{if } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B \\ F \bullet (H \circ F') & \triangleq & (F \bullet H) \circ F' & \end{array}$$

We abuse notation by writing the lifting of $H \circ F$ operating on frames F to an operation on stacks H' as $H \circ H'$ defined by:

$$\begin{array}{rcl} H \circ Id & \triangleq & H \\ H \circ (H' \circ F) & \triangleq & (H \circ H') \circ F \end{array}$$

Similarly, we lift the $- \bullet -$ operation:

$$\begin{array}{rcl} Id \bullet H & \triangleq & H \\ (H' \circ F) \bullet H & \triangleq & H' \bullet (F \bullet H) \end{array}$$

Having both an append and a prepend operation on stacks turns out to be useful for proving results about observable behaviour by induction on one of the arguments. Of course, for any two stacks both operations produce the same result.

Lemma 6.3. $H \circ H' \equiv H \bullet H'$.

Proof. By induction on H using $Id \circ H \equiv H$ and $F \bullet H \equiv (Id \circ F) \circ H$. Both these subsidiary equalities are proven by induction on H. □

6.1.2 Semantics

We present evaluation for ELLA in terms of handler stacks in Figure 6.6. Evaluation is defined by a termination relation operating on *configurations*, pairs of a stack H and a focussed term n written $\langle H, n \rangle$. Configurations are ranged over by $\mathcal{C}$ and we consider only well-typed configurations (see rule CONFIG in Figure 6.6). The termination relation, $- \downarrow -$, is much like a standard termination relation for configurations [Pitts and Stark, 1998]. To improve readability, we omit explicit 'use-to-construction' injections, $\downarrow$, for result values, as well as typing annotations which are uninformative. We silently apply these conveniences to the rest of our development. Evaluation for ELLA is deterministic; the proof is a straightforward induction over the rules in Figure 6.6.

- E-IDVALUE. The base case for a focussed value terminating in the empty stack context;
- E-LET pushes a sequencing frame onto the stack containing the let body and continues evaluating the bound term;
- E-LETVALUE pops a sequencing frame off the top of the stack and evaluates the body after substituting the value for the **let**-bound variable;
- E-APP pushes an operator frame onto the stack containing the *second* argument and continues evaluating the *first* argument;
- E-APPV-FST stores the value for the first argument in the frame and begins evaluation of the second argument;
- E-APPV-SND pattern matches the value arguments against the operator and continues with the corresponding body of the matching clause, if the match is successful.

The remaining evaluation rules consider the cases when the focussed term is a stuck normal form invoking some command c. The rules search for the nearest dynamically enclosing handler for c by peeling frames off the stack (E-FORWARDCMD) until the

handling operator is found (E-HDLCMD-FST or E-HDLCMD-SND). If we reach the end of the stack without handling the command (E-IDCMD) then the invoked command must be supported by the ambient ability.

The behaviour of the E-APP rule accords with the left-to-right evaluation order of ELLA (and FRANK). The rules E-APPV-SND and E-HDLCMD-SND rely on an auxiliary judgement for pattern matching which computes the result of the application.

The pattern matching judgement (Figure 6.7) computes a substitution from the matching clause and applies it to the body of the clause. Since the type system ensures clauses in a computation cover the argument type (the coverage condition in T-COMP), we need not consider the case of no matching clause. For value variable patterns, M-VAR, the appropriate value substitution is computed. To match a command (M-COMMAND) or catch-all pattern (M-CATCHALL-VAL or M-CATCHALL-REQ), a suspended computation, representing the continuation, is created from the argument term. By coverage, every well-typed sequence $\overline{u}$ of arguments applied to a well-typed computation always yields a matching clause using the rules in Figure 6.7. Since we have not formally defined coverage, we state the property formally below without proof.

Proposition 6.4. *If* $\cdot \vdash e : C$ *with* $C = \overline{\langle\Delta\rangle A} \to [\Sigma]B$ *and* $\cdot \, [\Sigma_i']\text{-} \; u_i : A_i$ *with* $\Sigma_i' = \Delta_i(\Sigma)$ *such that* $\Delta_i(\varnothing)$-normal(u_i) *then there exists an* n *such that* $e : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u} \dashv [\Sigma] \; n$.

Given the structurally inductive character of handler stacks, the rules in Figure 6.6 can be straightforwardly translated to give a small-step semantics which we present in Figure 6.8.

Clearly the small-step semantics respects termination, we omit the details. We refer to each rule using the prefix 'S-' followed by the corresponding suffix used in the rules for Figure 6.6, *e.g.* E-APP becomes S-APP, E-LET becomes S-LET, *etc.* We stratify the small-step semantics by whether the reduction step is an administrative step, $\longrightarrow_\alpha$, or a beta step, $\longrightarrow_\beta$. Intuitively, a step is administrative if it does not change the term represented by the configuration. In other words, we have

$$\langle H, n\rangle \longrightarrow_\alpha \langle H', n'\rangle \text{ if and only if } H@n \equiv H'@n'.$$

Define $\longrightarrow_\alpha^*$ to be the reflexive-transitive closure of $\longrightarrow_\alpha$, and $\mathcal{C} \longrightarrow \mathcal{C}'$ to be shorthand for $\mathcal{C} \longrightarrow_\alpha^* \longrightarrow_\beta \mathcal{C}'$.

The next lemma establishes a progress result for well-typed configurations. Such configurations are either normal — a value or a stuck term paired with an empty stack — or reduce by exactly one rule in Figure 6.8.

Well-typedness

$$\textsc{Config}\ \frac{\Gamma\,[\Sigma]\text{–}\ H : [\Sigma']A \multimap B \qquad \Gamma\,[\Sigma']\text{–}\ n : A}{\Gamma\,[\Sigma]\text{–}\ \langle H, n\rangle : B} \qquad \textsc{Term}\ \frac{\cdot\,[\Sigma]\text{–}\ C : A \qquad \cdot\,[\Sigma]\text{–}\ u : A}{\cdot \vdash C \downarrow_{[\Sigma]A} u}$$

Semantics

$\boxed{\langle H, n\rangle \downarrow_{[\Sigma]A} u}$

$$\textsc{E-IdValue}\ \frac{\cdot\,[\Sigma]\text{–}\ v \Rightarrow A}{\langle Id, \downarrow v\rangle \downarrow_{[\Sigma]A} v} \qquad \textsc{E-Let}\ \frac{\langle H \circ (x : A.n'), n\rangle \downarrow u}{\langle H, \mathbf{let}\ x : A = n\ \mathbf{in}\ n'\rangle \downarrow u} \qquad \textsc{E-LetValue}\ \frac{\langle H, n[v/x]\rangle \downarrow u}{\langle H \circ (x : A.n), \downarrow v\rangle \downarrow u}$$

$$\textsc{E-App}\ \frac{\langle H \circ (\mathbf{rec}\ f\{e\} : \{C\}, \Box\ n_2), n_1\rangle \downarrow u}{\langle H, \downarrow(\mathbf{rec}\ f\{e\} : \{C\})\ \overline{n}\rangle \downarrow u} \qquad \textsc{E-AppV-Fst}\ \frac{\langle H \circ (\mathbf{rec}\ f\{e\} : \{C\}, \downarrow v\ \Box), n\rangle \downarrow u}{\langle H \circ (\mathbf{rec}\ f\{e\} : \{C\}, \Box\ n), \downarrow v\rangle \downarrow u}$$

$$\textsc{E-AppV-Snd}\ \frac{\mathbf{unfold}(e) : \overline{\langle\Delta\rangle A} \longleftarrow u, v \text{–}[\Sigma]\ n \qquad \langle H, n\rangle \downarrow u'}{\langle H \circ (\mathbf{rec}\ f\{e\} : \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\}, u\ \Box), \downarrow v\rangle \downarrow u'}$$

$$\textsc{E-HdlCmd-Fst}\ \frac{C = \overline{\langle\Delta\rangle A} \to [\Sigma]B \quad c : A_1 \to B_1 \in \Delta_1(\varnothing) \quad \langle H \circ (\mathbf{rec}\ f\{e\} : \{C\}, (H' \star c\ v)\ \Box), n\rangle \downarrow u}{\langle H \circ (\mathbf{rec}\ f\{e\} : \{C\}, \Box\ n), H' \star c\ v\rangle \downarrow u}$$

$$\textsc{E-HdlCmd-Snd}\ \frac{c : A_1 \to B_1 \in \Delta_2(\varnothing) \qquad \mathbf{unfold}(e) : \overline{\langle\Delta\rangle A} \longleftarrow u, H' \star c\ v \text{–}[\Sigma]\ n \qquad \langle H, n\rangle \downarrow u}{\langle H \circ (\mathbf{rec}\ f\{e\} : \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\}, u\ \Box), H' \star c\ v\rangle \downarrow u}$$

$$\textsc{E-ForwardCmd}\ \frac{c\text{-stuck}(F) \quad \langle H, F \bullet H' \star c\ v\rangle \downarrow u}{\langle H \circ F, H' \star c\ v\rangle \downarrow u} \qquad \textsc{E-IdCmd}\ \frac{c : A' \to B' \in \Sigma}{\langle Id_\Sigma, H \star c\ v\rangle \downarrow H \star c\ v}$$

Figure 6.6: Ella handler stack semantics

$$\textsc{M-Head}\ \frac{(r_i : \langle \Delta_i \rangle A_i \longleftarrow u_i \dashv [\Sigma]\ \theta_i)_{i=1,2}}{(\overline{r} \mapsto n :: e) : \overline{\langle \Delta \rangle A} \longleftarrow \overline{u} \dashv [\Sigma]\ \overline{\theta}(n)}$$

$$\textsc{M-Tail}\ \frac{\exists i = 1,2.\ r_i : \langle \Delta_i \rangle A_i \longleftarrow u_i \dashv [\Sigma]\ \theta \implies \bot \qquad e : \langle \Delta \rangle A \longleftarrow u \dashv [\Sigma]\ n}{(\overline{r} \mapsto n :: e) : \overline{\langle \Delta \rangle A} \longleftarrow \overline{u} \dashv [\Sigma]\ n}$$

$$\textsc{M-Var}\ \frac{}{x : A \longleftarrow v \dashv [v/x]} \qquad \textsc{M-Bool}\ \frac{b \in \{\mathbf{tt}, \mathbf{ff}\}}{b : \mathbf{bool} \longleftarrow b \dashv \cdot}$$

$$\textsc{M-Value}\ \frac{p : A \longleftarrow v \dashv \theta}{p : \langle \Delta \rangle A \longleftarrow v \dashv [\Sigma]\ \theta} \qquad \textsc{M-Catchall-Val}\ \frac{C = \{\overline{\langle \iota \rangle \mathbf{unit}} \to [\Delta(\Sigma)]A\}}{\langle x \rangle : \langle \Delta \rangle A \longleftarrow v \dashv [\Sigma]\ [\uparrow^C((), () \mapsto v)/x]}$$

$$\textsc{M-Catchall-Req}\ \frac{C = \{\overline{\langle \iota \rangle \mathbf{unit}} \to [\Delta(\Sigma)]A\}}{\langle x \rangle : \langle \Delta \rangle A \longleftarrow H \star c\ v \dashv [\Sigma]\ [\uparrow^C((), () \mapsto H \star c\ v)/x]}$$

$$\textsc{M-Command}\ \frac{c : A_1 \to B_1 \in \Sigma \quad p : A_1 \longleftarrow v \dashv \theta \quad C = \{\langle \iota \rangle B_1, \langle \iota \rangle \mathbf{unit} \to [\Delta(\Sigma)]A\}}{\langle c\ p \to z \rangle : \langle \Delta \rangle A \longleftarrow H \star c\ v \dashv [\Sigma]\ \theta[\uparrow^C(x, () \mapsto H @ \downarrow x)/z]}$$

Figure 6.7: ELLA Pattern Matching

$$
\begin{array}{rcl}
\langle H, \downarrow(\mathbf{rec}\, f\{e\} : \{C\})\, \overline{n}\rangle & \longrightarrow_\alpha & \langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, n_2), n_1\rangle \\
\langle H, \mathbf{let}\, x : A = n\, \mathbf{in}\, n'\rangle & \longrightarrow_\alpha & \langle H \circ (x : A.n'), n\rangle \\
\langle H \circ (x : A.n), \downarrow v\rangle & \longrightarrow_\beta & \langle H, n[v/x]\rangle \\
\langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, \Box\, n), \downarrow v\rangle & \longrightarrow_\alpha & \langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, \downarrow v\, \Box), n\rangle \\
\langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, u\, \Box), \downarrow v\rangle & \longrightarrow_\beta & \langle H, n\rangle \\
\quad \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B & & \quad \text{if } \mathbf{unfold}(e) : \overline{\langle\Delta\rangle A} \longleftarrow u, v \text{ -}[\Sigma]\ n \\
\langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, \Box\, n), H' \star c\ v\rangle & \longrightarrow_\alpha & \langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, (H' \star c\ v)\, \Box), n\rangle \\
\quad \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B & & \quad \text{if } c : A_c \to B_c \in \Delta_1(\varnothing) \\
\langle H \circ (\mathbf{rec}\, f\{e\} : \{C\}, u\, \Box), H' \star c\ v\rangle & \longrightarrow_\beta & \langle H, n\rangle \\
\quad \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B & & \quad \text{if } c : A_c \to B_c \in \Delta_2(\varnothing) \text{ and} \\
 & & \quad \mathbf{unfold}(e) : \overline{\langle\Delta\rangle A} \longleftarrow u, H' \star c\ v \text{ -}[\Sigma]\ n \\
\langle H \circ F, H' \star c\ v\rangle & \longrightarrow_\beta & \langle H, F \bullet H' \star c\ v\rangle \\
 & & \quad \text{if } c\text{-stuck}(F)
\end{array}
$$

Figure 6.8: ELLA small-step stack semantics

Lemma 6.5. *Given* $\cdot\ [\Sigma]\text{–}\ H : [\Sigma']A \multimap B$ *and* $\cdot\ [\Sigma']\text{–}\ n : A$ *then either*

(a) $H \equiv Id$ *and* $n \equiv \downarrow v$ *with* $\cdot\ [\Sigma]\text{–}\ v \Rightarrow B$*;*

(b) $H \equiv Id$ *and* $n \equiv H' \star c\ v$ *with* $c : A_c \to B_c \in \Sigma$, $\cdot\ [\Sigma']\text{–}\ v \Rightarrow A_c$, $\cdot\ [\Sigma]\text{–}\ H' : [\Sigma']B_c \multimap A$ *and* c-stuck(H')*;*

(c) $\exists!\mathcal{C}.\langle H, n\rangle \longrightarrow \mathcal{C} \wedge \cdot\ [\Sigma]\text{–}\ \mathcal{C} : B$.

Proof. By case analysis on the form of n using Proposition 6.4. For $n \equiv \downarrow v$ and $n \equiv H' \star c\ v$ we additionally perform a case analysis on H. □

Lemma 6.6. $\langle H, H'@n\rangle \longrightarrow_\alpha^* \langle H \circ H', n\rangle$.

Lemma 6.7. *If* $\langle H_1, n_1\rangle \longrightarrow \langle H_2, n_2\rangle$ *then* $\langle H' \circ H_1, n_1\rangle \longrightarrow \langle H' \circ H_2, n_2\rangle$

Proof. The statement follows from:

1. $\langle H_1, n_1\rangle \longrightarrow_\beta \langle H_2, n_2\rangle \implies \langle H' \circ H_1, n_1\rangle \longrightarrow_\beta \langle H' \circ H_2, n_2\rangle$;
2. $\langle H_1, n_1\rangle \longrightarrow_\alpha^* \langle H_2, n_2\rangle \implies \langle H' \circ H_1, n_1\rangle \longrightarrow_\alpha^* \langle H' \circ H_2, n_2\rangle$.

which follow by induction on the $\longrightarrow_\beta$ and $\longrightarrow_\alpha^*$ premiss, respectively. □

6.2 Formalising Contexts

We develop the same machinery for observational behaviour as described in Chapter 4. Namely, we define observational approximation to be parametric in the notion of contexts e.g. *variable-capturing contexts* (VCCs), *value-substituting contexts* (VSCs), *etc.* Our grammar for contexts reflects the inferrable and checkable distinction made at the term-level.

A *term context* $\mathcal{N}$ or $\mathcal{M}$ is a possibly-open term of arbitrary type containing zero or more occurrences of a well-typed and well-scoped hole, where $\mathcal{N}$ ranges over *checkable* term contexts and $\mathcal{M}$ ranges over *inferrable* term contexts. Analogously, a *value context* $\mathcal{V}$ or $\mathcal{W}$ is a possibly-open value of arbitrary type containing zero or more occurrences of such a hole. Finally, a *computation context* $\mathcal{E}$ is a possibly-open computation of arbitrary computation type containing zero or more occurrences of a well-typed and well-scoped hole.

Figure 6.9 presents the typing rules for VCCs; the rules for VSCs are analogous except in the case of the hole which carries a well-typed substitution. The form of a well-typed and well-scoped context $\mathcal{N}$ is $\Gamma_0 \, [\Sigma_0] \vdash \mathcal{N} : \langle\langle \Gamma_1 \, [\Sigma_1] \vdash A \rangle\rangle^{\mathrm{VCC}} B$, which says that term context $\mathcal{N}$ checks against type B in environment Γ_0 and ambient ability Σ_0 and contains zero or more occurrences of a hole of type A in an environment Γ_1 and ambient ability Σ_1. We enforce the restriction that the hole must be a checkable term to simplify the definition of contexts. We can always place an inferrable term in the hole using the explicit injection '$\downarrow$' operator. In cases where this restriction prevents us from representing a certain context, e.g. `put` $\langle\langle - \rangle\rangle$ for command `put : bool -> unit`, we can always convert it to a valid context which checks the hole against its type using **`let`**, e.g. **`let`** `x : bool =` $\langle\langle - \rangle\rangle$ **`in`** `put x`.

Hence, operations on contexts are defined only for checkable term contexts. In particular, we define context instantiation, denoted by $\mathcal{N}[n]^{\mathrm{VCC}}$, for any (checkable) term context $\mathcal{N}$ and (checkable) term n by structural traversal on $\mathcal{N}$.

Another simplification to the grammar of term contexts is the VCCCOMMAND rule where the command invocation is not paired with a handler stack (or its VCC equivalent). This does not affect the expressive power of contexts because stacks are simply an alternative representation to evaluation contexts; they do not add expressivity. However, context instantiation must take this discrepancy into account:

$$(c \; \mathcal{V})[n] \triangleq Id \star c \; \mathcal{V}[n]$$

$\boxed{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{M} \Rightarrow \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B}$

VCCTHUNK
$$\frac{\Gamma_0, f : \{C\} \vdash \mathcal{E} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} C}{\Gamma_0\,[\Sigma_0]\text{–}\ (\mathbf{rec}\ f\{\mathcal{E}\} : \{C\}) \Rightarrow \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} \{C\}}$$

VCCAPP
$$\frac{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{V} \Rightarrow \langle\langle \Gamma_1\,[\Sigma_3]\text{–}\ A_3\rangle\rangle^{\mathrm{VCC}} \{\overline{\langle\Delta\rangle A} \to [\Sigma] B\} \quad \Delta_i(\Sigma_0) = \Sigma_i \qquad (\Gamma_0\,[\Sigma_i]\text{–}\ \mathcal{N}_i : \langle\langle \Gamma_1\,[\Sigma_3]\text{–}\ A_3\rangle\rangle^{\mathrm{VCC}} A_i)_{i=1,2}}{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{V}\ \overline{\mathcal{N}} \Rightarrow \langle\langle \Gamma_1\,[\Sigma_3]\text{–}\ A_3\rangle\rangle^{\mathrm{VCC}} B}$$

$\boxed{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{N} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B}$

VCCHOLE
$$\frac{}{\Gamma\,[\Sigma]\text{–}\ \langle\langle -\rangle\rangle : \langle\langle \Gamma\,[\Sigma]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} A}$$

VCCTRM
$$\frac{\Gamma_0\,[\Sigma_0]\text{–}\ n : B}{\Gamma_0\,[\Sigma_0]\text{–}\ \mathbf{trm}\ n : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B}$$

VCCSWITCH
$$\frac{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{M} \Rightarrow \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B \qquad B = B'}{\Gamma_0\,[\Sigma_0]\text{–}\ \downarrow\mathcal{M} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B'}$$

VCCCOMMAND
$$\frac{c : B \to B' \in \Sigma_0 \quad \Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{V} \Rightarrow \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B}{\Gamma_0\,[\Sigma_0]\text{–}\ c\ \mathcal{V} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B'}$$

VCCLET
$$\frac{\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{N}_1 : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A_1\rangle\rangle^{\mathrm{VCC}} A \qquad \Gamma_0, x : A\,[\Sigma_0]\text{–}\ \mathcal{N}_2 : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A_1\rangle\rangle^{\mathrm{VCC}} B}{\Gamma_0\,[\Sigma_0]\text{–}\ \mathbf{let}\ x : A = \mathcal{N}_1\ \mathbf{in}\ \mathcal{N}_2 : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A_1\rangle\rangle^{\mathrm{VCC}} B}$$

$\boxed{\Gamma_0 \vdash \mathcal{E} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} C}$

VCCCOMP
$$\frac{(r_{i,j} : T_j \dashv [\Sigma_0]\ \Gamma'_{i,j})_{i,j} \qquad (r_{i,j})_{i,j}\ \text{covers}\ (T_j)_j \qquad (\Gamma_0, (\Gamma'_{i,j})_j\,[\Sigma_0]\text{–}\ \mathcal{N}_i : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} B)_i}{\Gamma_0 \vdash \overline{r \mapsto \mathcal{N}} : \langle\langle \Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{\mathrm{VCC}} (\overline{T} \to [\Sigma_0] B)}$$

Figure 6.9: ELLA typing rules for variable-capturing contexts

Now we define observational approximation for ELLA. The key depature from our definition in Chapter 4 is that for ELLA, we need only observe termination rather than the inspection of ground values for $\lambda^{\rightarrow}_{FG}$ — a consequence of introducing general recursion.

Definition 6.8 ($\mathcal{K}$ Approximation)**.** For terms n_1, n_2 of type A in environment Γ and ambient Σ:

$$\Gamma \vdash n_1 \lesssim^{\mathcal{K}} n_2 : [\Sigma]A$$

asserts that for all program contexts, $[\varnothing]\text{–}\, \mathcal{P} : \langle\langle \Gamma\, [\Sigma]\text{–}\, A\rangle\rangle^{\mathcal{K}}\mathbf{unit}$,

$$\mathcal{P}[n_1]^{\mathcal{K}} \Downarrow \implies \mathcal{P}[n_2]^{\mathcal{K}} \Downarrow$$

where evaluation of a closed term n, written $n \Downarrow$, is defined by:

$$n \Downarrow \triangleq \langle Id, n\rangle \downarrow$$

where $\langle Id, n\rangle \downarrow \triangleq \exists u.\langle Id, n\rangle \downarrow u$.

