Normalization for algebraic effect handlers

Normalizacja dla handlerów efektów algebraicznych

Wiktor Kuchta

Praca licencjacka

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

23 czerwca 2022

Abstract

Algebraic effects and handlers are gaining popularity in the theory and practice of programming languages as they provide unparalleled ability to define and combine computational effects. In this work, we demonstrate a novel proof of termination of evaluation and type soundness for a λ-calculus with deep effect handlers and a type-and-effect system that disallows recursive effects. The key idea is a fixed-point construction of logical relations for the language.

Efekty algebraiczne i ich handlery zyskują na popularności w teorii i praktyce języków programowania dzięki ich niezrównanym możliwościom definiowania i łączenia efektów obliczeniowych. W niniejszej pracy prezentujemy nowy dowód terminacji ewaluacji i poprawności typów dla rachunku λ z głębokimi handlerami oraz systemem typów i efektów zabraniającym rekurencyjnych efektów. Kluczowym pomysłem jest stałopunktowa konstrukcja relacji logicznych dla języka.

Contents

Chapter 1

Introduction

A programming language needs to be more than just a lambda calculus, capable only of functional abstraction and evaluation of expressions. Programs need to have an effect on the outside world and, thinking more locally, in our programs we would like to have fragments that do not merely reduce to a value in isolation, but also nontrivially affect the execution of surrounding code. This is what we call computational effects, the typical examples of which are: input/output, mutable state, exceptions, nondeterminism, and coroutines.

Today, we are at the mercy of programming language designers and we can only hope they provide the effects we would like to use and make sure that they all interact with each other well. A traditional way to remedy this is to (purely functionally) model effects using monads [Wad90]. This requires the use of monadic style, which among other things causes stratification of code into two styles and does not always cooperate well with the language's native facilities. Moreover, monads do not compose in general, so modular programming with monads has some overheads.

1.1 Algebraic effects

In recent years, a new promising approach has emerged: *algebraic effects and handlers* [PP13]. A language featuring algebraic effects lets programmers express all common computational effects as library code and use them in the usual direct style. This is done by decoupling effects into purely syntactic *operations* with their type signatures and giving them meaning separately, here by using *effect handlers*.

Effect handlers generalize exception handlers. We can *perform* an operation, just as we can raise an exception in a typical high-level programming language. The operation is then handled by the nearest enclosing *handler* for the specific operation. The handler receives the value given at perform-point and also, unlike normal exception handlers, a *resumption* – a first-class functional value representing the rest of the code to execute inside the handler from the perform-point. By calling the resumption with a value we can resume at the perform-point as if performing the operation evaluated to the value. But more interestingly, we can resume multiple times, or never, or store the resumption for later use.

To prevent crashes because of unhandled operations and aid understanding of effectful code, languages with algebraic effects typically have powerful *type-and-effect systems*,[1] not only tracking the type of value an expression might evaluate to, but

[1] We will often say just "type" when referring to both types and effects.

also what kind of effects it may perform along the way. Type-and-effect systems allow us to deduce some basic information about programs, for example that a pure expression cannot evaluate to two different values, or, in some systems, that it will *always* evaluate to a value.

1.2 Contributions

In this work we demonstrate a (to our knowledge) novel proof of termination of evaluation (also known as normalization) for a language [PPS19] equipped with algebraic effects. We obtain type soundness as a corollary. We also explain how the proof can carry over to other calculi. Further, we present code that mirrors the computational content of the proof.

Termination of evaluation might be of interest for several reasons, for example, it can simplify formal semantics and reasoning, just by virtue of eliminating nontermination as a case to consider. It can also be regarded as another safety property, just like "never crashing" (though in practice nonterminating programs are indistinguishable from very long-running ones). Although most programming languages support recursion, loops, and other features allowing for nontermination (e.g., recursive types), it might be useful to know that fragments that do not use these features cannot be blamed for it.

Nontermination is sometimes also viewed as a computational effect, which however does not neatly fit the algebraic effect framework. Despite that, it can still be added as a built-in effect to be tracked by the effect system, provided that we have sufficiently identified possible sources of nontermination. This is done, for example, in the Koka programming language [Lei14].