Simply observing equitermination for all `unit`-valued contexts is sufficient. For any other ELLA ground type, we can construct an operator which will distinguish distinct values via pattern matching. Again, value-substituting contexts are the largest class of contexts and thus form our definition of observational approximation.

Definition 6.9. Let observational approximation be $\mathcal{K}$ approximation with $\mathcal{K} = \text{VSC}$.

Unsurprisingly, VCCs are contained within VSCs. The proof is the same as for Lemma 4.9.

Lemma 6.10. $\Gamma \vdash n_1 \lesssim^{VSC} n_2 : [\Sigma]A \implies \Gamma \vdash n_1 \lesssim^{VCC} n_2 : [\Sigma]A$.

We have an analogous way of closing a VCC using a substitution that we presented in Chapter 4 (Definition 4.12).

Definition 6.11 (Let-Redexes from Substitution)**.** Let θ be a Γ_0-closing substitution. For $\Gamma_0\, [\Sigma_0]\text{–}\, \mathcal{D} : \langle\langle \Gamma\, [\Sigma]\text{–}\, A\rangle\rangle^{\text{VCC}} B$, define $\mathcal{D}^{\theta}$ by induction on the size of Γ_0:

$$\mathcal{D}^{\theta} = \begin{cases} \mathcal{D} & \Gamma_0 = \cdot \\ (\mathbf{let}\, x : A = v\, \mathbf{in}\, \mathcal{D})^{\theta'} & \Gamma_0 = \Gamma_0', x : A \text{ and} \\ & \theta = \theta'[v/x] \end{cases}$$

$\mathcal{D}^{\theta}$ represents a new (closed) VCC obtained from $\mathcal{D}$ by constructing a sequence of let redexes on the outside of $\mathcal{D}$ from the substitution θ. For $\Gamma\, [\Sigma]\text{–}\, n : A$, let n^{θ} denote the analogous operation for constructing a closed term from n and θ.

$$\frac{}{n \rightsquigarrow^s n}\ \text{L-REFL} \qquad \frac{n \rightsquigarrow^s \mathbf{let}\, x : A = \downarrow v\, \mathbf{in}\, n'}{n \rightsquigarrow^s n'[v/x]}\ \text{L-RED} \qquad \frac{n_1 \rightsquigarrow^s n_1'}{\mathbf{let}\, x : A = n_1\, \mathbf{in}\, n_2 \rightsquigarrow^s \mathbf{let}\, x : A = n_1'\, \mathbf{in}\, n_2}\ \text{L-LET}$$

$$\frac{n_1 \rightsquigarrow^s n_1' \quad n_2 \rightsquigarrow^s n_2'}{\uparrow e\ n_1\ n_2 \rightsquigarrow^s \uparrow e\ n_1'\ n_2'}\ \text{L-APP}$$

Figure 6.10: ELLA iterated reduction of let bindings (term sequencing)

Figure 6.10 defines iterated reduction of term sequencing. Just like the $\rightsquigarrow^S$ relation in Chapter 4, $\rightsquigarrow^s$ respects the evaluation semantics of ELLA. Furthermore, we have the result

Lemma 6.12. *For* $\Gamma\,[\Sigma]\text{–}\ n : A$*, and* Γ*-closing substitution* θ*, then we have*

$$n^{\theta} \rightsquigarrow^s \theta(n)$$

Proof. By induction on the size of environment Γ and then case analysis on θ.

Case $\Gamma = \cdot$ and $\theta = \cdot$. Immediate.

Case $\Gamma = \Gamma', x : A$ and $\theta = \theta'[v/x]$.

By Definition 6.11 our goal expands to showing:

$$(\mathbf{let}\, x : A = v\, \mathbf{in}\, n)^{\theta'} \rightsquigarrow^s \theta'(n[v/x])$$

From the induction hypothesis we know:

$$\begin{array}{rcl} (\mathbf{let}\, x : A = v\, \mathbf{in}\, n)^{\theta'} & \rightsquigarrow^s & \theta'(\mathbf{let}\, x : A = v\, \mathbf{in}\, n) \\ & \equiv & \mathbf{let}\, x : A = v\, \mathbf{in}\, \theta'(n) \quad (v \text{ is a closed value}, x \notin dom(\Gamma')) \\ & \rightsquigarrow^s & \theta'(n)[v/x] \quad (\text{L-RED}) \end{array}$$

□

Lemma 6.13. *For all* $\Gamma\,[\Sigma]\text{–}\ n : A$*,* $\Gamma_0\,[\Sigma_0]\text{–}\ \mathcal{C} : \langle\langle \Gamma\,[\Sigma]\text{–}\ A\rangle\rangle^{VCC} B$*, and* Γ_0*-closing substitutions,* θ*, we have*

$$(\mathcal{C}^{\theta})[n] \equiv (\mathcal{C}[n])^{\theta}$$

Proof. By induction on the size of the environment Γ_0. □

Like the frame stack development in Chapter 4, it turns out that we need only consider the class of the contexts corresponding to the frames comprising handler stacks.

Definition 6.14 (CIU Contexts)**.** Let *CIU contexts* be the class of contexts defined by the following inference rules:

$$\textsc{ciuHole}\quad \frac{\theta : \Gamma_1 \vDash \Gamma_0}{\Gamma_0\,[\Sigma_0]\text{-}\ \langle\langle \theta - \rangle\rangle : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A\rangle\rangle^{\mathrm{CIU}} A}$$

$$\textsc{ciuLet}\quad \frac{\Gamma_0\,[\Sigma_0]\text{-}\ \mathcal{D} : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} A \qquad \Gamma_0, x : A\,[\Sigma_0]\text{-}\ n : B}{\Gamma_0\,[\Sigma_0]\text{-}\ \mathbf{let}\ x : A = \mathcal{D}\ \mathbf{in}\ n : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} B}$$

$$\textsc{ciuApp-Fst}\quad \frac{C = \{\overline{\langle\Delta\rangle A'} \to [\Sigma_0]B\} \quad \Gamma_0, f : \{C\} \vdash e : C \quad (\Delta_i(\Sigma_0) = \Sigma_i')_{i=1,2} \quad \Gamma_0\,[\Sigma_1']\text{-}\ \mathcal{D} : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} A_1' \quad \Gamma_0\,[\Sigma_2']\text{-}\ n : A_2'}{\Gamma_0\,[\Sigma_0]\text{-}\ (\uparrow^C e)\ \mathcal{D}\ n : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} B}$$

$$\textsc{ciuApp-Snd}\quad \frac{C = \{\overline{\langle\Delta\rangle A'} \to [\Sigma_0]B\} \quad \Gamma_0, f : \{C\} \vdash e : C \quad (\Delta_i(\Sigma_0) = \Sigma_i')_{i=1,2} \quad \Gamma_0\,[\Sigma_1']\text{-}\ n : A_1' \quad \Gamma_0\,[\Sigma_2']\text{-}\ \mathcal{D} : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} A_2'}{\Gamma_0\,[\Sigma_0]\text{-}\ (\uparrow^C e)\ n\ \mathcal{D} : \langle\langle \Gamma_1\,[\Sigma_1]\text{-}\ A_1\rangle\rangle^{\mathrm{CIU}} B}$$

Definition 6.15. Define the operation $\blacklozenge : \mathit{CIU} \to \mathit{VCC}$, which transforms a CIU context into a closed VCC, by structural recursion on the CIU context:

$$\begin{aligned}
\blacklozenge(\langle\langle \theta - \rangle\rangle) &= \langle\langle - \rangle\rangle^{\theta} \\
\blacklozenge(\mathbf{let}\ x : A = \mathcal{D}\ \mathbf{in}\ n) &= \mathbf{let}\ x : A = \blacklozenge(\mathcal{D})\ \mathbf{in}\ n \\
\blacklozenge(\uparrow e\ \mathcal{D}\ n) &= \uparrow e\ \blacklozenge(\mathcal{D})\ n \\
\blacklozenge(\uparrow e\ n\ \mathcal{D}) &= \uparrow e\ n\ \blacklozenge(\mathcal{D})
\end{aligned}$$

Lemma 6.16. *For* $\Gamma\,[\Sigma]\text{-}\ n : A$ *and* $\cdot\,[\Sigma]\text{-}\ \mathcal{P} : \langle\langle \Gamma\,[\Sigma]\text{-}\ A\rangle\rangle^{\mathit{CIU}} B$*,*

$$(\blacklozenge(\mathcal{P}))[n] \rightsquigarrow^s \mathcal{P}[n]$$

Proof. The proof proceeds by induction on the structure of $\mathcal{P}$.

Case $\mathcal{P} = \langle\langle \theta - \rangle\rangle$.

By definition, $\blacklozenge(\mathcal{P}) = \langle\langle - \rangle\rangle^{\theta}$. We calculate:

$$\begin{aligned}
(\langle\langle - \rangle\rangle^{\theta})[n] &\equiv n^{\theta} && \text{by Lemma 6.13} \\
&\rightsquigarrow^s \theta(n) && \text{by Lemma 6.12.}
\end{aligned}$$

Case $\mathcal{P} = \mathbf{let}\, x : B' = Q\, \mathbf{in}\, n'$.

By the induction hypothesis, $(\blacklozenge(Q))[n] \rightsquigarrow^s Q[n]$. We calculate:

$$\begin{array}{rcll} (\mathbf{let}\, x : B' = \blacklozenge(Q)\, \mathbf{in}\, n')[n] & = & \mathbf{let}\, x : B' = (\blacklozenge(Q))[n]\, \mathbf{in}\, n' & \\ & \rightsquigarrow^s & \mathbf{let}\, x : B' = Q[n]\, \mathbf{in}\, n' & \text{by I.H. and L-LET} \\ & = & \mathcal{P}[n] & \text{as required.} \end{array}$$

Case $\mathcal{P} = \uparrow e\ Q\ n'$.

By the induction hypothesis, $(\blacklozenge(Q))[n] \rightsquigarrow^s Q[n]$. We calculate:

$$\begin{array}{rcll} (\uparrow e\ \blacklozenge(Q)\ n')[n] & = & \uparrow e\ (\blacklozenge(Q))[n]\ n' & \\ & \rightsquigarrow^s & \uparrow e\ Q[n]\ n' & \text{by I.H., L-APP, and L-REFL for } n' \\ & = & \mathcal{P}[n] & \text{as required.} \end{array}$$

Case $\mathcal{P} = \uparrow e\ n\ Q$.

This case is similar to the previous one.

□

Lemma 6.17.

$$\Gamma \vdash n_1 \lesssim^{VCC} n_2 : [\Sigma]A \implies \Gamma \vdash n_1 \lesssim^{CIU} n_2 : [\Sigma]A$$

Proof. Assume given a CIU context $\mathcal{P}$. Instantiate the premiss with $\blacklozenge(\mathcal{P})$. The proof proceeds by induction on the structure of $\mathcal{P}$ using Lemma 6.16 and closure of evaluation under $\rightsquigarrow^s$. □

6.2.1 Handler Stacks and CIU Contexts

We establish a connection between the left action over a handler stack and CIU context instantiation.

Definition 6.18. Let $H\langle\langle @ \rangle\rangle\mathcal{P}$ be the operation which takes a handler stack H and a closed CIU context $\mathcal{P}$ to produce a closed CIU context:

$$\begin{array}{rcl} Id@\mathcal{P} & \triangleq & \mathcal{P} \\ (H \circ (x : A.n'))@\mathcal{P} & \triangleq & H@(\mathbf{let}\, x : A = \mathcal{P}\, \mathbf{in}\, n') \\ (H \circ (\mathbf{rec}\, f\{e\} : \{C\}, \Box\ n))@\mathcal{P} & \triangleq & H@((\uparrow^C e\ \mathcal{P}\ n) \\ (H \circ (\mathbf{rec}\, f\{e\} : \{C\}, u\ \Box))@\mathcal{P} & \triangleq & H@((\uparrow^C e\ u\ \mathcal{P}) \end{array}$$

Using the above operation, we have the following "commutativity" result for context instantiation and left action over a handler stack. Regardless of the order of the operations, the same term arises.

Lemma 6.19. $(H\langle\langle @ \rangle\rangle \mathcal{P})[n]^{CIU} \equiv H @ (\mathcal{P}[n]^{CIU})$

Proof. By induction on the handler stack H. □

6.3 Relational Reasoning

We have formalised a notion of contexts for ELLA together with a definition of contextual approximation parametric in the notion of context. From this, we defined observational approximation to be contextual approximation in which the contexts are VSCs. Now we want to define applicative and logical notions of behavioural approximation for ELLA to complete our triangle of relations.

In this section we introduce our relations for reasoning about ELLA programs based on $\texttt{FOL}^{\mu\triangleright}$ presented in Chapter 5. The signature Ω of $\texttt{FOL}^{\mu\triangleright}$ is instantiated with a universe of codes corresponding to the phrase classes of ELLA defined in Figure 6.1, and the evaluation and small-step reduction relations defined on configurations.

6.3.1 Basic Observable

Both of our structural relations rely on a relation capturing basic observable behaviour similar to ground equivalence (Definition 4.6) for $\lambda^{\rightarrow}_{FG}$. However, just like contextual approximation for ELLA, our basic observation is equitermination of **unit**-valued configurations, not equality on ground values.

As discussed previously, we have elected to not stratify our evaluation relations with step-indices (cf. Pitts [2010]), instead we define a basic observation relation by recursion on the step-index. The recursive occurrence appears under the $\triangleright$ modality to ensure the definition is well-founded.

Definition 6.20 (Basic Observable in $\texttt{FOL}^{\mu\triangleright}$)**.** The basic observable relation is the following fixed-point predicate between pairs of closed, pure, **unit**-valued configurations.

$$\mathbf{O} \triangleq \mu P : \texttt{Config} \times \texttt{Config}.(\mathcal{C}_1 : \texttt{Config}, \mathcal{C}_2 : \texttt{Config}).\phi$$

where $\texttt{Config} \triangleq \texttt{Stk} \times \texttt{Trm}^{\downarrow}$ and

$$\phi \triangleq (\mathcal{C}_1 \equiv \langle Id, \downarrow() \rangle \wedge \mathcal{C}_2 \downarrow) \vee (\exists\, \mathcal{C}_1' : \texttt{Config}.\mathcal{C}_1 \longrightarrow \mathcal{C}_1' \wedge \triangleright P(\mathcal{C}_1', \mathcal{C}_2)).$$

It is straightforward to show that the above definition satisfies the contractivity constraint (Example 5.9) required of a fixed-point. Note that for brevity we have omitted the necessary typing and scoping constraints on configurations to ensure they are closed, pure, **unit**-valued configurations. These constraints arise as arguments to the codes, *e.g.* $n : \mathtt{Trm}^{\downarrow}\ \Gamma\ \Sigma\ A$ specifies the well-typed and-scoped term $\Gamma\ [\Sigma]\text{–}\ n : A$.

We simplify the definition slightly for readability purposes, omitting sort ascription on binders and conflating **O** with the predicate variable P, yielding the following definition. We adopt the terminological convention of using "relation" for a binary predicate in $\mathtt{FOL}^{\mu\triangleright}$.

Definition 6.21. The basic observable relation is the following binary predicate between pairs of closed, pure, **unit**-valued configurations.

$$\mathbf{O}(\mathcal{C}_1, \mathcal{C}_2) \triangleq (\mathcal{C}_1 \equiv \langle Id, \downarrow() \rangle \wedge \mathcal{C}_2 \downarrow) \vee (\exists\ \mathcal{C}_1'.\mathcal{C}_1 \longrightarrow \mathcal{C}_1' \wedge \triangleright \mathbf{O}(\mathcal{C}_1', \mathcal{C}_2)).$$

Definition 6.21 captures approximate behaviour between configurations. That is, $\mathcal{C}_1$ behaviourally approximates $\mathcal{C}_2$ if and only if $\mathcal{C}_1$ terminates then so does $\mathcal{C}_2$ [1]. We establish the following adequacy result which captures this property of **O**.

Lemma 6.22 (Adequacy)**.** *If* $\Vdash \mathbf{O}(\mathcal{C}_1, \mathcal{C}_2)$ *and* $\mathcal{C}_1 \downarrow$ *then* $\mathcal{C}_2 \downarrow$

Proof. By the correspondence between termination and small-step semantics, we know there exists a j such that $\mathcal{C}_1 \longrightarrow^j \langle Id, \downarrow() \rangle$. The result follows by induction on the number of reduction steps j. □

Remark 6.23. Definition 6.21 defines our basic observable recursively using the $\triangleright$ modality. A simpler formulation of equi-termination such as:

$$\mathcal{C}_1 \downarrow () \implies \mathcal{C}_2 \downarrow () \quad (\star)$$

does not suffice because it fails to marry the implicit step-index with the reduction steps of the configuration $\mathcal{C}_1$. This connection is essential to correctly model recursive features using step-indexing and justify our appeals to the Löb rule described in Chapter 5. An alternative formulation could make use of $(\star)$ for non-recursive β-steps, *e.g.* S-LetValue, and reserve step-indexing only for the recursive features. This approach was taken by Dreyer et al. [2011] but has limited benefit in our Ella setting due to the small number of rules to which such a condition would apply.

[1] It is an asymmetric relation (cf. ground equivalence of Definition 4.6). As mentioned previously, we only consider the approximations in this dissertation so this asymmetric definition is sufficient.

Remark 6.24. Theorem statements in this chapter of the form "If X_1 and X_2 and ... and X_n then Y" should be interpreted within the logical framework provided by $ML_{\mathtt{UTT}}$, in the same sense of the discussion regarding inference rules (Section 5.2), *i.e.* interpreted using the function-space of $ML_{\mathtt{UTT}}$, rather than the logic $\mathtt{FOL}^{\mu\triangleright}$. If a statement is to be understood entirely within $\mathtt{FOL}^{\mu\triangleright}$ logic, we use the main judgement $\overline{X} \Vdash Y$, assuming appropriate injections for $\overline{X}$ and Y into `IProp`.

Lemma 6.25. *If* $\Vdash \mathbf{O}(\mathcal{C}_1, \mathcal{C}_2)$ *and* $\Vdash \mathbf{O}(\mathcal{C}_2, \mathcal{C}_3)$ *then* $\Vdash \mathbf{O}(\mathcal{C}_1, \mathcal{C}_3)$.

Proof. By Löb induction, case analysis on the first premiss, and Lemma 6.22. □

Lemma 6.26. **O** *is closed under expansion:*

1. $\mathcal{C}_1 \longrightarrow \mathcal{C}_1' \Vdash \triangleright\mathbf{O}(\mathcal{C}_1', \mathcal{C}_2) \implies \mathbf{O}(\mathcal{C}_1, \mathcal{C}_2)$
2. $\mathcal{C}_2 \longrightarrow \mathcal{C}_2' \Vdash \mathbf{O}(\mathcal{C}_1, \mathcal{C}_2') \implies \mathbf{O}(\mathcal{C}_1, \mathcal{C}_2)$

Proof. Part 1 holds by definition while Part 2 is by Löb induction. □

Recall that administrative steps rearrange the components of a configuration but do not alter the term a configuration represents. Given Definition 6.21, we may perform a finite but arbitrary number of administrative steps without decreasing the step-index. This is important for rearranging configurations in proofs in much the same way as one would informally rearrange an evaluation context paired with its instantiating term.

Lemma 6.27. *The basic observable approximation satisfies the properties:*

1. $\mathcal{C}_1 \longrightarrow_\alpha^* \mathcal{C}_2 \Vdash \mathbf{O}(\mathcal{C}_1, \mathcal{C}_3) \iff \mathbf{O}(\mathcal{C}_2, \mathcal{C}_3)$
2. $\mathcal{C}_1 \longrightarrow_\alpha^* \mathcal{C}_2 \Vdash \mathbf{O}(\mathcal{C}_3, \mathcal{C}_1) \iff \mathbf{O}(\mathcal{C}_3, \mathcal{C}_2)$

Proof. Consider each property separately:

1. For $\Longleftarrow$, perform a case analysis on the approximation assumption. For $\Longrightarrow$, we appeal to the property:

   $$\mathcal{C}_1 \longrightarrow_\alpha^* \mathcal{C}_2 \wedge \mathcal{C}_1 \longrightarrow \mathcal{C}_1' \implies \mathcal{C}_2 \longrightarrow \mathcal{C}_1'$$

   which is proven using determinacy of reduction.

2. For $\Longleftarrow$, perform a case analysis on the approximation assumption. For $\Longrightarrow$, we additionally appeal to Löb induction.

□

From the above property, we establish a useful property of observable behaviour between the left action over a handler stack and stack concatenation.

Corollary 6.28.

1. $\Vdash \mathbf{O}(\langle H \circ H', n\rangle, \mathcal{C}) \iff \mathbf{O}(\langle H, H'@n\rangle, \mathcal{C})$
2. $\Vdash \mathbf{O}(\mathcal{C}, \langle H \circ H', n\rangle) \iff \mathbf{O}(\mathcal{C}, \langle H, H'@n\rangle)$

Proof. By Lemma 6.27 using Lemma 6.6. □

6.3.2 Logical Approximation

The definition of the logical approximation relation is presented in Figure 6.11. We extend biorthogonality to the $\top\top$-lifting of *normal forms*, given by $\mathbf{U}[\![-]\!]$, over *handler stacks*, given by $\mathbf{S}[\![-]\!]$. All of the relations with the exception of $\approx$, which is an $ML_{\mathtt{UTT}}$-level definition, are defined within the $\mathtt{FOL}^{\mu\triangleright}$ logic as fixed-point predicates but we have presented them as mutually recursive function definitions for the sake of readability. For these definitions to be well-founded, they must be contractive in their recursive occurrences. Admittedly, it is not immediately apparent that they are contractive given their nontrivial mutually recursive nature. We provide a proof of contractivity for the relations in Figure 6.11 in Appendix B to avoid disturbing the flow of our relational reasoning development by presenting it here. Our proof mostly follows the mechanised development by Biernacki et al. [2018] but with a few differences due to the generalisation of function application to incorporate effect handling, and the absence of effect rows and effect polymorphism.

In this section, we prove some properties for reasoning about logical term approximation, and show logical approximation is closed under all VSCs, yielding one side of our triangulation result.

Lemma 6.29. *The following properties hold for any type A and ability* Σ*:*

1. $\mathbf{V}[\![A]\!]^{\lessapprox}(v_1, v_2) \Vdash \mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}(\downarrow v_1, \downarrow v_2)$
2. $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}(u_1, u_2) \Vdash \mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}(u_1, u_2)$

Proof. Both parts follow from the definitions of $\mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}$, $\mathbf{S}[\![[\Sigma]A]\!]^{\lessapprox}$ and $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}$. □

Component Relations

$$
\begin{array}{lcl}
\mathbf{V}[\![\ \tau\ ]\!]^{\lessapprox}(v_1,v_2) & \triangleq & v_1 \equiv v_2 \\
\mathbf{V}[\![\ \{C\}\ ]\!]^{\lessapprox}(\uparrow e_1,\uparrow e_2) & \triangleq & \forall \overline{u_1},\overline{u_2},n_1.\ \bigwedge_i(\mathbf{U}[\![\ [\Delta_i(\Sigma)]A_i\ ]\!]^{\lessapprox}(u_{1,i},u_{2,i})) \wedge e_1' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_1} \dashv [\Sigma]\ n_1 \implies \\
 & & \qquad \exists n_2.\ e_2' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_2} \dashv [\Sigma]\ n_2 \wedge \triangleright \mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}(n_1,n_2) \\
 & & \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B, \quad e_i' = \mathbf{unfold}(e_i),\ i = 1,2 \\
 & & \\
\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1,n_2) & \triangleq & \forall H_1,H_2.\mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1,H_2) \implies \mathbf{O}(\langle H_1,n_1\rangle,\langle H_2,n_2\rangle) \\
 & & \\
\mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1,H_2) & \triangleq & \forall u_1,u_2.\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(u_1,u_2) \implies \mathbf{O}(\langle H_1,u_1\rangle,\langle H_2,u_2\rangle) \\
 & & \\
\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(\downarrow v_1,\downarrow v_2) & \triangleq & \mathbf{V}[\![\ A\ ]\!]^{\lessapprox}(v_1,v_2) \\
\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(u_1,u_2) & \triangleq & c : A_c \to\ B_c \in \Sigma\ \wedge (c\text{-stuck}(H_i))_{i=1,2} \wedge \triangleright \mathbf{V}[\![\ A_c\ ]\!]^{\lessapprox}(v_1,v_2) \wedge \\
 & & \qquad \forall w_1,w_2.\triangleright \mathbf{V}[\![\ B_c\ ]\!]^{\lessapprox}(w_1,w_2) \implies \triangleright \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1@{\downarrow}w_1,H_2@{\downarrow}w_2) \\
 & & \text{where } u_i = H_i \star c\ v_i,\ i = 1,2 \\
 & & \\
\mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1,\theta_2) & \triangleq & \forall (x:A) \in \Gamma.\ \mathbf{V}[\![\ A\ ]\!]^{\lessapprox}(\theta_1(x),\theta_2(x))
\end{array}
$$

Logical Relation

$$
\begin{array}{lcl}
\Gamma \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma]A & \triangleq & \theta_1,\theta_2 : \Gamma \vDash \cdot \Vdash \mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1,\theta_2) \implies \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(\theta_1(n_1),\theta_2(n_2)) \\
 & & \text{and similar for } \lessapprox^{val},\lessapprox^{nf},\lessapprox^{stk} \\
\Gamma \vdash n_1 \approx n_2 : [\Sigma]A & \triangleq & \Gamma \vdash n_1 \lessapprox n_2 : [\Sigma]A \times \Gamma \vdash n_2 \lessapprox n_1 : [\Sigma]A
\end{array}
$$

Figure 6.11: ELLA Logical Approximation

By treating the arguments to $\mathbf{T}[\![-]\!]^{\lessapprox}$ as $H@n$ for some configuration $\mathcal{C} = \langle H, n\rangle$, we may show the relation is closed under expansion.

Lemma 6.30. $\mathbf{T}[\![-]\!]^{\lessapprox}$ *is closed under expansion:*

1. $\langle H_1, n_1\rangle \longrightarrow \langle H_1', n_1'\rangle \Vdash \triangleright \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1'@n_1', n_2) \implies \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1@n_1, n_2)$

2. $\langle H_2, n_2\rangle \longrightarrow \langle H_2', n_2'\rangle \Vdash \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1, H_2'@n_2') \implies \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1, H_2@n_2)$

Proof. By Lemmas 6.26 and 6.7, and Corollary 6.28. □

Lemma 6.31. *Given H, H' such that c-stuck(H) and c-stuck(H'), let j denote the length of H. We have,*

1. $\triangleright^j \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}((H \circ H') \star c\ v, n) \Vdash \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H@(H' \star c\ v), n)$

2. $\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n, (H \circ H') \star c\ v) \Vdash \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n, H@(H' \star c\ v))$

Proof. By induction on H using Lemma 6.30. □

Definition 6.32. For handler stacks H_1, H_2 such that $(\cdot\ [\Sigma']\text{-}\ H_i : [\Sigma]A \multimap B)_{i=1,2}$, define *partial* logical stack approximation at $[\Sigma']B$, written $\mathbf{P}[\![\ [\Sigma]A \rightsquigarrow [\Sigma']B\ ]\!]^{\lessapprox}(H_1, H_2)$, as

$$\forall u_1, u_2.\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(u_1, u_2) \implies \mathbf{T}[\![\ [\Sigma']B\ ]\!]^{\lessapprox}(H_1@u_1, H_2@u_2)$$

and we have the open extension $\Gamma \vdash H_1 \lessapprox^{prt} H_2 : [\Sigma]A \rightsquigarrow [\Sigma']B$ defined by:

$$\forall \theta_1, \theta_2.\, \mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1, \theta_2) \implies \mathbf{P}[\![\ [\Sigma]A \rightsquigarrow [\Sigma']B\ ]\!]^{\lessapprox}(\theta_1(H_1), \theta_2(H_2))$$

Using partial stack approximation we can decompose a complex term-level approximation into two simpler proof obligations.

Lemma 6.33 (Term Decomposition)**.**

$$\mathbf{P}[\![\ [\Sigma]A \rightsquigarrow [\Sigma']B\ ]\!]^{\lessapprox}(H_1, H_2), \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1, n_2) \Vdash \mathbf{T}[\![\ [\Sigma']B\ ]\!]^{\lessapprox}(H_1@n_1, H_2@n_2)$$

Proof. By definition using Corollary 6.28. □

We can simplify relational reasoning about partial stacks since it suffices to consider their behaviour only with respect to values (of the appropriate type), and commands handled within the partial handler stacks. The following lemmas capture this simplification.

First, stacks which do not handle any effects are related as partial stacks if they agree for all related values and are stuck for all commands in the ambient ability.

Lemma 6.34. *For all handler stacks H_1, H_2 such that $(\cdot \,[\Sigma]\text{–}\, H_i : [\Sigma]A \multimap B)_{i=1,2}$, and propositions Φ, the following inference rule is derivable:*

$$\frac{\forall c : A_1 \to B_1 \in \Sigma,\, c\text{-stuck}(H_i)_{i=1,2} \qquad \Phi, \mathbf{V}[\![\, A \,]\!]^{\lessapprox}(v_1, v_2) \Vdash \mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}(H_1 @ \downarrow v_1, H_2 @ \downarrow v_2)}{\Phi \Vdash \mathbf{P}[\![\, [\Sigma]A \rightsquigarrow [\Sigma]B \,]\!]^{\lessapprox}(H_1, H_2)}$$

Proof. By Löb induction assume $\triangleright \mathbf{P}[\![\, [\Sigma]A \rightsquigarrow [\Sigma]B \,]\!]^{\lessapprox}(H_1, H_2)$ holds. By definition, we assume (i) $\mathbf{U}[\![\, [\Sigma]A \,]\!]^{\lessapprox}(u_1, u_2)$ and require to show $\mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}(H_1 @ u_1, H_2 @ u_2)$. Now consider the forms of u_1 and u_2 such that (i) holds.

Case u_1 and u_2 are values. Then the result follows by our premiss for plugging related values into the stacks.

Case $u_1 = H_1' \star c\ v_1$ and $u_2 = H_2' \star c\ v_2$ for some $c : A_c \to B_c \in \Sigma$.

We require to show

$$\begin{aligned}
&\mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}(H_1 @ (H_1' \star c\ v_1), H_2 @ (H_2' \star c\ v_2)) \\
\text{i.e.}\quad &\triangleright^{|H_1|} \mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}((H_1 \circ H_1') \star c\ v_1, (H_2 \circ H_2') \star c\ v_2) && \text{(by Lemma 6.31)} \\
\text{i.e.}\quad &\mathbf{U}[\![\, [\Sigma]B \,]\!]^{\lessapprox}((H_1 \circ H_1') \star c\ v_1, (H_2 \circ H_2') \star c\ v_2) && \text{(by mono \& Lemma 6.29(2))}
\end{aligned}$$

By definition of $\mathbf{U}[\![\, [\Sigma]B \,]\!]^{\lessapprox}$, given $\triangleright \mathbf{V}[\![\, B_c \,]\!]^{\lessapprox}(w_1, w_2)$, it remains to show:

$$\begin{aligned}
&\triangleright \mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}((H_1 \circ H_1') @ \downarrow w_1, (H_2 \circ H_2') @ \downarrow w_2) \\
\text{i.e.}\quad &\triangleright \mathbf{T}[\![\, [\Sigma]B \,]\!]^{\lessapprox}(H_1 @ (H_1' @ \downarrow w_1), H_2 @ (H_2' @ \downarrow w_2))
\end{aligned}$$

Applying Lemma 6.33 and the Löb induction hypothesis, we are left to prove:

$$\triangleright \mathbf{T}[\![\, [\Sigma]A \,]\!]^{\lessapprox}(H_1' @ \downarrow w_1, H_2' @ \downarrow w_2)$$

which follows from (i).

□

Next, we define a normal form property for handler stacks that pertains to a specified command (Definition 6.35). For handler stacks which handle a collection of effects, we use the normal form property to express progress for command invocations (Lemma 6.37).

Definition 6.35. A pair of handler stacks (H_1, H_2) are normal with respect to a command $c : A_c \to B_c \in \Sigma$ at type $[\Sigma]A$, written $c\text{-normal}(H_1, H_2, [\Sigma]A)$, if $c\text{-stuck}(H_1)$, $c\text{-stuck}(H_2)$ and for all $\triangleright \mathbf{V}[\![\, B_c \,]\!]^{\lessapprox}(w_1, w_2)$,

$$\triangleright \mathbf{T}[\![\, [\Sigma]A \,]\!]^{\lessapprox}(H_1 @ \downarrow w_1, H_2 @ \downarrow w_2).$$

Remark 6.36. Do not confuse the above c-normal property on *pairs of handler stacks* with the Σ-normal property on *normal forms*. The latter asserts a normal form is either a value or stuck with respect to a command in Σ, whereas the former specifies a notion of observable progress for c-stuck handler stacks.

Lemma 6.37. *Given handler stacks H_1, H_2 such that:*

1. $\mathbf{V}[\![A]\!]^{\lessapprox}(v_1, v_2) \implies \mathbf{T}[\![[\Sigma]B]\!]^{\lessapprox}(H_1 @ \downarrow v_1, H_2 @ \downarrow v_2))$*;*
2. $\forall c : A_c \to B_c \notin \Delta(\varnothing)$*,* c-stuck(H_1) *and* c-stuck(H_2)*;*
3. $\forall c : A_c \to B_c \in \Delta(\varnothing), \triangleright \mathbf{V}[\![A_c]\!]^{\lessapprox}(v_1, v_2)$ *and* H_1', H_2' *such that* c-normal$(H_1', H_2', [\Delta(\Sigma)]A)$ *then*
$$\mathbf{T}[\![[\Sigma]B]\!]^{\lessapprox}(H_1 @ (H_1' \star c\ v_1), H_2 @ (H_2' \star c\ v_2));$$

then $\mathbf{P}[\![[\Delta(\Sigma)]A \rightsquigarrow [\Sigma]B]\!]^{\lessapprox}(H_1, H_2)$.

Proof. There are three distinct cases to consider based on the particular normal forms plugged into the contexts. Condition (1) handles the value normal forms. Condition (3) handles the case for stuck terms where the command c is in Δ. For all other commands the proof is identical to the stuck term case of Lemma 6.34. □

The following result regarding pattern matching is important for establishing $\lessapprox^{trm}$ compatible with the term formers of ELLA.

Lemma 6.38. *Given $p : A \dashv \Gamma$, and v_1 and θ_1 such that $p : A \longleftarrow v_1 \dashv \theta_1$ then for all v_2:*

1. $\mathbf{V}[\![A]\!]^{\lessapprox}(v_1, v_2) \Vdash \exists \theta_2 . p : A \longleftarrow v_2 \dashv \theta_2 \wedge \mathbf{G}[\![\Gamma]\!]^{\lessapprox}(\theta_1, \theta_2)$

Similarly, given $r : \langle \Delta \rangle A \dashv [\Sigma]\ \Gamma$, and u_1 and θ_1 such that $r : \langle \Delta \rangle A \longleftarrow u_1 \dashv [\Sigma]\ \theta_1$ then for all u_2:

2. $\mathbf{U}[\![[\Delta(\Sigma)]A]\!]^{\lessapprox}(u_1, u_2) \Vdash \exists \theta_2 . r : \langle \Delta \rangle A \longleftarrow u_2 \dashv [\Sigma]\ \theta_2 \wedge \triangleright \mathbf{G}[\![\Gamma]\!]^{\lessapprox}(\theta_1, \theta_2)$

Proof. By simultaneous induction on the pattern matching derivations $p : A \longleftarrow v_1 \dashv \theta_1$ and $r : \langle \Delta \rangle A \longleftarrow u_1 \dashv [\Sigma]\ \theta_1$. We consider the interesting case of a request pattern below.

Case M-COMMAND: $\langle c\ p \to z \rangle : \langle \Delta \rangle A \longleftarrow H_1 \star c\ v_1 \dashv [\Sigma]\ \theta_1 [\uparrow^C (x, () \mapsto H_1 @ \downarrow x)/z]$

Given $\mathbf{U}[\![[\Delta(\Sigma)]A]\!]^{\lessapprox}(H_1 \star c\ v_1, u_2)$, we require to prove:

$$\exists \theta_2 . r : \langle \Delta \rangle A \longleftarrow u_2 \dashv [\Sigma]\ \theta_2 \wedge \triangleright \mathbf{G}[\![\Gamma]\!]^{\lessapprox}(\theta_1, \theta_2) \quad (\star)$$

Appeal to the inductive hypothesis for (1) to obtain substitutions that are approximate for the command argument, *i.e.* for $c : A_c \to B_c$ we have,

$$\mathbf{U}[\![\ [\Delta(\Sigma)]A\ ]\!]^{\lessapprox}(H_1 \star c\ v_1, H_2 \star c\ v_2) \quad \text{(i)}$$

from which we obtain values $\triangleright\mathbf{V}[\![\ A_c\ ]\!]^{\lessapprox}(v_1, v_2)$ matching pattern p. Furthermore, by Proposition 6.4, we deduce there exists θ_2 such that $p : A \longleftarrow v_2 \dashv \theta_2$ and hence (ii) $\triangleright\mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1, \theta_2)$.

Instantiate $(\star)$ with the substitution $\theta_2[\uparrow^C(x, () \mapsto H_2@\downarrow x)/z]$. Using (ii) our obligation reduces to showing:

$$\mathbf{V}[\![\ \{\langle\iota\rangle B_c, \langle\iota\rangle\mathbf{unit} \to [\Delta(\Sigma)]A\}\ ]\!]^{\lessapprox}(\uparrow^C(x, () \mapsto H_1@\downarrow x), \uparrow^C(x, () \mapsto H_2@\downarrow x))$$

By definition this reduces to showing for all $\mathbf{V}[\![\ B_c\ ]\!]^{\lessapprox}(w_1, w_2)$:

$$\triangleright\mathbf{T}[\![\ [\Delta(\Sigma)]A\ ]\!]^{\lessapprox}(H_1@\downarrow w_1, H_2@\downarrow w_2)$$

which follows from (i).

□

Lemma 6.39. *Logical approximation is closed under the compound term formers (Figure 6.12).*

Proof. By Lemmas 6.29, 6.33 and 6.34. □

We have shown that logical approximation is closed under all term formers in the language. From this fact, we are able to establish the fundamental property of logical relations. That is, the relation is reflexive.

Lemma 6.40. *Logical approximation is reflexive:*

1. $\Gamma \vdash w \lessapprox^{val} w : [\Sigma]A$;
2. $\Gamma \vdash n \lessapprox^{trm} n : [\Sigma]A$;
3. $\Gamma \vdash H \lessapprox^{stk} H : [\Sigma]A$;
4. $\Gamma \vdash H \lessapprox^{prt} H : [\Sigma]A \rightsquigarrow [\Sigma']B$;
5. $\Gamma \vdash u \lessapprox^{nf} u : [\Sigma]A$.

C-APP

$$\frac{\mathbf{V}[\![\ \{\overline{\langle\Delta\rangle A} \to [\Sigma]B\}\ ]\!]^{\lessapprox}(v_1, v_2) \qquad \mathbf{T}[\![\ [\Delta_1(\Sigma)]A_1\ ]\!]^{\lessapprox}(n_1, n_2) \qquad \mathbf{T}[\![\ [\Delta_2(\Sigma)]A_2\ ]\!]^{\lessapprox}(n_1', n_2')}{\mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}(\downarrow(v_1\ n_1\ n_1'), \downarrow(v_2\ n_2\ n_2'))}$$

C-COMMAND

$$\frac{c : A_c \to B_c \in \Sigma \quad (c\text{-stuck}(H_i))_{i=1,2} \quad \mathbf{P}[\![\ [\Sigma']B_c \rightsquigarrow [\Sigma]B\ ]\!]^{\lessapprox}(H_1, H_2) \quad \mathbf{V}[\![\ A\ ]\!]^{\lessapprox}(v_1, v_2)}{\mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}(H_1 \star c\ v_1, H_2 \star c\ v_2)}$$

C-LET

$$\frac{\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1, n_1') \quad x : A \vdash n_2 \lessapprox^{trm} n_2' : [\Sigma]B}{\mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}(\mathbf{let}\ x : A = n_1\ \mathbf{in}\ n_2, \mathbf{let}\ x : A = n_1'\ \mathbf{in}\ n_2')}$$

Figure 6.12: $\mathbf{T}[\![\ -\ ]\!]$ closed under the compound term formers

Proof. By simultaneous induction on the derivations:

$$\Gamma\,[\Sigma]\vdash w : A \qquad \Gamma\,[\Sigma]\vdash n : A \qquad \Gamma\,[\varnothing]\vdash H : [\Sigma]A \multimap \mathbf{unit}$$

$$\Gamma\,[\Sigma']\vdash H : [\Sigma]A \multimap B \qquad \Gamma\,[\Sigma]\vdash u : A$$

using Lemma 6.39 for (2). Note that, for $\Gamma \vdash \uparrow e \lessapprox^{val} \uparrow e : \{C\}$ in (1) we use Lemma 6.38.

Part (4) is by Lemma 6.29(2), term decomposition (Lemma 6.33), Lemma 6.39, and the inductive hypothesis for (2). Part (5) follows from the inductive hypotheses for (1) and (4). For (3), we perform a further case analysis on the possible normal forms and appeal to the inductive hypothesis for (2). The case for $H = Id$ is immediate by definition, we consider the other cases below whilst eliding approximate substitutions since they are just threaded throughout.

Case $H = H' \circ (x : A.n)$.

There are two cases to consider depending on the possible normal forms. Suppose we have value normal forms: $\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(w_1, w_2)$. Then we require to show:

$$\mathbf{O}(\langle H, w_1\rangle, \langle H, w_2\rangle)$$

which follows by Lemma 6.26, and the inductive hypotheses for H' and (2).

Now suppose we have normal forms (i) $\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1 \star c\ v_1, H_2 \star c\ v_2)$. We

require to show:

$$\mathbf{O}(\langle H, H_1 \star c\ v_1\rangle, \langle H, H_2 \star c\ v_2\rangle)$$

i.e. $\triangleright\mathbf{O}(\langle H', (x : A.n) \bullet H_1 \star c\ v_1\rangle, \langle H', (x : A.n) \bullet H_2 \star c\ v_2\rangle)$ by Lemma 6.26

by the inductive hypothesis for H' it suffices to show (assuming $c : A_c \to B_c \in \Sigma$) for all $\triangleright\mathbf{V}[\![\ B_c\ ]\!]^{\lessapprox}(w_1, w_2)$,

$$\triangleright\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(((x : A.n) \bullet H_1)@{\downarrow}w_1, ((x : A.n) \bullet H_2)@{\downarrow}w_2)$$

i.e. $\triangleright\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}((Id \circ (x : A.n))@(H_1@{\downarrow}w_1), (Id \circ (x : A.n))@(H_2@{\downarrow}w_2))$

by term decomposition (Lemma 6.33) and (i) it remains to show:

$$\triangleright\mathbf{P}[\![\ [\Sigma]A \rightsquigarrow [\Sigma]B\ ]\!]^{\lessapprox}(Id \circ (x : A.n), Id \circ (x : A.n))$$

and the result follows by the inductive hypothesis for (4).

Case $H = H' \circ (\mathbf{rec}\ f\{e\} : \{C\}, \Box\ n)$ where $C = \overline{\langle\Delta\rangle A} \to [\Sigma]B$.

Again, we consider the possible normal forms paired with H. Consider the case when $\mathbf{U}[\![\ [\Delta_1(\Sigma)]A_1\ ]\!]^{\lessapprox}(H_1 \star c\ v_1, H_2 \star c\ v_2)$ s.t. $c : A_c \to B_c \in \Delta_1$. We have to show:

$$\begin{array}{ll} & \mathbf{O}(\langle H, H_1 \star c\ v_1\rangle, \langle H, H_2 \star c\ v_2\rangle) \\ \text{i.e.} & \triangleright\mathbf{O}(\langle H_1', n\rangle, \langle H_2', n\rangle) \qquad \text{by Lemma 6.26} \\ & \text{where } H_i' = H' \circ (\mathbf{rec}\ f\{e\} : \{C\}, (H_i \star c\ v_i)\ \Box) \end{array}$$

by the inductive hypothesis for (2) we require to show:

$$\triangleright\mathbf{S}[\![\ [\Delta_2(\Sigma)]A_2\ ]\!]^{\lessapprox}(H_1', H_2')$$

That is, given $\triangleright\mathbf{U}[\![\ [\Delta_2(\Sigma)]A_2\ ]\!]^{\lessapprox}(u_1, u_2)$ we must show

$$\triangleright\mathbf{O}(\langle H_1', u_1\rangle, \langle H_2', u_2\rangle)$$

If $u_i = w_i$ or $u_i = H_i'' \star c'\ w_i$ for $c' : A_c' \to B_c' \notin \Delta_2$ then we proceed similarly to the previous case. Otherwise, $c' : A_c' \to B_c' \in \Delta_2$ and we apply Lemma 6.26 using (S-HDLCMD-SND) with Proposition 6.4. The result then follows by inductive hypothesis for H' and (2).