Normalization also implies that the calculus is sound as a logic, looking through the lens of the Curry-Howard isomorphism [SU06]. However, it is hard to give logical interpretations to effect annotations [KS07] and it is unclear how useful it is to treat calculi with algebraic effects as logics.

The primary tool for proving normalization (and many other properties) of typed formal calculi is *logical relations* [Sko19].[2] Logical relations are type-indexed relations on terms that essentially help carry additional information about *local* behavior of terms to make a type derivation-directed proof of the *global* property (such as normalization) have strong enough induction hypotheses and ultimately go through.

In this work we define direct-style logical relations to prove termination of evaluation in our language. A novelty of our approach is that the definition of the logical relations is recursive, more concretely, it is a solution to a fixed-point equation. The fixed-point construction replaces the use of step-indexing which appears in the logical relation construction we are most directly inspired by [Bie+18].

1.3 Acknowledgements

I would like to thank Dariusz Biernacki, Filip Sieczkowski, and Piotr Polesiuk for their help with this work.

[2] In the context of normalization, logical relations have many names, including: (Tait's) reducibility predicates, reducibility candidates, and saturated sets.

Chapter 2

The language

We will study the deep handler calculus and type-and-effect system formulated in [PPS19]. It is a refreshingly minimal language – the call-by-value lambda calculus with a few extensions to be able to express the essence of algebraic effects. There is only the universal operation, performed $\mathsf{do}\ v$. To be able to reach beyond the closest effect handler, the *lift* operator, written $[e]$, is introduced, which makes operations in e skip the nearest handler. In contrast to much work on algebraic effects, the effect-tracking system here is structural – signatures of effects do not have to be separately defined and named before their use. Finally, the type system features polymorphic expressions and polymorphic operations.

2.1 Untyped syntax and semantics

The syntax of the calculus is shown in fig. 2.1. We use metavariables x and r to refer to expression variables. As is standard, we work modulo α-equivalence and distinguish separate instances of metavariables by using subscripts, superscripts, or adding primes. Capture-avoiding substitution of v for x in e is denoted $e\{v/x\}$.

Evaluation contexts are expressions with a hole ($\square$) and encode a left-to-right evaluation order. We use the standard notation $K[x]$ for substituting x for the hole in K. There is a concept of *freeness* of evaluation contexts (fig. 2.2), a quantity increased by lifts and decreased by handlers. Intuitively, an evaluation context is n-free if an operation performed inside the context (in place of the hole) would be handled by the n-th handler outside of the context.

Naturally, freeness is used in the operational semantics (fig. 2.3), where we can see that effect handling occurs if there is a 0-free context between the operation and the handler. The resumption v_c wraps the context K with the same handler, which is why the effect handlers are called *deep* – all operations are handled, not just the first one (as in so-called *shallow* handlers). The other rules are the standard β-reduction of the λ-calculus and reductions in cases where an (effect-free) value is wrapped by an effect-related construct. It is straightforward to see that reduction is deterministic and compatible with respect to evaluation contexts.

2.2 Type-and-effect system

In fig. 2.4 the type system is introduced. We have three kinds: types, effects, and rows. We use α to range over type variables and Δ to range over type contexts, which assign kinds to type variables. Similarly, contexts Γ assign types to term-level variables. We

$$
\begin{array}{rll}
v, u & ::= x \mid \lambda x.\, e & \text{(values)} \\
e & ::= v \mid e\ e \mid [e] \mid \mathsf{do}\ v \mid \mathsf{handle}\ e\,\{x, r.\, e;\ x.\, e\} & \text{(expressions)} \\
K & ::= \Box \mid K\ e \mid v\ K \mid [K] \mid \mathsf{handle}\ K\,\{x, r.\, e;\ x.\, e\} & \text{(evaluation contexts)}
\end{array}
$$

Figure 2.1: Syntax.