Case $H = H' \circ (\mathbf{rec}\ f\{e\} : \{C\}, u\ \Box)$.

By similar reasoning to the above two cases.

□

From Lemmas 6.40 and 6.39 we get closure of our logical relation under VSCs.

Lemma 6.41. *If* $\Gamma_1 \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma_1]A$ *and* $\Gamma_0\,[\Sigma]\text{–}\ \mathcal{N} : \langle\langle\Gamma_1\,[\Sigma_1]\text{–}\ A\rangle\rangle^{VSC}B$ *then*

$$\Gamma_1 \vdash \mathcal{N}[n_1] \lessapprox^{trm} \mathcal{N}[n_2] : [\Sigma_1]B.$$

Component Relations

$$
\begin{array}{lcl}
\mathbf{V}[\![\ \tau\ ]\!]^{\lesssim}(v_1,v_2) & \triangleq & v_1 \equiv v_2 \\
\mathbf{V}[\![\ \{C\}\ ]\!]^{\lesssim}(\uparrow e_1,\uparrow e_2) & \triangleq & \forall \overline{u}, n_1.\, e_1' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u} \dashv [\Sigma]\ n_1 \implies \\
 & & \qquad \exists n_2.\ e_2' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u} \dashv [\Sigma]\ n_2 \wedge \triangleright \mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lesssim}(n_1,n_2) \\
 & & \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B, \quad e_i' = \mathbf{unfold}(e_i),\ i = 1,2 \\
 & & \\
\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(n_1,n_2) & \triangleq & \forall H.\, \mathbf{O}(\langle H, n_1\rangle, \langle H, n_2\rangle) \\
 & & \\
\mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(H_1,H_2) & \triangleq & \forall u.\, \mathbf{O}(\langle H_1, u\rangle, \langle H_2, u\rangle) \\
 & & \\
\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(\downarrow v_1,\downarrow v_2) & \triangleq & \mathbf{V}[\![\ A\ ]\!]^{\lesssim}(v_1,v_2) \\
\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(u_1,u_2) & \triangleq & c : A_c \to\ B_c \in \Sigma\ \wedge (c\text{-stuck}(H_i))_{i=1,2} \wedge \triangleright \mathbf{V}[\![\ A_c\ ]\!]^{\lessapprox}(v_1,v_2) \wedge \\
 & & \qquad \forall w.\, \triangleright \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1 @ \downarrow w, H_2 @ \downarrow w) \\
 & & \text{where } u_i = H_i \star c\ v_i,\ i = 1,2
\end{array}
$$

Logical Relation

$$
\begin{array}{lcl}
\Gamma \vdash n_1 \lesssim^{trm} n_2 : [\Sigma]A & \triangleq & \theta : \Gamma \vDash \cdot \Vdash \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(\theta(n_1),\theta(n_2)) \\
 & & \text{and similar for } \lesssim^{val}, \lesssim^{nf}, \lesssim^{stk} \\
\Gamma \vdash n_1 \sim n_2 : [\Sigma]A & \triangleq & \Gamma \vdash n_1 \lesssim n_2 : [\Sigma]A \times \Gamma \vdash n_2 \lesssim n_1 : [\Sigma]A
\end{array}
$$

Figure 6.13: ELLA Applicative Approximation

6.3.3 Applicative Approximation

The applicative approximation relation is presented in Figure 6.13. We define closed value, term, stack and normal form relations. The relation on values is defined by recursion on the type structure. This definition of applicative approximation is contained within CIU approximation.

Lemma 6.42.

$$\frac{\Gamma \vdash n_1 \lesssim^{CIU} n_2 : [\Sigma]A}{\Gamma \vdash n_1 \lesssim n_2 : [\Sigma]A}$$

Proof. First prove:

$$\Vdash \forall H_1, H_2, n_1, n_2.(H_1 @ n_1 \Downarrow \downarrow() \implies H_2 @ n_2 \Downarrow \downarrow()) \implies \mathbf{O}(\langle H_1, n_1\rangle, \langle H_2, n_2\rangle) \quad (\star)$$

by Löb induction and Lemma 6.5.

For the main result, assume given a Γ-closing substitution θ and handler stack H. In the CIU approximation assumption, set the context $\mathcal{P}$ to be $H\langle\langle @ \rangle\rangle\langle\langle \theta - \rangle\rangle$ and rewrite using Lemma 6.19. The result follows by $(\star)$. □

Lemma 6.43. *Observational approximation implies applicative approximation.*

Proof. By Lemmas 6.10, 6.17 and 6.42. □

Occasionally, we can simplify the proof of an example approximation by using the applicative notion rather than the logical one. We prove that both the applicative and logical notions coincide. Such a correspondence permits us to reuse the properties established for the logical relation in proofs which use the applicative notion.

First, it is straightforward to show that logical approximation implies applicative approximation.

Lemma 6.44. $\Gamma \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma]A \implies \Gamma \vdash n_1 \lesssim^{trm} n_2 : [\Sigma]A$

Proof. By Lemma 6.40(3). □

To show the converse, we again go via a "transitivity" property, reminiscent of Howe's method, which we introduced in Chapter 4. There is a slight difference in the approach taken here since the introduction of step-indexing invalidates transitivity at a fixed index. However, transitivity does hold at the limit. That is, for the relations quantified over all step-indices. As a stepping stone to this result, we prove a transitivity-esque result which relies on the applicative approximation holding for all step-indices.

Lemma 6.45. *The following transitivity properties hold:*

1. *If* $\mathbf{V}[\![A]\!]^{\lessapprox}(v_1, v_2)$ *and* $\vdash v_2 \lesssim^{val} v_3 : [\varnothing]A$ *then* $\mathbf{V}[\![A]\!]^{\lessapprox}(v_1, v_3)$
2. *If* $\mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}(n_1, n_2)$ *and* $\vdash n_2 \lesssim^{trm} n_3 : [\Sigma]A$ *then* $\mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}(n_1, n_3)$
3. *If* $\mathbf{S}[\![[\Sigma]A]\!]^{\lessapprox}(H_1, H_2)$ *and* $\vdash H_2 \lesssim^{stk} H_3 : [\Sigma]A$ *then* $\mathbf{S}[\![[\Sigma]A]\!]^{\lessapprox}(H_1, H_3)$
4. *If* $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}(u_1, u_2)$ *and* $\vdash u_2 \lesssim^{nf} u_3 : [\Sigma]A$ *then* $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}(u_1, u_3)$

Proof. Part (1), is by case analysis on A using (2) for when $A = \{C\}$. For (2) & (3), unfold definitions and apply Lemma 6.25. Part (4) is by definition using (1) and (2). □

Lemma 6.46. *Applicative approximation implies logical approximation:*

$$\textit{If } \Gamma \vdash n_1 \lesssim^{trm} n_2 : [\Sigma]A \textit{ then } \Gamma \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma]A$$

Proof. By Lemmas 6.40 and 6.45. □

We establish analogues of the reasoning principles we developed for logical approximation. These are useful for proofs that go via the applicative approximation (see Chapter 7).

Lemma 6.47. *The following property hold for any type A and ability Σ:*

1. $\mathbf{V}[\![A]\!]^{\lesssim}(v_1, v_2) \Vdash \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(\downarrow v_1, \downarrow v_2)$
2. $\mathbf{U}[\![[\Sigma]A]\!]^{\lesssim}(u_1, u_2) \Vdash \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(u_1, u_2)$

Proof. By Lemmas 6.40, 6.45 and 6.29. □

Lemma 6.48. $\mathbf{T}[\![-]\!]^{\lesssim}$ *is closed under expansion:*

1. $\langle H_1, n_1\rangle \longrightarrow \langle H_1', n_1'\rangle \Vdash \triangleright \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(H_1' @ n_1', n_2) \implies \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(H_1 @ n_1, n_2)$
2. $\langle H_2, n_2\rangle \longrightarrow \langle H_2', n_2'\rangle \Vdash \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(n_1, H_2' @ n_2') \implies \mathbf{T}[\![[\Sigma]A]\!]^{\lesssim}(n_1, H_2 @ n_2)$

Proof. By Lemmas 6.26 and 6.7, and Corollary 6.28. □

Definition 6.49. For handler stacks H_1, H_2 such that $(\cdot\, [\Sigma']\text{⊢}\ H_i : [\Sigma]A \multimap B)_{i=1,2}$, define *partial* applicative stack approximation at $[\Sigma']B$, written $\mathbf{P}[\![[\Sigma]A \rightsquigarrow [\Sigma']B]\!]^{\lesssim}(H_1, H_2)$, as

$$\forall u.\cdot\, [\Sigma]\text{⊢}\ u : A \wedge \Sigma\text{-normal}(u) \implies \mathbf{T}[\![[\Sigma']B]\!]^{\lesssim}(H_1 @ u, H_2 @ u)$$

Lemma 6.50 (Term Decomposition)**.**

$$\mathbf{P}[\![[\Sigma]A \rightsquigarrow [\Sigma']B]\!]^{\lesssim}(H_1, H_2) \Vdash \mathbf{T}[\![[\Sigma']B]\!]^{\lesssim}(H_1 @ n, H_2 @ n)$$

Proof. By Lemmas 6.40 and 6.45 using Corollary 6.28. □

Lemma 6.51. *Given handler stacks H_1, H_2 such that:*

1. $\forall v : A.\, \mathbf{T}[\![[\Sigma]B]\!]^{\lesssim}(H_1 @ \downarrow v, H_2 @ \downarrow v)$*;*
2. $\forall c : A_c \to B_c \notin \Delta(\varnothing)$*,* c-stuck(H_1) *and* c-stuck(H_2)*;*

3. $\forall c : A_c \to B_c \in \Delta(\varnothing)$, *closed value* v *of type* A, *and* H' *such that* c-stuck(H') *then*

$$\mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lesssim}(H_1@(H' \star c\ v), H_2@(H' \star c\ v));$$

then $\mathbf{P}[\![\ [\Delta(\Sigma)]A \rightsquigarrow [\Sigma]B\ ]\!]^{\lesssim}(H_1, H_2)$.

Proof. Assume given $\cdot\,[\Sigma]\text{–}\ u : A$ such that $\Delta(\Sigma)$-normal(u). Perform case analysis on the derivation of u. When $u = \downarrow v$ and $u = H' \star c\ v$ with $c : A_c \to B_c \in \Delta$, the result follows by (1) and (3), respectively. Finally, for stuck terms where $c : A_c \to B_c \notin \Delta$, the proof follows analogous reasoning to the corresponding case in Lemma 6.34. □

6.4 ELLA's Context Lemma

Lemmas 6.10 and 6.17 give the forward direction of the following context lemma for ELLA.

Lemma 6.52 (Context Lemma)**.**

$$\Gamma \vdash n_1 \lesssim^{VSC} n_2 : [\Sigma]A \ \textit{ if and only if } \ \Gamma \vdash n_1 \lesssim^{CIU} n_2 : [\Sigma]A$$

Now, we establish the remaining implication of the triangle. Namely that, from Lemma 6.41, logical approximation is sound with respect to observational approximation.

Lemma 6.53. *If* $\Gamma \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma]A$ *then* $\Gamma \vdash n_1 \lesssim^{VSC} n_2 : [\Sigma]A$

Finally, we arrive at the triangulation result for ELLA.

Lemma 6.54 (Handler Stack Triangulation)**.** *Observational, applicative frame and logical approximation coincide.*

Proof. Using Lemmas 6.43, 6.46 and 6.53. □

6.5 Discussion

We discuss the key influences from the literature behind the design of ELLA and its reasoning principles. We also report on the progress made towards its formalisation in AGDA.

6.5.1 ELLA and FRANK

This chapter has introduced ELLA which aims to capture the essence of effectful programming in FRANK. We can view ELLA as a target calculus for well-typed FRANK programs, where instantiation of polymorphic definitions yields particular monomorphic variants. Although we have only described binary handlers in ELLA, a generalisation to an n-ary version would involve a straightforward adaption of the frames for operators, and the $\star$-HDLCMD-FST and $\star$-APPV-FST rules. We leave the specifics for future work.

6.5.2 ELLA Reasoning Principles

Regarding the paricular reasoning principles we have established, they are based heavily on the properties Biernacki et al. [2018] prove for their core effect calculus, $\lambda^{\mathsf{H/L}}$. While their calculus does not support binary handlers and utilises a separate handling construct, their development is more general than ours in other ways. First, $\lambda^{\mathsf{H/L}}$ supports effect polymorphism which yields a more flexible $\mathbf{U}[\![-]\!]$ relation, capable of relating arbitrary expressions, rather than just syntactically identical normal forms of ELLA's corresponding relation. Second, $\lambda^{\mathsf{H/L}}$ supports the lift operation, previously mentioned in Section 3.11, which extends an effect row with a specified effect label. In contrast, ELLA does not support FRANK's mask operation, or adaptors more generally. Finally, $\lambda^{\mathsf{H/L}}$ admits subtyping of both value and effect types; the latter is usually called "effect subtyping" [Biernacki and Polesiuk, 2015], or "subeffecting" [Lindley et al., 2017]). Effect subtyping and adaptors enable reasoning not possible in our current ELLA formalism by allowing terms to appear in more permissive ambient environments. We discuss this point further in Section 7.3.

6.5.3 Formalisation Progress

Our formalisation in AGDA of this chapter is only partially complete. Thus far we have formalised the calculus of Figure 6.1 using the generic universe of syntaxes with binding due to Allais et al. [2018]. Currently, ELLA terms are well-*sorted*, but not well-typed. Well-sortedness ensures that terms are well-scoped and either checkable or inferrable, by construction. We have yet to define the logical and applicative approximations in Figures 6.11 and 6.13 using $\mathtt{FOL}^{\mu\triangleright}$ although we conjecture their encodings should be close to our description in Appendix B. In contrast, the definition

for $\lambda^{\rightarrow}_{FG}$ logical approximation is defined by a set of mutually recursive functions — one for each of values, terms and stacks — acting on well-typed and well-scoped syntax. Such a mutually recursive collection of definitions would not be permitted in the case of ELLA since it would not satisfy the AGDA termination checker [2]. The only novel aspect is pattern matching for interpreting $\{C\}$ types, in lieu of straightforward quantification over values of the argument type.

The generic framework by Allais et al. [2018] defines a traversal over a universe of syntaxes with binding. Then, a user of the library only has to define their calculus using the constructors of the universe, which automatically yields various pre-defined semantics: renaming, substitution, and their basic properties. Choosing this framework is particularly fruitful if the calculus in question is in a state of flux, and subject to change. We can alter the constructs within the language simply by altering the grammar without concern for knock-on effects to syntactic traversals or fusion of semantics.

6.5.4 Handler Stacks

We have chosen to establish our results by generalising frame stacks to incorporate handlers, and the handling of effects. Thus, we generalise typical notions of biorthogonality / $\top\top$-lifting from values to effectful normal forms; the approximation relation on terms being defined by the $\top\top$-lifting of $\mathbf{V}[\![-]\!]$ over $\mathbf{S}[\![-]\!]$.

A natural question to ask is whether all this complexity is truly needed: would an applicative notion similar to Milner's [1977] original, extended to account for stuck terms, suffice? Certainly, all our programs are morally *purely functional* since handlers give an interpretation to all effects within an ELLA program. In other words, ELLA does not contain any features (*e.g.* local state [Koutavas et al., 2010]) which preclude a big-step semantics-style development similar to the one we presented for $\lambda^{\rightarrow}_{FG}$ in Section 4.2. We do not explore this alternative for a few reasons. First, we believe it would not yield as useful reasoning principles, since ours, *e.g.* Lemmas 6.33 and 6.37, fundamentally rely on biorthogonality. In particular, the ability to reason about related handler stacks would not be possible in a big-step formulation because we would not have the $\mathbf{P}[\![-]\!]$ or $\mathbf{S}[\![-]\!]$ relations. Their absence would impact the usefulness of our current definition of stuck terms, $\mathbf{U}[\![-]\!]$, whose main proof obligation often relies on being able to 'rewrite' configurations using Lemma 6.27 and its Corollary 6.28.

[2] In particular, $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}$, while certainly contractive, appeals to $\mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}$ which in turn appeals to $\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}$ through $\mathbf{S}[\![-]\!]$.

Hence, we would no longer be able to enclose effectful terms into larger contexts using $\mathbf{S}[\![-]\!]$ to produce **unit**-valued pure programs, preventing us from using the simple basic observable relation $\mathbf{O}$. Second, basing our approximation relations on a big-step semantics à la Section 4.2, or even a small-step semantics, would expose step-indexing arithmetic in the reduction relations, *e.g.* Ahmed [2006], undoing our attempts to hide their presence with the $\triangleright$ modality. Third, it is well-known that Milner's result does not scale to more exotic calculi, *e.g.* local state [Stark, 1994; Koutavas et al., 2010]. In our setting, we would run into difficulties if state were built-in and not given an interpetation by handlers, *e.g.* ML-style references in FRANK; it remains an open question how to incorporate the algebraic account of local state [Plotkin and Power, 2002] with effect handlers. Thus, to support such extensions, we require to move to the CIU theorem [Mason and Talcott, 1991] and define our approximation relations using handler stacks.

Our presentation of biorthogonality is inspired by Pitts and Stark [1998] who introduce frame stacks to provide a structurally inductive notion of termination (see § 3 *loc. cit.*). Johann et al. [2010] have employed biorthogonality using frame stacks to prove a CIU theorem for call-by-name PCF with polymorphism and algebraic effects. Their metatheory is parameterised by a basic observation relation on ground type computations. This relation describes the particular collection of effects available and their semantics. The operational semantics is defined by reduction of programs to computation trees as introduced by Plotkin and Power [2001] (see Example 2.5). Their work inspired research into modal logics for behavioural equivalence where the modalities capture observations on programs [Matache, 2018; Simpson and Voorneveld, 2018] (see Section 6.6.2). Their context lemma result establishes that applicative contexts (in the Milner sense) suffice for observational approximation due to their call-by-name semantics and compositionality of their basic observation relation.

Stack-based machine semantics for algebraic effects and handlers have been presented before, *e.g.* the CEK-based machines [Felleisen and Friedman, 1987] by Hillerström and Lindley [2016] and Biernacki et al. [2019b], being a common implementation strategy for handlers. Our work represents the first use of stacks for relational reasoning. Additionally, our configurations are simpler in comparison to *loc. cit.* since we avoid an additional stack for accumulating bypassed handlers during command forwarding. Instead, we leverage continuation-passing style commands which store the accumulated frames. However, some of the added complexity in the case of Biernacki et al. is to account for effect coercions and local effects which we do not yet support

in our formalism. Extending our results to incorporate adaptors could take inspiration from Leijen's [2018] 'handler contexts' which formalise the semantics for *injecting* — Leijen's terminology for lift/mask — an effect into a row in a concise form which resembles our handler stacks (cf. recording the explicit nesting depth using Biernacki et al.'s [2018] 'n-freeness' relation).

6.6 Related Work

We discuss the broader context of reasoning about algebraic effects and handlers that we have not detailed so far.

6.6.1 Denotational Methods

Bauer and Pretnar [2014] present a fine-grained call-by-value calculus modelling the core of the EFF programming language. The calculus supports subtyping to address the poisoning problem, and effect *instances* to allow multiple occurrences of the same effect within a computation. Handlers are deep, distinct from functions, and may handle operations with respect to particular instance(s).

Remark 6.55. The current version of EFF (`https://www.eff-lang.org/`) is slightly different to the version in the published article by Bauer and Pretnar. In particular, the newest versions do not support effect instances.

Bauer and Pretnar [2014] provide both a big-step and small-step operational semantics for their calculus. The core calculus represents operations in continuation-passing style which is similar to ELLA stuck terms. The authors formalise their calculus, operational semantics and type soundness in Twelf [Pfenning and Schürmann, 1999].

They construct a denotational semantics for core EFF and prove that it agrees with the operational semantics (adequacy). The semantics require the solution of recursive domain equations to soundly interpret computation types.

Their solution relies crucially on the *minimal invariant* property due to Pitts [1996] which characterises unique, mixed (co)inductive solutions to domain equations. One can view the $\triangleright$ modality and contractivity as a way of achieving minimal invariance *syntactically* by explicitly representing the finite approximations within a stratified deductive system. Though, 'syntactic minimal invariance' typically refers to the technique introduced by Birkedal and Harper [1999], who prove minimality for a calculus with a single recursive type by a series of external inductive arguments.

The minimal solution yields recursion and induction principles for computation types. The recursion principle serves as the interpretation for handlers while the induction principles allows one to prove properties of programs. The induction principle admits a term decomposition lemma akin to our Lemma 6.50. Thus, in combination with their subtyping discipline, their method can be used to validate equations of the underlying algebraic theories of effects, which the authors demonstrate for global state. By adequacy, denotational equivalence implies observational equivalence but the authors do not prove the converse.

Kammar [2014] gives Gifford-style type- and effect-systems a general semantics using the theory of algebraic effects by considering annotation sets as the collection of operations which give rise to effects. Kammar develops MAIL, an extension of Levy's [2004] call-by-push-value to incorporate effect annotations. The general semantics is capable of validating a number of equivalences, including the equations associated to the theory as well as global properties of effects, *e.g.* discarding, copy, or swapping effectful terms. His work does not consider handlers for algebraic effects. It would be interesting to be able to prove the equivalences validated by Kammar's semantic framework in ELLA using our operational techniques. Extending Kammar's semantic account to incorporate ELLA or other calculi using algebraic effect handlers, would be an interesting direction for future work. Crucially, we would need to more seriously consider handlers which validate equations outright, since the optimisations Kammar considers rely on the effect theory. More generally, establishing a connection to Kammar's denotational approach and our operational account is worth investigating. Likewise, we have yet to investigate validating examples derived from Führmann's [2002] *effectoids*.

6.6.2 Logics

Pretnar [2010] presents a first-order logic, extended with fixed-point predicates, for a simply-typed calculus with algebraic effects and handlers. The calculus is based on Levy's [2004] call-by-push value. He derives various notions of program equivalence in his logic via translations of existing program logics, but does not consider observational equivalence.

Matache [2018] establishes observational equivalence using coinductive techniques for ECPS, a simply-typed call-by-value λ-calculus in continuation-passing style supporting algebraic effects and general recursion; termination being treated as an effect

with an explicit operation. The calculus does not track effects in the type system. Matache develops an *endogenous* logic $\mathcal{F}$, meaning a logical formula describes the properties of a single computation or program but is independent of the syntax of the programming language. This type of logic contrasts with Pretnar's *exogenous* logic (and $\mathtt{FOL}^{\mu\triangleright}$) where computations form the term language of the logic. The $\mathcal{F}$ logic induces a notion of program equivalence: $\mathcal{F}$-logical equivalence which, in conjunction with applicative bisimilarity using Howe's method, establishes a triangulation result for observational equivalence. Unlike our concrete contexts, Matache adopts a coinductive characterisation of contexts similar to Lassen [1998b]. Matache's investigation is inspired by the behavioural equivalence of Simpson and Voorneveld [2018] for a direct-style calculus, EPCF. She demonstrates an embedding of EPCF into ECPS by CPS translation but notes the unlikelihood of an embedding in the other direction due to ECPS being able to express control operators not expressible in EPCF.

The modal logics developed by Simpson and Voorneveld [2018] and Matache [2018] determine notions of program equivalence where programs are considered equivalent if and only if they satisfy the same set of formulas. In contrast, $\mathtt{FOL}^{\mu\triangleright}$ and Pretnar's [2010] logic do not determine a particular notion, but allow one to reason about multiple notions of program equivalence. However, neither logic has thus further been mechanised in a theorem prover.

6.6.3 Bisimulations and Howe's Method

Dal Lago et al. [2017] define applicative bisimilarity for a call-by-value computational λ-calculus with algebraic effects given by a monadic semantics. Their approach does not presuppose a particular collection of effects but instead is parametric with respect to a given computational monad T. Assuming T satisfies suitable conditions, the authors describe a lifting, called a *relator*, of a relation on a set X to a relation on $T\ X$. Relators express (abstractly) the observable aspects of the computational effects arising from the monad T. In this sense, it performs a similar function to the basic observable relation used by Johann et al. [2010] in their abstract logical relations-based account. Given a computational monad and associated relators, Dal Lago et al. can abstractly characterise applicative bisimilarity and prove its soundness with respect to observational equivalence using Howe's method. However, they do not prove completeness, noting that obtaining such a result, at least for *applicative* bisimilarity, would impact the range of effects captured by the technique.

Dal Lago and Gavazzo [2019] describe *normal form* bisimilarity in the above abstract setting, noting the improved reasoning techniques afforded by this kind of bisimulation in contrast to the applicative notion. In light of our discussion in Section 6.5.4, we conjecture a similar loss in reasoning power would be observed if we dispensed with biorthogonality and used Milner's [1977] original applicative notion instead.

It is an open research question to investigate a yet more general abstract theory for characterising Morris-style [1968] observational equivalence which encapsulates both (step-indexed) logical relations and bisimulation approaches. Hur et al. [2012] present a sound (but not complete) method combining bisimulations and Kripke logical relations, and subsequently scale the technique to support inter-language reasoning [Neis et al., 2015]. The approach combines the appealing aspects of the two techniques: using coinduction for recursive features (as in bisimulations) and state transition systems for local state (as in Kripke logical relations). Thus, their approach does not require step-indexing. A particular interesting thread to pursue would be connecting our logical relations method for ELLA with the complete bisimulations for delimited-control operators by Biernacki et al. [2019a]; taking advantage of the mutual simulation result between handlers and delimited-control operators [Kammar et al., 2013; Bauer and Pretnar, 2015].

Chapter 7

Examples

This chapter uses the techniques developed in the previous chapter to reason about program equivalences in ELLA. In proving concrete examples in this chapter, we elide some of the notation introduced in Chapter 6. In particular, conversions from inferrable terms to checkable terms are usually left implicit for the sake of brevity.

7.1 Encapsulation Example

We demonstrate that ELLA handlers can abstract their implementation, encapsulating some local state along the way. Concretely, we show that two implementations of a counter are observationally equivalent. The act of counting is represented by the `Tick` effect:

```
interface Tick = get : Unit -> Int
               | inc : Unit -> Unit
```

giving the option to retrieve the current counter value or increment the counter.

A handler `f` for the `Tick` effect has the following type signature:

```
{C} = {Int -> <Tick>Unit -> [∅]Int}
```

where the handled task returns a value of type **unit**. Such a handler may be used in a larger program without exposing the state to the client:

e' `= <tsk> () ↦ (rec f{e}:{C}) 0 (tsk ())`

where `tsk` has type $\{\overline{\langle \iota \rangle \textbf{unit}} \to [\Delta(\varnothing)]\textbf{int}\}$ [1], $\Delta = \iota, \text{Tick}$ and e is a particular implementation of the ticker handler. Given the following instantiations for e:

[1] We have used our syntactic sugar `tsk` $() \equiv$ `tsk` $() ()$ from Figure 6.5 in the above code snippet. We shall employ similar syntactic sugar throughout this chapter without further comment.

Listing 7.1: e_+

```
{ x <get -> k> ↦ f x (k x)
| x <inc -> k> ↦ f (x + 1) (k ())
| x      ()    ↦ x}
```

Listing 7.2: e_-

```
{ x <get -> k> ↦ f x (k (-x))
| x <inc -> k> ↦ f (x - 1) (k ())
| x      ()    ↦ -x}
```

the goal is to prove that for corresponding definitions of e'_+ and e'_-:

$$\cdot \vdash \{e'_+\} \approx \{e'_-\} : [\varnothing]\{\langle \text{Tick} \rangle \mathbf{unit}, \langle \iota \rangle \mathbf{unit} \to [\varnothing]\mathbf{int}\}$$

We shall prove only one direction of the equivalence since the other direction is analogous. Define the atomic relation $\mathsf{AbsV} : \mathbb{Z}, \mathbb{Z}$ by the following equation:

$$\mathsf{AbsV}(k_1, k_2) \triangleq k_1 \equiv -k_2$$

It suffices to show:

$$\mathbf{T}[\![\ [\Delta(\varnothing)]\mathbf{unit}\ ]\!]^{\lessapprox}(n_1, n_2), \mathsf{AbsV}(w_1, w_2) \Vdash \mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lessapprox}(\uparrow e_+\ w_1\ n_1, \uparrow e_-\ w_2\ n_2) \quad (P)$$

Our main result follows from Lemma 6.29 and (P) taking $w_1 = w_2 = 0$. It remains to prove that property (P) holds.

Proof. The proof proceeds by Löb induction. Assume P holds one step later – ($\triangleright$P). Using Lemma 6.33 our proof obligation simplifies to:

$$\mathbf{P}[\![\ [\Delta(\varnothing)]\mathbf{unit} \rightsquigarrow [\varnothing]\mathbf{int}\ ]\!]^{\lessapprox}(Id \circ (\uparrow^C e_+, w_1 \Box), Id \circ (\uparrow^C e_-, w_2 \Box))$$

By Lemma 6.37 it suffices to check the following conditions hold:

1. $\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lessapprox}(\uparrow e_+\ w_1\ (), \uparrow e_-\ w_2\ ())$;
2. $\forall c : A_1 \to B_1 \notin \Delta$, c-stuck$(Id \circ (\uparrow^C e_+, w_1 \Box))$ and c-stuck$(Id \circ (\uparrow^C e_-, w_2 \Box))$
3. $\forall c : A_c \to B_c \in \Delta, \triangleright \mathbf{V}[\![\ A_c\ ]\!]^{\lessapprox}(v_1, v_2)$ and H_1, H_2 such that c-normal$(H_1, H_2, [\Delta(\varnothing)]\mathbf{unit})$ then
$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lessapprox}(\uparrow e_+\ w_1\ (H_1 \star c\ v_1), \uparrow e_-\ w_2\ (H_2 \star c\ v_2));$$

Obligation (2) is straightforward. For (1), we use Lemma 6.30 and our assumption $\mathsf{AbsV}(w_1, w_2)$. Obligation (3) has two cases, one for each command in the `Tick` effect interface:

Case c = get.

We get the following sequence of simplifications of the goal:

$$\begin{array}{lll}
& \triangleright\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\uparrow e_+\ w_1\ (z_1\ w_1), \uparrow e_-\ w_2\ (z_2\ (-w_2))) & \text{By Lemma 6.30} \\
& \qquad \text{where } z_i = \uparrow(x, () \mapsto H_i@x) & \\
\Leftarrow & \triangleright^2\mathbf{T}[\![\ [\varnothing]\mathbf{unit}\ ]\!]^{\lesssim}(\uparrow e_+\ w_1\ (H_1@w_1), \uparrow e_-\ w_2\ (H_2@(-w_2))) & \text{By Lemma 6.30} \\
\Leftarrow & \triangleright\mathbf{T}[\![\ [\Delta(\varnothing)]\mathbf{unit}\ ]\!]^{\lesssim}(H_1@w_1, H_2@(-w_2)) & \text{By mono. and } (\triangleright\text{P})
\end{array}$$

By c-normal$(H_1, H_2, [\Delta(\varnothing)]\mathbf{unit})$ it remains to show:

$$\triangleright\mathbf{V}[\![\ \mathbf{int}\ ]\!]^{\lesssim}(w_1, -w_2)$$

which follows by definition and $\mathsf{AbsV}(w_1, w_2)$.

Case c = inc.

This case follows similar reasoning as above using Lemma 6.30 and ($\triangleright$P) leading to the obligation:

$$\begin{array}{lll}
\mathsf{AbsV}(w_1 + 1, w_2 - 1) & = & w_1 + 1 \equiv -(w_2 - 1) \\
& \iff & w_1 + 1 \equiv -w_2 + 1 \\
& \iff & w_1 + 1 \equiv w_1 + 1 \qquad \text{by assumption.}
\end{array}$$

□

7.2 Associativity of Pipe

In this section we apply our techniques to a canonical example of a multihandler called pipe. The pipe multihandler permits communication between two computations, one which expects to receive values and another which sends values. The handler facilitates the communication by passing the sent value as an argument to the receiver's continuation. We restate the signature for pipe given in Chapter 3:

```
pipe : {<Send X>Unit -> <Recv X>Y -> [Abort]Y}
```

where

```
interface Recv X = recv : X
interface Send X = send : X -> Unit
```

Recall the definition of pipe first introduced in Chapter 3:

```
{ <send x -> s> <recv -> r> ↦ f (s ()) (r x)
|      <_>             y      ↦ y
|      unit           <_>     ↦ abort () }
```

where we have replaced `pipe` with `f` to denote a recursive call.

The definition is subtle. Ordering of the arguments matters and requires that we take `Recv` to be primary; discarding any unmatched send effects.

We wish to prove that pipe is associative. That is, roughly, for appropriately typed terms a, b and c:

$$\text{pipe } a \text{ (pipe } b \text{ } c) \lesssim^{\text{VSC}} \text{pipe (pipe } a \text{ } b) \text{ } c \quad \text{(P)}$$

and vice versa.

We now develop the example in ELLA using the techniques from Chapter 6 to prove an approximation result corresponding to (P) – the other direction follows analogous reasoning.

Let e denote the previously given definition for pipe. Since ELLA does not support value polymorphism, we restrict ourselves to a version of pipe which transfers integer values between the two computations, eventually returning an integer on successful completion. Furthermore, since ELLA does not support effect polymorphism we work with a fully specified ambient ability. Given these constraints, the type signature for pipe becomes:

```
{C} = {<Send>Unit -> <Recv>Int -> [∅|Abort]Int}
```

where

```
interface Recv = recv : Unit -> Int
interface Send = send : Int -> Unit
```

Lastly, we require two additional type signatures for the innermost invocations of pipe because they occur in more permissive ambients than allowed by C.

```
{Cr} = {<Send>Unit -> <Recv>Int -> [∅|Abort,Recv]Int}
{Cw} = {<Send>Unit -> <Recv>Unit -> [∅|Abort,Send]Unit}
```

and the corresponding associativity property becomes:

$$\cdot \vdash \text{pipe } a \text{ (pipeR } b \text{ } c) \lesssim^{\text{VSC}} \text{pipe (pipeW } a \text{ } b) \text{ } c : [\text{Abort}(\varnothing)]\textbf{int} \quad \text{(G)}$$

where

$$\begin{aligned} \text{pipe} &= \uparrow^{C} e \\ \text{pipeR} &= \uparrow^{C_r} e \\ \text{pipeW} &= \uparrow^{C_w} e \end{aligned}$$

Intuitively, to prove (G) we need to analyse the argument computations a, b and c, reducing the applications to a particular clause in the corresponding definitions. To prove (G), we will leverage the triangle and simplify the terms using term decomposition and effect handling lemmas. Before proving (G), we prove some useful properties regarding `abort`-stuck terms and the above pipe definition.

Lemma 7.1 (Aborting Stacks)**.** *Given handler stacks H_1, H_2 and Σ s.t. abort* : **unit** $\to$ **zero** $\in \Sigma$*, abort*-stuck(H_1) *and abort*-stuck(H_2)*:*

$$\Vdash \mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(H_1 \star \mathit{abort}\ (), H_2 \star \mathit{abort}\ ())$$

Proof. By definition of $\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lesssim}$ we require to prove $\triangleright\mathbf{V}[\![\ \mathbf{unit}\ ]\!]^{\lessapprox}((),())$ (trivial) and for all $\cdot\,[\Sigma]\text{–}\ w \Rightarrow \mathbf{zero}$,

$$\triangleright\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lesssim}(H_1 @ \downarrow w, H_2 @ \downarrow w)$$

Since there are no inhabitants of **zero**, the result follows. □

Lemma 7.2. *If $k \in \mathbb{Z}$, $\Sigma = \varnothing$, Send, and*

$$\cdot\,[\Sigma]\text{–}\ u_a : \mathbf{int} \quad \textit{s.t.} \quad \Sigma\text{-normal}(u_a)$$

$$\cdot\,[\Sigma]\text{–}\ u_b : \mathbf{unit} \quad \textit{s.t.} \quad \Sigma\text{-normal}(u_b)$$

then

$$\mathbf{T}[\![\ [\text{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\text{pipe}\ u_a\ (\text{pipeR}\ u_b\ k), \text{pipe}\ (\text{pipeW}\ u_a\ u_b)\ k)$$

Proof. We have,

$$\begin{array}{rcl} \text{pipe}\ u_a\ (\text{pipeR}\ u_b\ k) & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) \circ (\uparrow^C \text{pipeR}, u_b \Box) @ k \\ & \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) @ k \\ & \longrightarrow & k \end{array}$$

For the right-hand side term, we consider the possible forms of u_b.

Case $u_b = ()$.

Calculating,

$$\begin{array}{rcl} \text{pipe}\ (\text{pipeW}\ u_a\ u_b)\ k & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, \Box k) \circ (\uparrow^C \text{pipeW}, u_a \Box) @ () \\ & \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, \Box k) @ () \\ & \longrightarrow & k \end{array}$$

Case $u_b = H_b \star \mathit{send}\ w_b$.

Calculating,

$$\begin{array}{lcl} \text{pipe}\ (\text{pipeW}\ u_a\ u_b)\ k & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, \Box k) \circ (\uparrow^C \text{pipeW}, u_a \Box) @ (H_b \star \mathit{send}\ w_b) \\ & \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, \Box k) @ ((\uparrow^C \text{pipeW}, u_a \Box) \bullet H_b) \star \mathit{send}\ w_b \\ & \longrightarrow & k \end{array}$$

In both cases, the result follows by Lemmas 6.48 and 6.47(1). □

Lemma 7.3. *If* $\Sigma = \varnothing, \mathit{Send}$, $\Sigma' = \varnothing, \mathit{Recv}$ *and*

$$\cdot\ [\Sigma] \vdash u_a : \mathbf{int} \quad \textit{s.t.} \quad \Sigma\text{-normal}(u_a)$$

$$\cdot\ [\Sigma'] \vdash u_c : \mathbf{unit} \quad \textit{s.t.} \quad \Sigma'\text{-normal}(u_c)$$

then

$$\Vdash \mathbf{T}[\![\ [\text{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}_{\approx}(\text{pipe}\ u_a\ (\text{pipeR}\ ()\ u_c), \text{pipe}\ ()\ u_c)$$

Proof. We proceed by case analysis on u_c.

Case $u_c = k$, for some $k \in \mathbb{Z}$.

We have,

$$\begin{array}{lcl} \text{pipe}\ u_a\ (\text{pipeR}\ ()\ u_c) & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) \circ (\uparrow^C \text{pipeR}, () \Box) @ k \\ & \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) @ k \\ & \longrightarrow & k \end{array}$$

and

$$\text{pipe}\ ()\ k \longrightarrow k$$

and the result follows by Lemmas 6.48 and 6.47(1).

Case $u_c = H_c \star \mathit{recv}\ ()$.

We have

$$\begin{array}{lcl} \text{pipe}\ u_a\ (\text{pipeR}\ ()\ u_c) & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) \circ (\uparrow^C \text{pipeR}, () \Box) @ (H_c \star \mathit{recv}\ ()) \\ & \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, u_a \Box) @ (\mathit{Id} \star \mathit{abort}\ ()) \\ & \longrightarrow & ((\uparrow^C \text{pipe}, u_a \Box) \bullet \mathit{Id}) \star \mathit{abort}\ () \end{array}$$

and on the right-hand side,

$$\begin{array}{lcl} \text{pipe}\ ()\ (H_c \star \mathit{recv}\ ()) & \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, () \Box) @ (H_c \star \mathit{recv}\ ()) \\ & \longrightarrow & \mathit{Id} \star \mathit{abort}\ () \end{array}$$

So, by Lemma 6.48 it suffices to prove for some $j > 0$,

$$\triangleright^j \mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(((\uparrow^C \mathrm{pipe}, u_a \Box) \bullet Id) \star abort\ (), Id \star abort\ ())$$

which follows by Lemmas 6.47(2) and 7.1.

□

We are now in a position to prove (G).

Proof. Since both sides can be decomposed into distinct evaluation contexts plugged with the same argument, we can prove (G) using just the applicative approximation. Moreover, using the applicative relation simplifies the proof because we need only consider the one normal form plugged into both sides, rather than related normal forms as for logical approximation. Using Lemma 6.43, it suffices to show:

$$\Vdash \forall a, b, c.\, \mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{pipe}\ a\ (\mathrm{pipeR}\ b\ c), \mathrm{pipe}\ (\mathrm{pipeW}\ a\ b)\ c) \quad (A)$$

By Löb induction, assume the goal holds one step later – (▷A). We now simplify our goal one argument at a time. Applying Lemma 6.50 followed by Lemma 6.51, we split our goal into two subgoals:

(i) $\mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{pipe}\ ()\ (\mathrm{pipeR}\ b\ c), \mathrm{pipe}\ (\mathrm{pipeW}\ ()\ b)\ c)$

(ii) $\mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{pipe}\ (H_a \star send\ v_a)\ (\mathrm{pipeR}\ b\ c), \mathrm{pipe}\ (\mathrm{pipeW}\ (H_a \star send\ v_a)\ b)\ c)$

In both subgoals we apply Lemmas 6.50 and 6.51 again to analyse the argument b.

Consider subcases arising from (i).

Case $b = ()$.

We have that

$$\mathrm{pipeW}\ ()\ () \longrightarrow ()$$

so using Lemma 6.48 we can simplify the right-hand side term:

$$\mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{pipe}\ ()\ (\mathrm{pipeR}\ ()\ c), \mathrm{pipe}\ ()\ c)$$

Now both terms are in the form $H@c$ for some H such that $abort$-stuck(H). So we can apply Lemmas 6.50 and 6.51 once more to argument c. Both cases follow by Lemma 7.3.

Case $b = H_b \star \mathit{send}\ v_b$.

It suffices to show:

$$\text{pipe}\ ()\ (\text{pipeR}\ (H_b \star \mathit{send}\ v_b)\ c) \lesssim^{trm} \text{pipe}\ (((\uparrow^C \text{pipeW}, ()\,\Box) \bullet H_b) \star \mathit{send}\ v_b)\ c$$

at $[\text{Abort}(\varnothing)]\mathbf{int}$.

Once more, Lemmas 6.50 and 6.51 apply since both terms form *abort*-stuck handler stacks paired with c.

Case $c = k$, for some $k \in \mathbb{Z}$.

By Lemma 7.2.

Case $c = H_c \star \mathit{recv}\ ()$.

We have,

$$\begin{array}{rl} & \text{pipe}\ ()\ (\text{pipeR}\ (H_b \star \mathit{send}\ v_b)\ (H_c \star \mathit{recv}\ ())) \\ \equiv & \mathit{Id} \circ (\uparrow^C \text{pipe}, ()\,\Box) \circ (\uparrow^C \text{pipeR}, (H_b \star \mathit{send}\ v_b)\,\Box) @ (H_c \star \mathit{recv}\ ()) \\ \longrightarrow & \mathit{Id} \circ (\uparrow^C \text{pipe}, ()\,\Box) @ (\text{pipeR}\ (\{x, () \mapsto H_b @ x)\}\ ())\ (\{x, () \mapsto H_c @ x\}\ v_b)) \\ \equiv & \text{pipe}\ ()\ (\text{pipeR}\ (\{x, () \mapsto H_b @ x)\}\ ())\ (\{x, () \mapsto H_c @ x\}\ v_b)) \\ \longrightarrow & \text{pipe}\ ()\ (\text{pipeR}\ (H_b @ ())\ (\{x, () \mapsto H_c @ x\}\ v_b)) \end{array}$$

and

$$\begin{array}{rl} & \text{pipe}\ (((\uparrow^C \text{pipeW}, ()\,\Box) \bullet H_b) \star \mathit{send}\ v_b)\ (H_c \star \mathit{recv}\ ()) \\ \longrightarrow & \text{pipe}\ (\{x, () \mapsto ((\uparrow^C \text{pipeW}, ()\,\Box) \bullet H_b) @ x\}\ ())\ (\{x, () \mapsto H_c @ x\}\ v_b) \\ \equiv & \text{pipe}\ (\{x, () \mapsto \text{pipeW}\ ()\ (H_b @ x)\}\ ())\ (\{x, () \mapsto H_c @ x\}\ v_b) \\ \longrightarrow & \text{pipe}\ (\text{pipeW}\ ()\ (H_b @ ()))\ (\{x, () \mapsto H_c @ x\}\ v_b) \end{array}$$

So the result follows by Lemma 6.48 and the Löb induction hypothesis.