$$
\frac{}{0\text{-free}(\Box)} \qquad \frac{n\text{-free}(K)}{n\text{-free}(K\ e)} \qquad \frac{n\text{-free}(K)}{n\text{-free}(v\ K)} \qquad \frac{n\text{-free}(K)}{n+1\text{-free}([K])}
$$

$$
\frac{n+1\text{-free}(K)}{n\text{-free}(\mathsf{handle}\ K\,\{x, r.\, e_h;\ x.\, e_r\})}
$$

Figure 2.2: Evaluation context freeness.

furthermore have the concept of *type-level substitutions*: we write $\Delta \vdash \delta :: \Delta'$ if δ maps type variables in Δ' to types of the corresponding kinds, well-formed in Δ. We assume that all types are well-formed (well-kinded) in the appropriate contexts in the sense of fig. 2.5.

Type-level substitutions are used only in the rule for do to instantiate the polymorphic variables Δ' of effect $\Delta'.\, \tau_1 \Rightarrow \tau_2$. Complementarily, the rule for handle states that the effect handler clause e_h has to be essentially parametric in Δ'.

We can intuitively understand effect rows as follows: if $e : \tau \,/\, \rho$ and effect ε appears at position n in ρ, then subexpressions performing ε are wrapped by n lifts inside e. Clearly, the order of effects in the row matters. The type system enables *row polymorphism* since we can have a row ending in a polymorphic type variable. Such rows are usually called *open* (and *closed* otherwise).

In fig. 2.6 the subtyping rules are introduced. Their main purpose is to allow effect rows to be subsumed by rows with more effects at the end. For that to be of use in more places, we also have subsumption rules for universal types and function types. In particular, closed rows are subsumed by open rows.

$$
\frac{e_1 \mapsto e_2}{K[e_1] \to K[e_2]} \qquad (\lambda x.\, e)\ v \mapsto e\{v/x\} \qquad [v] \mapsto v
$$

$$
\frac{0\text{-free}(K) \qquad v_c = \lambda z.\, \mathsf{handle}\ K[z]\,\{x, r.\, e_h;\ x.\, e_r\}}{\mathsf{handle}\ K[\mathsf{do}\ v]\,\{x, r.\, e_h;\ x.\, e_r\} \mapsto e_h\{v/x\}\{v_c/r\}}
$$

$$
\mathsf{handle}\ v\,\{x, r.\, e_h;\ x.\, e_r\} \mapsto e_r\{v/x\}
$$

Figure 2.3: Contraction ($\mapsto$) and single-step reduction ($\to$).

$$\kappa ::= \mathsf{T} \mid \mathsf{E} \mid \mathsf{R} \qquad \sigma, \tau, \varepsilon, \rho ::= \alpha \mid \tau \to_\rho \tau \mid \forall \alpha :: \kappa. \tau \mid \iota \mid \Delta. \tau \Rightarrow \tau \mid \varepsilon \cdot \rho$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau \,/\, \iota} \qquad \frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,/\, \rho}{\Delta; \Gamma \vdash \lambda x. e : \tau_1 \to_\rho \tau_2 \,/\, \iota}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to_\rho \tau_2 \,/\, \rho \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \,/\, \rho}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2 \,/\, \rho} \qquad \frac{\Delta \vdash \varepsilon :: \mathsf{E} \qquad \Delta; \Gamma \vdash e : \tau \,/\, \rho}{\Delta; \Gamma \vdash [e] : \tau \,/\, \varepsilon \cdot \rho}$$

$$\frac{\Delta, \alpha :: \kappa; \Gamma \vdash e : \tau \,/\, \iota}{\Delta; \Gamma \vdash e : \forall \alpha :: \kappa. \tau \,/\, \iota} \qquad \frac{\Delta \vdash \sigma :: \kappa \qquad \Delta; \Gamma \vdash e : \forall \alpha :: \kappa. \tau \,/\, \rho}{\Delta; \Gamma \vdash e : \tau\{\sigma/\alpha\} \,/\, \rho}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \rho_1 <: \rho_2 \qquad \Delta; \Gamma \vdash e : \tau_1 \,/\, \rho_1}{\Delta; \Gamma \vdash e : \tau_2 \,/\, \rho_2}$$