Case $b = H_b \star \mathit{recv}\ ()$.

We require to show:

$$\text{pipe}\ ()\ (((\uparrow^C \text{pipeR}, \Box c) \bullet H_b) \star \mathit{recv}\ ()) \lesssim^{trm} \text{pipe}\ (\text{pipeW}\ ()\ H_b \star \mathit{recv}\ ())\ c$$

at $[\text{Abort}(\varnothing)]\mathbf{int}$.

We have,

$$\text{pipe}\ ()\ (((\uparrow^C \text{pipeR}, \Box c) \bullet H_b) \star \mathit{recv}\ ()) \longrightarrow \mathit{Id} \star \mathit{abort}\ ()$$

and

$$\begin{array}{lll} \text{pipe}\ (\text{pipeW}\ ()\ H_b \star \mathit{recv}\ ())\ c & \longrightarrow & \text{pipe}\ (\mathit{Id} \star \mathit{abort}\ ())\ c \\ & = & \mathit{Id} \circ (\uparrow^C \text{pipe}, \Box c) \star \mathit{abort}\ () \end{array}$$

The result follows using Lemmas 6.48, 6.47(2) and 7.1.

Now consider the subcases where $a = H_a \star \mathit{send}\ v_a$.

Case $b = ()$.

We have,

$$\text{pipe}\ (\text{pipeW}\ (H_1 \star \mathit{send}\ v_1)\ ())\ c) \quad \longrightarrow \quad \text{pipe}\ ()\ c$$

and using Lemma 6.48, our goal simplifies to:

$$\triangleright \mathbf{T}[\![\ [\text{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\text{pipe}\ (H_a \star \mathit{send}\ v_a)\ (\text{pipeR}\ ()\ c), \text{pipe}\ ()\ c)$$

Both terms can be decomposed into *abort*-stuck handler stacks acted on by the term c. Therefore, simplify the goal using Lemmas 6.50 and 6.51. The remaining subgoals follow from Lemma 7.3.

Case $b = H_b \star \mathit{send}\ v_b$.

This case follows the same reasoning as the previous case for $b = H_b \star \mathit{send}\ v_b$.

Case $b = H_b[\mathit{recv}\ ()]$.

We have,

$$\begin{array}{lll} & \text{pipe}\ (H_a \star \mathit{send}\ v_a)\ ((\uparrow^C \text{pipeR}, \Box c) \bullet H_b \star \mathit{recv}\ ()) & \\ \longrightarrow & \text{pipe}\ (\{x, () \mapsto H_a @ x\}\ ())\ (\{x, () \mapsto ((\uparrow^C \text{pipeR}, \Box c) \bullet H_b) @ x\}\ v_a) & \\ \longrightarrow & \text{pipe}\ (H_a @ ())\ (\{x, () \mapsto \text{pipeR}\ (H_b @ x)\ c\}\ v_a) & (\star) \end{array}$$

and

$$\begin{array}{lll} & \text{pipe}\ (\text{pipeW}\ (H_a \star \mathit{send}\ v_a)\ (H_b \star \mathit{recv}\ ()))\ c & \\ \longrightarrow & \text{pipe}\ (\text{pipeW}\ (\{x, () \mapsto H_a @ x\}\ ())\ (\{x, () \mapsto H_b @ x\}\ v_a))\ c & \\ \longrightarrow & \text{pipe}\ (\text{pipeW}\ (H_a @ ())\ (\{x, () \mapsto H_b @ x\}\ v_a))\ c & (\diamond) \end{array}$$

So by Lemma 6.48 it suffices to show ($\star$) and ($\diamond$) are related by $\mathbf{T}[\![\ -\ ]\!]^{\lesssim}$. By Lemma 6.50, we reduce our obligation to showing for some appropriate normal form u:

$$\triangleright \mathbf{T}[\![\ -\ ]\!]^{\lesssim}(\text{pipe}\ u\ (\{x, () \mapsto \text{pipeR}\ (H_b @ x)\ c\}\ v_a), \text{pipe}\ (\text{pipeW}\ u\ (\{x, () \mapsto H_b @ x\}\ v_a))\ c)$$

Both sides now reduce again using Lemma 6.48 and it remains to show:

$$\triangleright \mathbf{T}[\![\ [\mathrm{Abort}(\varnothing)]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{pipe}\ u\ (\mathrm{pipeR}\ (H_b@v_a)\ c), \mathrm{pipe}\ (\mathrm{pipeW}\ u\ (H_b@v_a)\ c)$$

which follows by the Löb induction hypothesis.

□

7.3 Validating Equations

We would like to be able to use our reasoning principles and approximation relations to prove certain effect handlers satisfy the equations associated with the effect being handled. For example, given the interpretation of state in Example 3.18, we would like to prove that the equations in Example 2.8 are satisfied. In particular, suppose we wish to show:

$$\mathbf{lkp}\ \ell\ (\lambda s.\mathbf{lkp}\ \ell\ (\lambda s'.M\ s\ s'))) = \mathbf{lkp}\ \ell\ (\lambda s.M\ s\ s) \quad (\star)$$

We can express this equation in ELLA using our approximation relations. First, assume we have $x : \mathbf{int}, y : \mathbf{int}\ [\mathrm{State}(\varnothing)]\text{–}\ n : A$. Define the following ELLA programs:

$$\begin{aligned} n_{xy} &\triangleq \mathbf{let}\ x : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ \mathbf{let}\ y : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ n \\ n_x &\triangleq \mathbf{let}\ x : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ n[x/y] \end{aligned}$$

Now, we define the `state` handler from Example 3.18, where `State` is specialised to an integer state cell:

```
state : {Int -> <State>Int  -> [∅|Int]}
state    _          x       = x
state    s     <get -> k>   = state s (k s)
state    _    <put s -> k> = state s (k unit)
```

By triangulation for ELLA, we are free to choose the applicative or logical notion to express an observational approximation. We choose the applicative approximation but our choice does not make the proof simpler in this case (cf. Section 7.2). The proof below implicitly appeals to monotonicity of the $\triangleright$ modality throughout. One direction of $(\star)$ can be expressed as follows.

$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{state}\ k\ n_{xy}, \mathrm{state}\ k\ n_x) \quad (A)$$

where $k \in \mathbb{Z}$.

Note that,

$$\begin{array}{rcl} n_{xy} & \equiv & (Id \circ (x : \mathbf{int}.n'_{xy})) @ (Id \star \mathrm{get}\ ()) \\ & \longrightarrow & Id \circ (x : \mathbf{int}.n'_{xy}) \star \mathrm{get}\ () \\ & & \text{where } n'_{xy} = \mathbf{let}\ y : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ n \end{array}$$

and

$$\begin{array}{rcl} n_{x} & \equiv & (Id \circ (x : \mathbf{int}.n[x/y])) @ (Id \star \mathrm{get}\ ()) \\ & \longrightarrow & Id \circ (x : \mathbf{int}.n[x/y]) \star \mathrm{get}\ () \end{array}$$

Both sides of (A) reduce by Lemma 6.48 to yield the following proof obligation:

$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{state}\ k\ (\{z, () \mapsto H_1 @ z\}\ k), \mathrm{state}\ k\ (\{z, () \mapsto H_2 @ z\}\ k))$$

where $H_1 \triangleq Id \circ (x : \mathbf{int}.n'_{xy})$ and $H_2 \triangleq Id \circ (x : \mathbf{int}.n[x/y])$. We can reduce the right-hand side using Lemma 6.48:

$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{state}\ k\ (\{z, () \mapsto H_1 @ z\}\ k), \mathrm{state}\ k\ (n[k/x][k/y]))$$

Now the right-hand side is in its final form, since we do not know anything about the term n. Once again, we can reduce the second argument of the left-hand side to a normal form with respect to the `state` handler using Lemma 6.48:

$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{state}\ k\ (\{z, () \mapsto (Id \circ (y : \mathbf{int}.n[k/x])) @ z\}\ k), \mathrm{state}\ k\ (n[k/x][k/y]))$$

Further application of Lemma 6.48 produces identical left- and right-hand side terms. The result follows by reflexivity of applicative approximation.

Similar reasoning can be performed to validate the other equations for state. Additionally, by changing the stateful computations slightly we can validate the following FRANK operator, first introduced in Section 3.6, to increment a counter:

```
next  : {[State Int]Int}
next! = fst get! (put (get! + 1))
```

Roughly, the following approximation would validate the `next` example.

$$\mathbf{T}[\![\ [\varnothing]\mathbf{int}\ ]\!]^{\lesssim}(\mathrm{state}\ k\ (\mathrm{fst}\ n_x\ n_y), \mathrm{state}\ k\ (\mathbf{let}\ x : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ \mathrm{fst}\ x\ (n'[x/y]))) \quad (\star\star)$$

where

$$\begin{array}{rcl} n_x & \triangleq & Id \star \mathrm{get}\ () \\ n_y & \triangleq & \mathbf{let}\ y : \mathbf{int} = \mathrm{get}\ ()\ \mathbf{in}\ n' \\ \multicolumn{3}{l}{\cdot\ [\Sigma_s]\text{–}\ \mathrm{fst} \Rightarrow \overline{\langle \iota \rangle \mathbf{int}} \to [\Sigma_S]B, \quad \Sigma_S = \mathrm{State}(\varnothing)} \end{array}$$

Proving $(\star\star)$ follows exactly the same structure as the previous proof. We provide a brief sketch here. We start by handling the first `get` command on both sides. We

can now proceed to reduce the second argument, n_y, on the left-hand side only; the right-hand side is already in the desired form. Reduction of n_y involves handling the second `get` command. Afterwards, both sides satisfy the form of Lemma 6.50, and we complete the proof by reflexivity of applicative approximation.

It is comforting to know our reasoning principles are sufficient to prove some of the equations arising from the algebraic theory of effects, as well as natural examples arising from operator application. The above result for **let** bindings has also been shown for EFF by Bauer and Pretnar [2014] using denotational equivalence induced by their semantics. But how useful is the above result in practice? There are not too many examples like our `next` operator which make consecutive calls to `get` with no intermediate code between the two invocations. Unless some program transformation were to introduce these redundancies itself, the result appears quite limited.

Based on our modification to validate the `next` example, we can arrive at a more general statement which allows arbitrary code to execute in-between the two `get` invocations *provided* the intermediate code does not perform any `State` effects. Indeed, our n_x term above was a special instance of `State`-free intermediate code; it returned immediately following the `get` command.

Our more general statement is as follows.

$$\text{state } k\ (f\ n_x\ n_y) \lesssim \text{state } k\ (\textbf{let}\ x : \textbf{int} = \text{get}\ ()\ \textbf{in}\ f\ n_1\ (n_2[x/y]))$$

where

$$\begin{aligned} n_x &\triangleq \textbf{let}\ x : \textbf{int} = \text{get}\ ()\ \textbf{in}\ n_1 \\ n_y &\triangleq \textbf{let}\ y : \textbf{int} = \text{get}\ ()\ \textbf{in}\ n_2 \end{aligned}$$

and assuming $\texttt{State} \notin (\Delta_i)_{i=1,2}$, $\Sigma_1 = \Delta_1(\varnothing)$ and $\Sigma_S = \text{State}(\varnothing)$:

$$x : \textbf{int}\,[\Sigma_1] \vdash n_1 : A_1$$

$$\cdot\,[\Sigma_S] \vdash f \Rightarrow \overline{\langle\Delta\rangle A} \to [\Sigma_S]B$$

The result for `next` would follow by instantiating f to `fst`, n_1 to `x`, and n_2 to `put (y + 1)`. However, there is a catch. The ambient environment in which n_1 occurs, namely the first argument to f, includes the `State` effect. The current ELLA formalism does not permit effect subsumption; in particular the T-APP rule forces the operator's ability to match exactly with the ambient environment:

$$\frac{\Gamma\,[\Sigma] \vdash v \Rightarrow \{\overline{\langle\Delta\rangle A}^{\,k} \to [\Sigma]B\} \qquad \cdots \qquad \cdots}{\Gamma\,[\Sigma] \vdash v\ \overline{n} \Rightarrow B}$$

We would need to extend our type system for ELLA to support the mask adaptor. Then, we modify the n_x term to mask the `State` effect during the execution of n_1:

$$n_x \triangleq \textbf{let}\, x : \textbf{int} = \text{get}\ ()\ \textbf{in}\ \langle \text{State} \rangle (n_1)$$

and the approximation to prove is now:

$$\mathbf{T}[\![\ [\varnothing]\textbf{int}\ ]\!]^{\lesssim}(\text{state}\ k\ (f\ n_x\ n_y), \text{state}\ k\ (\textbf{let}\, x : \textbf{int} = \text{get}\ ()\ \textbf{in}\ f\ (\langle \text{State} \rangle (n_1))\ (n_2[x/y])))$$

Assuming we have an appropriate interpretation for adaptors, the rest of the proof followings similar reasoning to our earlier examples in this section. The crucial step is in replacing the first argument to f on both sides with an identical normal form (Lemma 6.50) which relies on the fact n_1 does not perform any `State` effects; if n_1 reduces, it is either a value or a normal form with respect to f.

Of course, the devil is in the detail and our sketch does not formalism how adaptors are interpreted by our approximation relations; even if our usage here is motivated by subsumption concerns rather than 'skipping' handlers. In any case, Biernacki et al. [2018] and Zhang and Myers [2019] demonstrate sound logical relations for reasoning about program approximation in the presence of lift and multiple effect instances, and we conjecture extending ELLA's formalism to support mask and more general adaptors would be similar. The computational content of adaptors is described by Convent et al. [2020], which could be used as a starting point. Our frame stack formalisation could prove to be a more natural way to express their computational content, without requiring distinct runtime syntax for 'frozen' evaluation contexts.

A slightly weaker version of our more general statement, using **let**s instead of an operator f, can be proven for EFF using the techniques developed by Bauer and Pretnar [2014]. Using their notation, we may express the property as:

$$\mathcal{H}[\textbf{lkp}\ ()\ (\lambda x_1.\textbf{let}\, y_1 = c_1\ \textbf{in let}\, y_2 = c_2[x_1/x_2]\ \textbf{in}\ c[x_1/x_2]), e_s] \equiv$$

$$\mathcal{H}[\textbf{let}\, y_1 = \textbf{lkp}\ ()\ (\lambda x_1.c_1)\ \textbf{in let}\, y_2 = \textbf{lkp}\ ()\ (\lambda x_2.c_2)\ \textbf{in}\ c, e_s]$$

where c ranges over effectful computations, e ranges over pure value expressions, and operations are written in the continuation-passing style described in Section 2.1. The 'context' $\mathcal{H}$ represents the outer state handler which receives the argument computation and the initial state e_s. The definition of $\mathcal{H}$ is analogous to the ELLA definition provided in this section, only defined using a unary handler and parameter-passing; we omit the details and refer to Bauer and Pretnar [2014]. We may validate the above equivalence under the assumptions: (i) c_1 binds $x : \textbf{int}$ and does not perform any state

operations; (ii) c_2 binds $x_2 : \mathbf{int}$; and (iii) c binds $y_1 : A_1$ and $y_2 : A_2$. Thus, we can see that this equivalence validates the 'return' case of our operator f, that is the case:

$$f \triangleq \{y_1\ y_2 \mapsto n\}$$

where n corresponds to the c computation in ELLA notation.

Using effect subtyping, the authors prove commutativity of non-interfering effects (§ 7.2 of *loc. cit.*). A simpler equivalence which they prove along the way suffices to show the above equivalence:

$$\mathcal{H}[\mathbf{lkp}\ ()\ (\lambda x_2.\mathbf{let}\ y_1 = c_1\ \mathbf{in}\ \mathbf{let}\ y_2 = c_2\ \mathbf{in}\ c), e_s] \equiv$$

$$\mathcal{H}[\mathbf{let}\ y_1 = c_1\ \mathbf{in}\ \mathbf{let}\ y_2 = \mathbf{lkp}\ ()\ (\lambda x_2.c_2)\ \mathbf{in}\ c, e_s].$$

The proof follows by the above equivalence in conjunction with the β rules for **let**.

Rather than adding adaptors, we could extend ELLA with a limited form of effect subtyping to be able to validate this example:

T-APP
$$\frac{\Gamma\,[\Sigma]\vdash v \Rightarrow \{\overline{\langle\Delta\rangle A}^{k} \to [\Sigma']B\} \quad \Sigma' \sqsubseteq \Sigma \quad (\Delta_i(\Sigma') = \Sigma'_i)_{i=1\ldots k} \quad (\Gamma\,[\Sigma'_i]\vdash n_i : A_i)_{i=1\ldots k}}{\Gamma\,[\Sigma]\vdash v\ \overline{n} \Rightarrow B}$$

$$\frac{}{\varnothing \sqsubseteq \Sigma} \qquad \frac{\Sigma \sqsubseteq \Sigma'}{\Sigma, I \sqsubseteq \Sigma', I}$$

However, we would miss out on the other benefits of adaptors, including encapsulation of intermediate effects. Further work is needed to better understand the trade-offs between effect subtyping and adaptors/coercions in this setting.

Chapter 8

Conclusions

8.1 Revisiting The Thesis Statement

Recall from Chapter 1 our central thesis:

> *There are sound (in)equational reasoning principles applicable to monomorphic* FRANK *programs which are amenable to formalisation in an implementation of type theory such as* AGDA.

We have presented evidence in support of this thesis in the following ways.

We reconstructed the triangulation proof method for establishing context lemmas, formalising it in AGDA for a simply-typed fine-grained call-by-value λ-calculus, $\lambda^{\rightarrow}_{FG}$. The formalisation effort provides confidence as to the correctness of our results and serves as a basis for extensions to more exotic calculi. We reported on our fruitful use of a state-of-the-art framework for handling the intricacies of syntaxes with binding [Allais et al., 2017]. In particular, its facilities for defining meta-operations (renamings, substitutions, *etc.*) generically, and the straightforward representation of concrete contexts.

We presented ELLA, a monomorphic calculus capturing the essence of FRANK programming with operators, generalising multihandlers and *n*-ary functions. We extended our triangulation method to characterise observational approximation for ELLA. To do so, we introduced a modal logic, $\mathtt{FOL}^{\mu\triangleright}$, for abstracting step-index arithmetic in proofs where steps arise from approximating recursive language features such as general recursive functions and handlers. We proved sound reasoning principles for ELLA using our applicative and logical approximation relations, thus substantiating our first subsidiary thesis:

Triangulation extends to characterise observational equivalence for ELLA *and the constituent relations admit a collection of sound reasoning principles*

Regarding formalisation of these results for ELLA, we have made partial progress. We presented a model for the $\mathtt{FOL}^{\mu\triangleright}$ logic in ML_{UTT}, based on our AGDA formalisation. Using $\mathtt{FOL}^{\mu\triangleright}$, we have mechanised an extension of our $\lambda^{\rightarrow}_{FG}$ calculus to support general recursive functions and a proof of the fundamental property for its logical approximation; a component of the triangle. Additionally, we have formalised the ELLA calculus using the generic framework due to Allais et al. [2018], automatically deriving the appropriate meta-operations and their properties. However, we have yet to formalise any of the approximation relations defined in Chapter 6, and the associated reasoning properties. Nevertheless, we provided justification for believing such a formalisation is possible: our approach has taken inspiration from existing modal logics and reasoning principles which have previously been formalised for similar calculi.

Using the reasoning techniques we proved some example ELLA approximations, including validation of standard state equations for the state handler, equivalence of handlers, and an associativity result for a canonical multihandler, a binary pipe for communicating processes. These examples corroborate our second subsidiary thesis:

The reasoning principles for ELLA *are capable of proving concrete* ELLA *program approximations.*

While our collection of examples is limited, we also conjectured specific extensions to our formalism which should lead to more general theorems, *i.e.* adaptors for validating more general equations than those induced by the algebraic theory.

8.2 Future Work

We have previously hinted at potential avenues of future work throughout the dissertation. We briefly describe further work to be taken with respect to FRANK, $\mathtt{FOL}^{\mu\triangleright}$, and ELLA.

8.2.1 FRANK

We already discussed interesting research questions regarding our Hindler-Milner type inference example. In particular, casting other type inference algorithms within a uniform interface, and considering how Atkey's [2015] work relates to our specification

of 'type-inference-in-context'. Additionally, we suggested exploring incorporation of session types to allow specification of more elaborate communication protocols.

We could investigate alternative notions of pattern matching to see if it is useful for modelling particular problems. In particular, we could consider probabilistic or nondeterministic matching semantics, where we dispense with the usual precedence convention for matching clauses (cf. Figure 6.7 for ELLA). Given FRANK's more expressive coroutining between arguments and operator than standard call-by-value, these alternative semantics would be nontrivial, since we could select commands from any argument as and when they 'arrive'. However, it is not at all clear how we would reason about the behaviour of such handlers.

Other further directions could extend FRANK to richer type systems, incorporating a module system, or dependent types following Ahman [2017]. The former has been studied recently by Biernacki et al. [2018] who design a 'conventional' ML-style module system for HELIUM, but there is room to explore the design space a bit more and consider where best to situate handlers: are they part of a module *implementation* interpreting particular effects rendered abstract to clients (the approach taken by Biernacki et al.), or are they part of a module *interface* whereby a module specifies up-front the external effects upon which it relies? It is not clear these approaches are mutually exclusive. Extending FRANK with dependent types would be a first step — albeit with a long way still to go — towards internalising the reasoning principles outlined in this dissertation, *e.g.* pairing an implementation of `pipe` with a proof of its associativity.

8.2.2 $\texttt{FOL}^{\mu\triangleright}$

We presented the modal logic $\texttt{FOL}^{\mu\triangleright}$ as a means for encoding step-indexed logical relations and their properties. Using $\texttt{FOL}^{\mu\triangleright}$ helps to hide explicit step-indices in theorem statements and most proofs. An notable exception is the need to introduce step-index assumptions regarding monotonicity when dealing with implication. Polesiuk's [2017] IXFREE library defines tactics to automatically eliminate most of these introductions such that the user of the library never has to perform explicit step-index arithmetic. We could leverage AGDA's new reflection library [Agda, 2019a] to define similar tactic scripts. Comparing to LTAC tactic scripts, such AGDA-based scripts could utilise the full expressive power of the AGDA programming language and type system. Moreover, proof debugging is much easier since it can leverage the same interactive mode available when constructing regular programs. Kokke and Swierstra [2015] have shown

how to develop a first-order proof search tactic, `Auto`, using reflection. Such a tactic would simplify proving contractivity by automatically applying the simple rewriting rules for the logical connectives. As always, there is a trade-off on the amount of automation provided versus the subsequent readability of proofs.

However, tactic-based techniques may be naturally more suitable for COQ, a system more adept at theorem proving, than AGDA, which is closer to programming. In particular, introduction of assumptions does not work in the same way in the two systems. In COQ, introduction adds assumptions to a context which can be manipulated through tactics, allowing the user to remove unwanted hypotheses, or rename them. In contrast, AGDA's introduction mechanism is the dependent function space, and renaming or removing cannot be achieved dynamically using tactics. Hence, it seems as though recent enhancements for IRIS proof development in COQ [Krebbers et al., 2017], may not be applicable in our setting. We should therefore investigate programming-oriented techniques which may ease using an embedded logic.

Another direction is to generalise step-indexing in $\mathtt{FOL}^{\mu\triangleright}$ to Kripke worlds capable of containing other information, *e.g.* heap state. Doing so would follow in the footsteps of existing formalisations, ModuRes [Sieczkowski et al., 2015] and IRIS [Jung et al., 2018b], both in the COQ proof assistant. It would be interesting to investigate the challenges encountered in porting these developments to a system such as AGDA, and whether or not libraries such as Allais et al.'s [2018] generic syntaxes framework can be extended to support such efforts.

8.2.3 Ella

ELLA is a simply-typed fine-grained λ-calculus aiming to capture the essence of FRANK's distinguishing features, namely its approach to effect handlers. There is still a great deal of further work before we arrive at a formal account of equivalence for FRANK programs. As mentioned in Chapter 6, incorporating effect polymorphism and adaptors should not pose any difficulties, based on a similar development for $\lambda^{\mathsf{H/L}}$ [Biernacki et al., 2018]. In particular, for adaptors we could follow Leijen's [2018] approach by stratifying our frames according to whether they are regular evaluation frames, or 'handler' frames which may be skipped during effect handling under adaptors. In contrast, Biernacki et al. [2019b] use an explicit counter to determine the number of handlers to skip. We are encouraged by Leijen's approach because it accords with our intrinsic semantics point-of-view: establish the invariants we care about by construction. Con-

vent et al. [2020] use *adaptor functions* which encode the computational content of adaptors, but they do not give an abstract machine semantics.

Other extensions such as (parametrised) algebraic datatypes and parametrised interfaces would increase the usability of our formalism, and the range of examples we could tackle.

We aim to complete of our formalisation of triangulation for ELLA in AGDA along the lines of our 'pen-and-paper' formalism in this dissertation. Furthermore, we are interested in formalising the example approximations in Chapter 7. Certainly, the length of the pipe associativity proof is a potential source of difficulty; motivation for investigating proof automation and search strategies in $\texttt{FOL}^{\mu\triangleright}$ as mentioned above.

Looking ahead, we would like to able to reason about external resources which are currently not captured by ELLA. Currently, FRANK provides effect interfaces for standard input and output streams and mutable references [Convent et al., 2020]:

```
interface Console = inch : Char
                  | ouch : Char -> Unit

interface RefState = new X : X -> Ref X
                   | read X : Ref X -> X
                   | write X : Ref X -> X -> Unit
```

These effects are special built-in effects which are handled externally by the language runtime. In fact, it is currently not possible to define a handler for `RefState` in FRANK, instead these effects are handled by annotating the main entry point for a FRANK program:

```
main : {[Console,RefState]A}
main = ···
```

The techniques presented in this dissertation do not account for reasoning about such external resources, *e.g.* `read-read` is equivalent to `read`, or `inch` followed by `ouch` actually prints the input character. Furthermore, supporting additional external resources such as File I/O or network resources introduces the need to ensure safe resource deallocation. In particular, since handlers may invoke the continuation multiple times or even not at all, resource cleanup through regular return clauses is not appropriate. To support such reasoning, our formalism would need to have some way of specifying the 'state of the world' [McBride, 2011], articulating the assumptions our program makes about its execution environment.

Resources have been considered in existing languages based on algebraic effects, many being inspired by the coalgebraic foundations of state [Uustalu, 2015], albeit with slightly different design considerations. Bauer and Pretnar [2015] introduced resources in EFF [Bauer and Pretnar, 2014], essentially pairing some state with a default handler. User code modifies the state through the handled operations. Notably, resources are a 'top-level' feature, prohibiting operations to appear in their operation clauses. Thus, this particular approach to resources distinguishes them from ordinary handlers. Leijen [2018] extends KOKA with resources and finalisation. Finalisation entails extending handlers with *finally* branch to perform any resource cleanup, *e.g.* closing a file handle. Operation clauses are expected to call a finalisation procedure, if they do not invoke the continuation, which in turn executes all the finally branches in scope. However, the semantics do not guarantee finalisation is performed. Ahman and Bauer [2019] introduce *runners* of algebraic effects, distinct from effect handlers, in a core calculus. Runners execute computations while mediating access to a resource. The approach is similar to Bauer and Pretnar's by distinguishing runners from handlers. In particular, runners do not have access to the continuation so, by design, cannot discard or call the continuation multiple times, in contrast to Leijen's approach. Thus, they ensure linear usage of a resource within a runner's body, whilst their denotational semantics guarantees, under normal execution [1], resources are safely reclaimed. However, they do not consider a facility akin to Leijen's [2018] intialisers, an additional clause of a resource handler which encapsulates the initialisation code. Arguably, initialisation is just as important as finalisation since without proper encapsulation, a resource may escape its finalisation code and become a dangling pointer to deallocated memory. Consider (a slightly modified version of) their File I/O example [Ahman and Bauer, 2019]:

```
let f = open "hello.txt"
in using fileIO @ f run
     write "Hello, world."
   finally {
     return x @ fh -> close fh,
     raise QuotaExceeded @ fh -> close fh,
     kill IOError -> return ()
   }; ··· f still in scope ···
```

[1] Normal execution assumes no `kill` signals are sent by enclosing runners. Signals indicate an unrecoverable failure mode.

where, as the authors themselves observe, their runner does not account for the opening of a file resource; we have made this more explicit here by `let`-binding the handle returned by open.

Runners seem like a natural way to encode concurrent actors with mailboxes, without resorting to mutable references, as presented by Convent et al. [2020] as their showpiece example leveraging adaptors. In the FRANK setting, a natural question to consider is whether *multirunners* offers any advantages over the unary sort. Perhaps more pressing is how to understand resource management in the presence of multiple interacting continuations each executing with respect to a collection of resources, some of which may be shared. Given the FRANK methodology of generalising function application to more exotic forms of command-response interaction between operator and argument, it is prudent to investigate richer structures and type systems which allow for the safe incorporation of resource handlers into the more expressive construct.

Appendix A

Effect Pollution à la Biernacki et al.

We illustrate (in FRANK) the original effect pollution problem presented by Biernacki et al. [2018] in $\lambda^{H/L}$.

```
interface Tick = tick : Unit

execTicks : Int -> <Tick>X -> Int
execTicks    n <tick -> k> = execTicks (n + 1) (k unit)
execTicks    n      x      = n

runTicks : Int -> <Tick>X -> Pair X Int
runTicks    n     <tick -> k> = runTicks (n + 1) (k unit)
runTicks    n         x       = pair x n

f : {{Bool -> Bool} -> Bool}
f g = g (g tt)

fCnt : {Bool -> [Tick]Bool} -> Int
fCnt g = handleTicks 0 (f {x -> tick!; g x}) -- leaky!
```

The code above defines an effect-polymorphic function `fCnt` which is intended to count the number of times the effect-polymorphic function `f` calls its argument function. However, as observed by Biernacki et al., under certain conditions `fCnt` exhibits unexpected behaviour. In particular, if g invokes the `tick` command, it will be handled by the `handleTicks` handler inside `fCnt` instead of being forwarded to the surrounding context.

```
runTicks 0 (fCnt {x -> tick!; x} ⟹ pair 4 0
```

Furthermore, the typing for `fCnt` leaks the implementation details of the counter by offering the `Tick` effect to its function argument; this offer is forced by the type system since the call to g occurs in a more permissive ambient.

The solution proposed by Biernacki et al. [2018] is to coerce the type of the expression `g x` to the type expected by its surrounding context and thereby allow a more polymorphic type for g. Biernacki et al. define an operation, *lift*, which extends the ability of an expression with an additional effect label. The equivalent formulation in FRANK is *mask*, which restricts the ambient ability by removing an interface instance. These two notions are equivalent in their outcome, the difference arising due to FRANK being based on a bidirectional type system where effects are pushed inwards, whereas $\lambda^{H/L}$ is based on Hindley-Milner type inference where effects flow outwards. Thus, to achieve the type `{Bool -> [ε]Bool}` for g, we remove (i.e. mask) the `Tick` effect from the ambient ability when calling g.

```
fCntAdp : {Bool -> Bool} -> Int
fCntAdp g = handleTicks 0 (f {x -> tick!; <Tick> (g x)})
```

FRANK supports masking through general ability rewiring afforded by *adaptors* (Section 3.12). In the example above, we wrap the call to g with an adaptor to achieve the desired typing and behaviour:

```
runTicks 0 (fCntAdp {x -> tick!; x} ⟹ pair 2 2
```

Appendix B

Contractivity of ELLA Logical Approximation

This appendix presents a proof of contractivity for ELLA logical approximation, hence by Theorem 5.10 the relation has a (unique) fixed point given by the fixed-point predicate in $\mathtt{FOL}^{\mu\triangleright}$. Recall the definition of ELLA logical approximation reproduced here in Figure B.1. These component relations are implicitly defined using the fixed-point predicate construct of $\mathtt{FOL}^{\mu\triangleright}$. To arrive at these definitions, we must consider the contractive functions which give rise to them. That is, we unfold the recursive calls by providing an explicit argument representing the fixed point of the relation being defined. Additionally, we inline the value case of $\mathbf{U}[\![-]\!]$ so that appeals to $\mathbf{U}[\![-]\!]$ only consider stuck terms. Folding the value case into $\mathbf{U}[\![-]\!]$ was for conciseness only, and does not represent a recursive occurrence using the predicate variable; justifying the absence of the $\triangleright$ modality in that case.

In the following we identify the codes within the universe $\mathbb{S}$ with their underlying interpretation in $\mathtt{Set}$ using $[\![-]\!]$. For ELLA the universe of codes consists of the codes corresponding to the phrase classes in Figure 6.1, and the standard constructions for dependent product and sum types. For simplicity, we assume we are dealing with closed terms; extending to open terms is straightforward but requires additional indices on codes to track the scope of terms.

Define the following relations on well-typed values and terms.

$$\begin{array}{lcl} \mathsf{VRel} & : & \mathtt{Set}_1 \\ \mathsf{VRel} & \triangleq & \mathsf{Pred}\ (\Pi A : \mathtt{Ty}\,.\,\mathtt{Val}\ A \times \mathtt{Val}\ A) \\ \mathsf{TRel} & : & \mathtt{Set}_1 \\ \mathsf{TRel} & \triangleq & \mathsf{Pred}\ (\Pi A : \mathtt{Ty}, \Sigma : \mathtt{Ab}.\mathtt{Trm}^{\downarrow}\ A\ \Sigma \times \mathtt{Trm}^{\downarrow}\ A\ \Sigma) \end{array}$$

Component Relations

$$\mathbf{V}[\![\ \tau\ ]\!]^{\lessapprox}(v_1,v_2) \triangleq v_1 \equiv v_2$$

$$\mathbf{V}[\![\ \{C\}\ ]\!]^{\lessapprox}(\uparrow e_1,\uparrow e_2) \triangleq \forall \overline{u_1},\overline{u_2},n_1.\ \textstyle\bigwedge_i(\mathbf{U}[\![\ [\Delta_i(\Sigma)]A_i\ ]\!]^{\lessapprox}(u_{1,i},u_{2,i})) \wedge e_1' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_1} \dashv [\Sigma]\ n_1 \implies \exists n_2.\ e_2' : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_2} \dashv [\Sigma]\ n_2 \wedge \triangleright\mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}(n_1,n_2)$$
where $C = \overline{\langle\Delta\rangle A} \to [\Sigma]B, \quad e_i' = \mathbf{unfold}(e_i),\ i = 1,2$

$$\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(n_1,n_2) \triangleq \forall H_1,H_2.\mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1,H_2) \implies \mathbf{O}(\langle H_1,n_1\rangle,\langle H_2,n_2\rangle)$$

$$\mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1,H_2) \triangleq \forall u_1,u_2.\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(u_1,u_2) \implies \mathbf{O}(\langle H_1,u_1\rangle,\langle H_2,u_2\rangle)$$

$$\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(\downarrow v_1,\downarrow v_2) \triangleq \mathbf{V}[\![\ A\ ]\!]^{\lessapprox}(v_1,v_2)$$

$$\mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(u_1,u_2) \triangleq c : A_c \to B_c \in \Sigma \wedge (c\text{-stuck}(H_i))_{i=1,2} \wedge \triangleright\mathbf{V}[\![\ A_c\ ]\!]^{\lessapprox}(v_1,v_2) \wedge \forall w_1,w_2.\triangleright\mathbf{V}[\![\ B_c\ ]\!]^{\lessapprox}(w_1,w_2) \implies \triangleright\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(H_1@{\downarrow}w_1,H_2@{\downarrow}w_2)$$
where $u_i = H_i \star c\ v_i,\ i = 1,2$

$$\mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1,\theta_2) \triangleq \forall (x:A) \in \Gamma.\ \mathbf{V}[\![\ A\ ]\!]^{\lessapprox}(\theta_1(x),\theta_2(x))$$

Logical Relation

$$\Gamma \vdash n_1 \lessapprox^{trm} n_2 : [\Sigma]A \triangleq \theta_1,\theta_2 : \Gamma \vDash \cdot \Vdash \mathbf{G}[\![\ \Gamma\ ]\!]^{\lessapprox}(\theta_1,\theta_2) \implies \mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}(\theta_1(n_1),\theta_2(n_2))$$
and similar for $\lessapprox^{val},\lessapprox^{nf},\lessapprox^{stk}$

$$\Gamma \vdash n_1 \approx n_2 : [\Sigma]A \triangleq \Gamma \vdash n_1 \lessapprox n_2 : [\Sigma]A \times \Gamma \vdash n_2 \lessapprox n_1 : [\Sigma]A$$

Figure B.1: ELLA Logical Approximation

We begin by defining the predicate for $\mathbf{U}[\![-]\!]^{\lessapprox}$, U_π, which has the following signature.

$$\mathsf{U}_\pi : \mathsf{VRel} \to \mathsf{TRel} \to \mathsf{Pred}(\Pi A : \mathtt{Ty}, \Sigma : \mathtt{Ab.NF}\ A\ \Sigma \times \mathtt{NF}\ A\ \Sigma)$$

Note that the signature for U_π specifies returning a relation over NF terms which captures both stuck terms and values. This is an abuse of notation: the actual relation will carve out only the stuck terms such that U_π is not defined for value normal forms. Indeed, U_π is defined by:

$$\begin{aligned}\mathsf{U}_\pi\ \rho_{\mathrm{V}}\ \rho_{\mathrm{T}}\ A\ \Sigma\ (H_1 \star c\ v_1, H_2 \star c\ v_2) \triangleq\ & c : A_c \to B_c \in \Sigma \wedge (c\text{-stuck}(H_i))_{i=1,2} \wedge \triangleright \rho_{\mathrm{V}}\ A_c\ (v_1, v_2) \wedge \\ & \forall w_1, w_2. \triangleright \rho_{\mathrm{V}}\ B_c\ (w_1, w_2) \implies \triangleright \rho_{\mathrm{T}}\ A\ \Sigma\ (H_1 @ \downarrow w_1, H_2 @ \downarrow w_2)\end{aligned}$$

Note that the use of the predicate value and term variables appear under the later modality. Whilst Definition 5.8 and Example 5.9 state contractivity in terms of functions of the form $\mathsf{Pred}\ \mathcal{S} \to \mathsf{Pred}\ \mathcal{S}$, we can derive analogous results even when the domain of the predicate variable differs from the domain of the relation being defined, as is the case in the next lemma.

Lemma B.1. *The* U_π *function is contractive with respect to* ρ_{V} *and* ρ_{T}*. That is, for all* $\pi_v, \pi_v' : \mathsf{VRel}$*, and* $\pi_t, \pi_t' : \mathsf{TRel}$*,*

$$\triangleright(\pi_v \sim \pi_v'), \triangleright(\pi_t \sim \pi_t') \Vdash \mathsf{U}_\pi\ \pi_v\ \pi_t \sim \mathsf{U}_\pi\ \pi_v'\ \pi_t'.$$

The signature for the predicate defining stack approximation is:

$$\mathsf{S}_\pi : \mathsf{VRel} \to \mathsf{TRel} \to \mathsf{Pred}(\Pi A : \mathtt{Ty}, \Sigma : \mathtt{Ab.Stk}\ A\ \Sigma\ \mathbf{unit}\ \varnothing \times \mathtt{Stk}\ A\ \Sigma\ \mathbf{unit}\ \varnothing)$$

and is defined by:

$$\begin{aligned}\mathsf{S}_\pi\ \rho_{\mathrm{V}}\ \rho_{\mathrm{T}}\ A\ \Sigma\ (H_1, H_2) \triangleq\ & (\forall v_1, v_2. \rho_{\mathrm{V}}\ A\ \Sigma\ (v_1, v_2) \implies \mathbf{O}(\langle H_1, v_1\rangle, \langle H_2, v_2\rangle)) \wedge \\ & (\forall u_1, u_2. U_\pi\ \rho_{\mathrm{V}}\ \rho_{\mathrm{T}}\ A\ \Sigma\ (u_1, u_2) \implies \mathbf{O}(\langle H_1, u_1\rangle, \langle H_2, u_2\rangle))\end{aligned}$$

where $\mathbf{O}$ is the basic observation relation defined in Section 6.3.1. Given this definition, we can see that S_π is contractive in ρ_{T}, but it is *non-expansive* in ρ_{V}.

Definition B.2. Given an OFE $\langle A, <, X, \sim\rangle$ a function $f : X \to X$ is *non-expansive* if for every $x, y \in X$ and $a \in A$ we have

$$x \sim_a y \implies f(x) \sim_a f(y).$$

In other words, for a predicate to be non-expansive, the relations substituted for the predicate variable ρ_V must be approximate at the current index, a stronger requirement than contractivity. The value case for S_π aligns with our definition of $\mathbf{S}[\![-]\!]$ in Figure B.1 where we draw values from $\mathbf{V}[\![-]\!]$ without decorating it with the later modality.

Lemma B.3. *For all* $\pi_v, \pi'_v : \mathsf{VRel}$*, and* $\pi_t, \pi'_t : \mathsf{TRel}$*,*

$$\pi_v \sim \pi'_v, \triangleright(\pi_t \sim \pi'_t) \Vdash S_\pi\ \pi_v\ \pi_t \sim S_\pi\ \pi'_v\ \pi'_t.$$

The signature and definition for logical term approximation follows the same pattern as for unfolding the stack approximation.

$$\begin{array}{rcl} T_\pi & : & \mathsf{VRel} \to \mathsf{TRel} \to \mathsf{TRel} \\ T_\pi\ \rho_V\ \rho_T\ A\ \Sigma\ (n_1, n_2) & \triangleq & \forall H_1, H_2.\, S_\pi\ \rho_V\ \rho_T\ A\ \Sigma\ (H_1, H_2) \implies \mathbf{O}(\langle H_1, n_1\rangle, \langle H_2, n_2\rangle) \end{array}$$

Lemma B.4. *Term approximation,* T_π*, is contractive (non-expansive) in* ρ_T *(*ρ_V*):*

$$\pi_v \sim \pi'_v, \triangleright(\pi_t \sim \pi'_t) \Vdash T_\pi\ \pi_v\ \pi_t \sim T_\pi\ \pi'_v\ \pi'_t.$$

We are now in a position to define our first fixed-point predicates for stacks, terms and normal forms. By Lemma B.4 and Theorem 5.10, it follows that T_π has a fixed point given by Definition 5.16. Hence,

$$\mathbf{T}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}_\pi\ \rho_V \triangleq (\mu\rho_T.T_\pi\ \rho_V)\ A\ \Sigma$$

We have yet to define the value relation so this definition still accepts an additional argument for that relation. The subscript π is used to indicate this is another intermediate definition. The relations U_π and S_π can now be specialised to use the above term approximation.

$$\begin{array}{rcl} \mathbf{S}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}_\pi\ \rho_V & \triangleq & S_\pi\ \rho_V\ (\mathbf{T}[\![-]\!]^{\lessapprox}_\pi\ \rho_V)\ A\ \Sigma \\ \mathbf{U}[\![\ [\Sigma]A\ ]\!]^{\lessapprox}_\pi\ \rho_V & \triangleq & U_\pi\ \rho_V\ (\mathbf{T}[\![-]\!]^{\lessapprox}_\pi\ \rho_V)\ A\ \Sigma \end{array}$$

Now we define the value approximation relation. As in Figure B.1, we define the $\mathtt{FOL}^{\mu\triangleright}$ recursive predicate by primitive recursion on the type A of values being related.

$$\begin{array}{rcl} V_\pi & : & \mathsf{VRel} \to \mathsf{VRel} \\ V_\pi\ \rho_V\ \tau\ (v_1, v_2) & \triangleq & v_1 \equiv v_2 \\ V_\pi\ \rho_V\ \{C\}\ (\uparrow e_1, \uparrow e_2) & \triangleq & (\forall \overline{v_1}, \overline{v_2}, n_1.\, \bigwedge_i (V_\pi\ \rho_V\ A_i\ (v_{1,i}, v_{2,i})) \wedge e'_1 : \overline{\langle\Delta\rangle A} \longleftarrow \overline{v_1} \dashv [\Sigma]\ n_1 \implies \\ & & \quad \exists n_2.\, e'_2 : \overline{\langle\Delta\rangle A} \longleftarrow \overline{v_2} \dashv [\Sigma]\ n_2 \wedge \triangleright \mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}_\pi\ \rho_V\ (n_1, n_2)) \\ & & \wedge \\ & & (\forall \overline{u_1}, \overline{u_2}, n_1.\, \bigwedge_i (\mathbf{U}[\![\ [\Delta_i(\Sigma)]A_i\ ]\!]^{\lessapprox}_\pi\ \rho_V\ (u_{1,i}, u_{2,i})) \wedge e'_1 : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_1} \dashv [\Sigma]\ n_1 \implies \\ & & \quad \exists n_2.\, e'_2 : \overline{\langle\Delta\rangle A} \longleftarrow \overline{u_2} \dashv [\Sigma]\ n_2 \wedge \triangleright \mathbf{T}[\![\ [\Sigma]B\ ]\!]^{\lessapprox}_\pi\ \rho_V\ (n_1, n_2)) \\ & & \text{where } C = \overline{\langle\Delta\rangle A} \to [\Sigma]B, \quad e'_i = \mathbf{unfold}(e_i),\ i = 1, 2 \end{array}$$

Observe that — for the thunk case — we have once again inlined the value case for normal forms, taking advantage of the fact that we are defining V_π by primitive recursion, we can inspect related arguments directly, without requiring to place it under the later modality. Intuitively, we wrap calls to $\mathbf{T}[\![-]\!]^{\lessapprox}_{\pi}$ in a later because we have β-reduced our operators. The technical reason for doing so is to make V_π contractive with respect to ρ_{V}, since T_π (hence $\mathbf{T}[\![-]\!]^{\lessapprox}_{\pi}$) is only non-expansive.