$$\frac{\Delta; \Gamma \vdash v : \delta(\tau_1) \,/\, \iota \qquad \Delta \vdash \delta :: \Delta' \qquad \Delta \vdash \Delta'. \tau_1 \Rightarrow \tau_2 :: \mathsf{E}}{\Delta; \Gamma \vdash \mathsf{do}\ v : \delta(\tau_2) \,/\, (\Delta'. \tau_1 \Rightarrow \tau_2)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \tau \,/\, (\Delta'. \tau_1 \Rightarrow \tau_2) \cdot \rho \\ \Delta, \Delta'; \Gamma, x : \tau_1, r : \tau_2 \to_\rho \tau_r \vdash e_h : \tau_r \,/\, \rho \qquad \Delta; \Gamma, x : \tau \vdash e_r : \tau_r \,/\, \rho\end{array}}{\Delta; \Gamma \vdash \mathsf{handle}\ e\ \{x, r. e_h; x. e_r\} : \tau_r \,/\, \rho}$$

Figure 2.4: Type system.

$$\frac{\alpha :: \kappa \in \Delta}{\Delta \vdash \alpha :: \kappa} \qquad \frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta \vdash \rho :: \mathsf{R} \qquad \Delta \vdash \tau_2 :: \mathsf{T}}{\Delta \vdash \tau_1 \to_\rho \tau_2 :: \mathsf{T}} \qquad \frac{\Delta, \alpha :: \kappa \vdash \tau :: \mathsf{T}}{\Delta \vdash \forall \alpha :: \kappa. \tau :: \mathsf{T}}$$

$$\frac{}{\Delta \vdash \iota :: \mathsf{R}} \qquad \frac{\Delta \vdash \varepsilon :: \mathsf{E} \qquad \Delta \vdash \rho :: \mathsf{R}}{\Delta \vdash \varepsilon \cdot \rho :: \mathsf{R}} \qquad \frac{\Delta, \Delta' \vdash \tau_1 :: \mathsf{T} \qquad \Delta, \Delta' \vdash \tau_2 :: \mathsf{T}}{\Delta \vdash \Delta'. \tau_1 \Rightarrow \tau_2 :: \mathsf{E}}$$

Figure 2.5: Well-formedness of types and rows.

$$\frac{}{\Delta \vdash \sigma <: \sigma} \qquad \frac{\Delta \vdash \rho :: \mathsf{R}}{\Delta \vdash \iota <: \rho} \qquad \frac{\Delta \vdash \rho_1 <: \rho_2}{\Delta \vdash \varepsilon \cdot \rho_1 <: \varepsilon \cdot \rho_2}$$

$$\frac{\Delta \vdash \tau_2^1 <: \tau_1^1 \qquad \Delta \vdash \rho_1 <: \rho_2 \qquad \Delta \vdash \tau_1^2 <: \tau_2^2}{\Delta \vdash \tau_1^1 \to_{\rho_1} \tau_1^2 <: \tau_2^1 \to_{\rho_2} \tau_2^2} \qquad \frac{\Delta, \alpha :: \kappa \vdash \tau_1 <: \tau_2}{\Delta \vdash \forall \alpha :: \kappa. \tau_1 <: \forall \alpha :: \kappa. \tau_2}$$

Figure 2.6: Subtyping.

Chapter 3

The logical relation

The logical relation is inspired by [Bie+18]. Some changes are due to language differences: we have only one universal operation which simplifies the treatment of effects, polymorphism does not manifest at the expression level (we do not have type lambdas), and our operations can be polymorphic. Instead of a binary step-indexed relation, our goal is to build a unary relation without step-indexing.