Lemma B.5. $\mathbf{T}[\![-]\!]^{\lessapprox}_{\pi}$ *is non-expansive in* ρ_{V}*:*

$$\pi_v \sim \pi'_v \Vdash \mathbf{T}[\![-]\!]^{\lessapprox}_{\pi}\ \pi_v \sim \mathbf{T}[\![-]\!]^{\lessapprox}_{\pi}\ \pi'_v.$$

Proof. By Löb induction using UNROLL (Section 5.2) and Lemma B.4. □

Lemma B.6. V_π *is contractive with respect to* ρ_{V}*:*

$$\triangleright(\pi_v \sim \pi'_v) \Vdash \mathsf{V}_\pi\ \pi_v \sim \mathsf{V}_\pi\ \pi'_v.$$

Proof. By induction on the type of values being related, using Lemmas B.5, B.3 and B.1. □

We can now tie the mutual recursive definitions together by defining the fixed-point predicate for value approximation:

$$\mathbf{V}[\![A]\!]^{\lessapprox} \triangleq (\mu\rho_{\mathrm{V}}.\mathsf{V}_\pi)\ A$$

The final versions of our other approximation relations are now definable:

$$\begin{array}{rcl}
\mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox} & \triangleq & \mathbf{T}[\![[\Sigma]A]\!]^{\lessapprox}_{\pi}\ (\mathbf{V}[\![-]\!]^{\lessapprox}) \\
\mathbf{S}[\![[\Sigma]A]\!]^{\lessapprox} & \triangleq & \mathbf{S}[\![[\Sigma]A]\!]^{\lessapprox}_{\pi}\ (\mathbf{V}[\![-]\!]^{\lessapprox}) \\
\mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox} & \triangleq & \mathbf{U}[\![[\Sigma]A]\!]^{\lessapprox}_{\pi}\ (\mathbf{V}[\![-]\!]^{\lessapprox})
\end{array}$$

This completes the formal definition of our ELLA logical approximation relations within the $\mathtt{FOL}^{\mu\triangleright}$ logic.

Bibliography

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMARK Reloaded: Mechanizing Proofs by Logical Relations. *Journal of Functional Programming*, 29:e19, 2019. doi:10.1017/S0956796819000170.
(Cited on page 92.)

Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive parsing for Parsec. In *Haskell*. ACM, 2014.
(Cited on page 64.)

Development Team Agda. Agda v2.6 reflection mechanisms, 2019a. Accessed Nov. 2019. `https://agda.readthedocs.io/en/v2.6.0.1/language/reflection.html`.
(Cited on page 173.)

Development Team Agda. Agda categories library, 2019b. Accessed Nov. 2019. `https://github.com/agda/agda-categories`.
(Cited on page 13.)

Danel Ahman. *Fibred Computational Effects*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 2017.
(Cited on pp. 2 and 173.)

Danel Ahman and Andrej Bauer. Runners in action, 2019. Drafted. Submitted for publication (ESOP 2020). arXiv:1910.11629.
(Cited on page 176.)

Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
(Cited on page 96.)

Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pp. 69–83. Springer, 2006. doi:10.1007/11693024_6.
(Cited on pp. 8, 96, and 152.)

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pp. 195–207, 2017.
(Cited on pp. 7, 25, 26, 69, 71, and 171.)

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, 2018.
(Cited on pp. 25, 92, 150, 151, 172, and 174.)

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of CSL'99*, volume 1683 of *LNCS*, pp. 453–468. Springer, 1999.
(Cited on page 23.)

Simon Ambler and Roy L. Crole. Mechanized operational semantics via (co)induction. In *TPHOLs'99, Nice, France*, pp. 221–238, 1999. doi:10.1007/3-540-48256-3_15.
(Cited on pp. 69, 92, and 93.)

Simon J. Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pp. 13–30. Springer, 2002.
(Cited on page 93.)

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pp. 109–122, 2007. doi:10.1145/1190216.1190235.
(Cited on pp. 96, 97, 111, and 112.)

Robert Atkey. An algebraic approach to typechecking and elaboration. `https://bentnib.org/docs/algebraic-typechecking-20150218.pdf`, 2015. Talk presented at HOPE 2015, Vancouver, BC, Canada.
(Cited on pp. 65 and 172.)

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08*, pp. 3–15, 2008.
(Cited on page 92.)

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pp. 50–65, 2005.
(Cited on pp. 25, 70, and 92.)

Henning Basold. *Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic*. PhD thesis, Radboud University, 2018.
(Cited on page 113.)

Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4:9), 2014.
(Cited on pp. 3, 14, 64, 153, 168, 169, and 176.)

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.
(Cited on pp. 3, 45, 48, 64, 156, and 176.)

Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, 2001.
(Cited on page 19.)

Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, pp. 42–122. Springer, 2002.
(Cited on page 12.)

Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

(Cited on page 25.)

Dariusz Biernacki and Piotr Polesiuk. Logical Relations for Coherence of Effect Subtyping. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 107–122, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TLCA.2015.107.
(Cited on page 150.)

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL):8:1–8:30, December 2018.
(Cited on pp. xi, 50, 65, 66, 101, 113, 116, 138, 150, 153, 169, 173, 174, 179, and 180.)

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Bisimulations for delimited-control operators. *Logical Methods in Computer Science*, Volume 15(Article 18): 18:1–18:57, 2019a.
(Cited on page 156.)

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL):6:1–6:28, January 2019b.
(Cited on pp. 45, 65, 66, 152, and 174.)

Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Information and Computation*, 155(1):3 – 63, 1999. doi:https://doi.org/10.1006/inco.1999.2828.
(Cited on pp. 96 and 153.)

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013. doi:10.1017/S095679681300018X.
(Cited on pp. 22 and 64.)

Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
(Cited on page 1.)

Lukas Convent. Enhancing a modular effectful programming language. Master's thesis, School of Informatics, University of Edinburgh, 2017.
(Cited on pp. 42, 57, 58, and 63.)

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Frank repository, 2019. `https://www.github.com/frank-lang/frank`.
(Cited on page 63.)

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming (special issue on algebraic effects and handlers)*, 30:e9, 2020. doi:10.1017/S0956796820000039.
(Cited on pp. 58, 63, 65, 116, 169, 174, 175, and 177.)

Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Types for Proofs and Programs Workshop*, 1992.
(Cited on page 5.)

Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76 (2-3):95–120, 1988.
(Cited on page 5.)

Ugo Dal Lago and Francesco Gavazzo. Effectful Normal Form Bisimulation. In Luís Caires, editor, *Programming Languages and Systems*, ESOP 2019, pp. 263–292. Springer, 2019.
(Cited on page 155.)

Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '17, pp. 57:1–57:12, 2017. doi:10.1109/LICS.2017.8005117.
(Cited on page 155.)

N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 75(5):381–392, 1972.
(Cited on pp. 23 and 56.)

David Delahaye. A Tactic Language for the System Coq. In *LPAR'00*, pp. 85–95, 2000.

(Cited on page 111.)

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Proceedings of TLCA*, volume 902 of *LNCS*, pp. 124–138. Springer, 1995.
(Cited on pp. 23 and 71.)

Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pp. 148–161. Springer, 2003.
(Cited on pp. 8, 97, 101, 102, 103, 104, 105, and 112.)

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, 2015.
(Cited on page 48.)

Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, Volume 7(Article 16), 2011.
(Cited on pp. 8, 97, 101, 110, 112, 113, 116, and 136.)

Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.
(Cited on page 21.)

Peter Dybjer. Inductive families. *Formal Aspects of Comp.*, 6(4):440–465, 1994.
(Cited on pp. 5, 23, and 71.)

Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pp. 129–146. Springer, 1999.
(Cited on page 99.)

Matthias Felleisen and P. Daniel Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pp. 193–217. Elsevier Science Publishers Ltd., 1987.
(Cited on page 152.)

Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *POPL*, pp. 372–385, 1996.
(Cited on page 65.)

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019.
(Cited on page 67.)

Carsten Führmann. Varieties of effects. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pp. 144–159. Springer, 2002.
(Cited on page 154.)

Andy Gill. mtl: Monad classes, using functional dependencies. `http://hackage.haskell.org/package/mtl-2.2.2`, 2018. Date last accessed: August 2019.
(Cited on page 48.)

Healfdene Goguen. The metatheory of UTT. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pp. 60–82. Springer, 1995.
(Cited on page 26.)

Andrew D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1st edition, 1994. ISBN 0521471036.
(Cited on pp. 20, 79, 91, and 92.)

Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer, 1979.
(Cited on page 5.)

Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pp. 275–291. Springer, 2014. doi:10.1007/978-3-319-08970-6_18.
(Cited on page 13.)

Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In *MSFP*. ACM, 2010.
(Cited on pp. 30, 54, 58, and 64.)

Adam Michael Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.
(Cited on pp. 30, 54, 55, 62, and 64.)

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. ISSN 0004-5411.
(Cited on pp. 22 and 93.)

Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe'16)*, pp. 15–27, 2016. doi:10.1145/2976022.2976033.
(Cited on pp. 3, 14, 45, 64, 65, and 152.)

Daniel Hillerström and Sam Lindley. Shallow effect handlers. *16th Asian Symposium on Programming Languages and Systems*, 2018.
(Cited on pp. 20 and 45.)

Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pp. 509–523. Springer, 1993.
(Cited on page 67.)

Douglas J. Howe. Equality in lazy computation systems. In *Fourth Annual Symposium on Logic in Computer Science*, pp. 198–203, 1989.
(Cited on pp. 4, 20, 82, and 91.)

Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
(Cited on pp. 4, 20, and 91.)

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pp. 59–72, 2012.
(Cited on page 156.)

Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70 –99, 2006.
(Cited on page 18.)

Rosalie Iemhoff. Intuitionism in the philosophy of mathematics. *The Stanford Encyclopedia of Philosophy*, 2019.
`https://plato.stanford.edu/archives/sum2019/entries/intuitionism/`.
(Cited on page 5.)

Mauro Jaskelioff. Modular monad transformers. In Giuseppe Castagna, editor, *18th European Symposium on Programming*, ESOP'09, pp. 64–79, 2009.
(Cited on page 18.)

Mauro Jaskelioff. Monatron: An extensible monad transformer library. In Sven-Bodo Scholz and Olaf Chitil, editors, *20th International Symposium on Implementation and Application of Functional Languages*, IFL'08, pp. 233–248, 2011.
(Cited on page 18.)

Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *LICS '10*, pp. 209–218. IEEE Computer Society, 2010.
(Cited on pp. 76, 91, 152, and 155.)

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2018a.
(Cited on page 98.)

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ale Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 2018b. Accepted for publication.
(Cited on pp. 96, 97, 102, and 174.)

Ohad Kammar. *Algebraic Theory of Type-and-Effect Systems*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 2014.
(Cited on pp. 2 and 154.)

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 145–158, 2013.
(Cited on pp. 20, 42, 45, 47, 48, and 156.)

Edward A. Kmett. Trifecta (1.5.2), 2015.
http://hackage.haskell.org/package/trifecta-1.5.2.
(Cited on page 64.)

Pepijn Kokke and Wouter Swierstra. Auto in Agda. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, pp. 276–301. Springer, 2015.
(Cited on page 173.)

Vasileios Koutavas, Blain Paul Levy, and Eijiro Sumii. Limitations of applicative bisimulation (preliminary report). Technical report, Dagstuhl Seminar Proceedings 10351, 2010. Modelling, Controlling and Reasoning about State.
http://drops.dagstuhl.de/opus/volltexte/2010/2807.
(Cited on pp. 151 and 152.)

Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 205–217, 2017.
(Cited on page 174.)

Soren Bogh Lassen. Relational reasoning about contexts. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pp. 91–135. Cambridge University Press, 1998a.
(Cited on pp. 20, 69, 74, 91, and 92.)

Soren Bogh Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, University of Aarhus, 1998b.
(Cited on pp. 20, 75, 93, and 155.)

Paulson C. Lawrence, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Formal Aspects of Comp.*, 2019. doi:10.1007/s00165-019-00492-1.
(Cited on page 5.)

Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. Gmeta: A generic formal metatheory framework for first-order representations. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pp. 436–455. Springer, 2012.
(Cited on page 25.)

Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, pp. 100–126, 2014. doi:10.4204/EPTCS.153.8.
(Cited on pp. 48, 64, and 66.)

Daan Leijen. Type directed compilation of row-typed algebraic effects. In Andrew D. Gordon, editor, *POPL*. ACM, 2017.
(Cited on pp. 3, 14, and 45.)

Daan Leijen. Algebraic effect handlers with resources and deep finalization. Technical report, Microsoft Research, 2018. MSR-TR-2018-10.
(Cited on pp. 153, 174, and 176.)

Daan Leijen and Paolo Martini. Parsec (3.1.9), 2015. `http://hackage.haskell.org/package/parsec-3.1.9`.
(Cited on page 64.)

Xavier Leroy. A locally nameless solution to the POPLMARK challenge, 2007. Research Report RR-6098. `https://hal.inria.fr/inria-00123945/en/`.
(Cited on page 92.)

Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.
(Cited on pp. 8, 32, 70, 116, and 154.)

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pp. 333–343, 1995.
(Cited on page 17.)

Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 500–514, 2017. doi:10.1145/3093333.3009897.
(Cited on pp. 3, 7, 44, 47, 115, 116, 122, and 150.)

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL'88*, pp. 47–57, 1988. doi:10.1145/73560.73564.
(Cited on page 2.)

Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., 1994. ISBN 0-19-853835-9.

(Cited on pp. 5, 22, and 26.)

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 2nd edition, 1971. ISBN 0-387-98403-8.
(Cited on page 13.)

Simon Marlow. Haskell 2010 language report.
https://www.haskell.org/onlinereport/haskell2010/, 2010.
(Cited on pp. 12 and 31.)

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.
(Cited on pp. 5 and 26.)

Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.
(Cited on pp. 4, 20, 69, 84, 92, and 152.)

Cristina Matache. Program Equivalence for Algebraic Effects via Modalities. Master's thesis, St Cross College, University of Oxford, 2018.
(Cited on pp. 152, 154, and 155.)

Conor McBride. Kleisli arrows of outrageous fortune. Unpublished. Available at https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf (Accessed November 2019), 2011.
(Cited on page 175.)

Conor McBride. Shonky, 2016. https://github.com/pigworker/shonky.
(Cited on page 64.)

Conor McBride. Category Theory Library shipped with CS410-18, 2018.
https://github.com/pigworker/CS410-18/tree/master/Lib/Cat.
(Cited on page 112.)

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
(Cited on pp. 5 and 24.)

Craig McLaughlin, James McKinna, and Ian Stark. Triangulating context lemmas. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pp. 102–114, 2018. doi:10.1145/3167081.

(Not cited.)

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pp. 124–144. Springer, 1991.
(Cited on page 20.)

Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.
(Cited on pp. 4, 21, 69, 77, 90, 92, 151, and 156.)

Robin Milner. A theory of type polymorphism in programming. *17*, 3:348–375, 1978.
(Cited on pp. 54 and 65.)

Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23, 1989a.
(Cited on pp. 12 and 32.)

Eugenio Moggi. An abstract view of programming languages. Technical report, University of Edinburgh, 1989b. ECS-LFCS-90-113.
(Cited on page 17.)

Alberto Momigliano, Simon Ambler, and Roy Crole. A Hybrid Encoding of Howe's Method for Establishing Congruence of Bisimilarity. *ENTCS*, 70(2):60–75, 2002. doi:10.1016/S1571-0661(04)80506-1.
(Cited on page 93.)

Alberto Momigliano, Brigitte Pientka, and David Thibodeau. A case study in programming coinductive proofs: Howe's Method. *Mathematical Structures in Computer Science*, pp. 1–35, 2018. doi:10.1017/S0960129518000415.
(Cited on page 93.)

James Hiram Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
(Cited on pp. 3, 20, 75, and 156.)

Hiroshi Nakano. Fixed-point logic with the approximation modality and its Kripke completeness. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, pp. 165–182.

Springer, October 2001. URL http://www602.math.ryukoku.ac.jp/~nakano/papers/modality-tacs01.pdf.
(Cited on pp. 97 and 112.)

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pp. 166–178. ACM, 2015. doi:10.1145/2784731.2784764.
(Cited on page 156.)

Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. ISBN 978-3-319-35759-1.
(Cited on page 22.)

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.
(Cited on pp. 5 and 27.)

Ulf Norell. Dependently typed programming in Agda. In *AFP Summer School*, pp. 230–266. Springer, 2009.
(Cited on page 70.)

Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pp. 328–345. Springer, 1993.
(Cited on pp. 5 and 22.)

Tomas Petricek and Don Syme. Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming. In *Proceedings of Practical Aspects of Declarative Languages*, PADL 2011, 2011.
(Cited on page 65.)

Frank Pfenning. Lecture notes on bidirectional type checking. 15-312: Foundations of Programming Languages.(Accessed November 2019), 2004.
(Cited on page 39.)

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Automated Deduction — CADE-16*, pp. 202–206. Springer, 1999.

(Cited on page 153.)

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pp. 371–382, 2008.
(Cited on pp. 22 and 93.)

Brigitte Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In *Proceedings of the 10th International Conference on Functional and Logic Programming*, FLOPS'10, pp. 1–12, 2010.
(Cited on page 22.)

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.
(Cited on pp. 55, 84, and 201.)

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.
(Cited on page 39.)

A. M. Pitts. Operationally-based theories of program equivalence. In A. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pp. 241–298. Cambridge University Press, 1997.
(Cited on pp. 20 and 91.)

A. M. Pitts. Typed operational reasoning. In *Advanced Topics in Types and Programming Languages* Pierce [2004], chapter 7, pp. 245–289.
(Cited on pp. 8, 20, 69, 70, 73, 82, 84, 91, 96, and 116.)

A. M. Pitts. Step-indexed biorthogonality: a tutorial example. In A. Ahmed, N. Benton, L. Birkedal, and M. Hofmann, editors, *Modelling, Controlling and Reasoning About State*, number 10351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2010/2806`.
(Cited on pp. 112 and 135.)

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, Publications of the Newton Institute, pp. 227–273. Cambridge University Press, 1998.
(Cited on pp. 4, 8, 20, 21, 69, 77, 82, 84, 88, 90, 96, 124, and 152.)

Andrew Pitts. Howe's method for higher-order languages. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science, pp. 197–232. Cambridge University Press, 2011.
(Cited on pp. 20, 69, 74, 92, and 93.)

Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.
(Cited on page 153.)

Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pp. 1–24. Springer, 2001.
(Cited on pp. 2, 7, 13, 29, and 152.)

Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pp. 342–356. Springer, 2002.
(Cited on pp. 2, 13, 15, 16, 29, and 152.)

Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, Feb 2003.
(Cited on pp. 13, 15, and 17.)

Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9((4:23)), 2013.
(Cited on pp. 2, 3, 7, 18, 19, and 29.)

Piotr Polesiuk. IxFree : Step-Indexed Logical Relations in Coq. `http://www.ii.uni.wroc.pl/~ppolesiuk/papers/coqpl17.pdf`, 2017. Talk presented at The Third International Workshop on Coq for Programming Languages, Paris, France.
(Cited on pp. 96, 101, 111, and 173.)

Matija Pretnar. *Logic and Handling of Algebraic Effects*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 2010.
(Cited on pp. 2, 16, 20, 113, 154, and 155.)

John C. Reynolds. The meaning of types — from intrinsic to extrinsic semantics. Technical report, Department of Computer Science, University of Aarhus, 2000. BRICS-RS-00-32.
(Cited on page 23.)

Davide Sangiorgi. *Coinduction and the duality with induction*, pp. 28–88. Cambridge University Press, 2011. doi:10.1017/CBO9780511777110.004.
(Cited on page 106.)

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: What binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, pp. 98–113. ACM, 2019. doi:10.1145/3331545.3342595.
(Cited on page 18.)

Filip Sieczkowski, Aleš Bizjak, and Lars Birkedal. ModuRes: A Coq Library for Modular Reasoning about Concurrent Higher-Order Imperative Programming Languages. In Christian Urban and Xingyuan Zhang, editors, *Interative Theorem Proving*, pp. 375–390. Springer, 2015.
(Cited on pp. 102, 111, and 174.)

Alex Simpson and Niels Voorneveld. Behavioural equivalence via modalities for algebraic effects. In Amal Ahmed, editor, *Programming Languages and Systems*, ESOP 2018, pp. 300–326. Springer, 2018.
(Cited on pp. 152 and 155.)

Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Inc., 2006.
(Cited on pp. 5 and 55.)

Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994.
(Cited on pp. 4, 20, 21, 77, 79, 90, 92, and 152.)

Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.
(Cited on page 86.)

Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP):105:1–105:28, August 2019.
(Cited on page 98.)

Tarmo Uustalu. Stateful runners of effectful computations. *Electronic Notes in Theoretical Computer Science*, 319:403–421, 2015. ISSN 1571-0661. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).
(Cited on page 176.)

Jérôme Vouillon. A solution to the POPLMark challenge based on de bruijn indices. *Journal of Automated Reasoning*, 49(3):327–362, 2012.
(Cited on page 92.)

Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, pp. 24–52. Springer, 1995.
(Cited on page 12.)

Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1st edition, 1994. ISBN 0262231697.
(Cited on page 2.)

Jeremy Yallop et al. Effects bibliography. (Accessed November 2019). `https://github.com/yallop/effects-bibliography`, 2019.
(Cited on page 62.)

Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, January 2019.
(Cited on pp. 66, 113, and 169.)