3.1 Definition

We begin with defining the interpretations of kinds. We call them the spaces of *semantic types* or, in the specific cases of E and R, *semantic effects*:

$$[\![\mathsf{T}]\!] = \mathcal{P}(\mathrm{CVal}) = \mathsf{Type}$$
$$[\![\mathsf{E}]\!] = [\![\mathsf{R}]\!] = \mathcal{P}(\mathrm{CExp} \times \mathbb{N} \times \mathsf{Type}) = \mathsf{Eff}$$

We write CVal and CExp for the sets of closed values and closed expressions respectively, so elements of Type are simply sets of closed values. Semantic effects are sets of triples that aim to describe a situation in which an expression tries to perform an effect but there is no handler for it – it gets *control-stuck*. The components of such a triple are: the operation with its argument, the freeness of the enclosing context, and the semantic type of values we could resume with – the return type of the operation.

Generalizing the preceding, type contexts are interpreted as mappings from type variables to semantic types:

$$[\![\Delta]\!] = \{\eta \mid \mathrm{dom}(\eta) = \mathrm{dom}(\Delta) \wedge \forall \alpha :: \kappa \in \Delta.\, \eta(\alpha) \in [\![\kappa]\!]\}$$

The logical relations, defined by (mutual) structural induction, appear in fig. 3.1. They are parameterized by a mapping η from type variables to semantic types.

The interpretation of a type variable is directly retrieved from the environment η. The treatment of function types is standard, except effect annotations also have to be taken into account. The definition may not look structurally recursive at first, but we can consider τ_2/ρ to be a subterm of $\tau_1 \to_\rho \tau_2$, just rearranged to a different notation. A universal type is interpreted, as usual, as the intersection of the interpretations for all possible choices of the semantic type for the type variable.

Dually, we can see that a polymorphic effect is interpreted as the *union* of the interpretations of the effect for all possible choices of the semantic types for the type variables. The interpretation of a single operation is implicitly treated as if it

$$
\begin{aligned}
[\![\alpha]\!]_\eta &= \eta(\alpha) \\
[\![\tau_1 \to_\rho \tau_2]\!]_\eta &= \{\lambda x.\, e \mid \forall v \in [\![\tau_1]\!]_\eta.\, e\{v/x\} \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta\} \\
[\![\forall \alpha :: \kappa.\, \tau]\!]_\eta &= \{v \mid \forall \mu \in [\![\kappa]\!].\, v \in [\![\tau]\!]_{[\alpha \mapsto \mu]\eta}\} \\
[\![\Delta.\, \tau_1 \Rightarrow \tau_2]\!]_\eta &= \{(\mathsf{do}\ v, 0, [\![\tau_2]\!]_{\eta\eta'}) \mid \eta' \in [\![\Delta]\!] \wedge v \in [\![\tau_1]\!]_{\eta\eta'}\} \\
[\![\varepsilon \cdot \rho]\!]_\eta &= [\![\varepsilon]\!]_\eta \cup \{(e, n+1, \mu) \mid (e, n, \mu) \in [\![\rho]\!]_\eta\} \\
[\![\iota]\!]_\eta &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\tau/\rho]\!]_\eta &= \{e \mid \exists v \in [\![\tau]\!]_\eta.\, e \to^* v \vee \exists e' \in \mathcal{S}[\![\tau/\rho]\!]_\eta.\, e \to^* e'\} \\
\mathcal{S}[\![\tau/\rho]\!]_\eta &= \{K[e] \mid \exists n, \mu.\, (e, n, \mu) \in [\![\rho]\!]_\eta \wedge n\text{-free}(K) \wedge \forall u \in \mu.\, K[u] \in \mathcal{E}[\![\tau/\rho]\!]_\eta\}
\end{aligned}
$$

Figure 3.1: Interpretation of types. Relations on expressions and stuck terms.[1]

were at the beginning of a row, so the freeness is 0. When we prepend to a row, we increase this freeness for every element of its semantic effect and combine with the interpretation of the prepended effect. Naturally, the semantic effect for the empty row is empty.

The definition of the set $\mathcal{E}[\![\tau/\rho]\!]_\eta$ is recursive and we interpret it inductively, i.e., as the least solution to the equations (we will elaborate on that soon). Note the similarity to the interpretation of function types in the definition of $\mathcal{S}[\![\tau/\rho]\!]_\eta$. The intuition is that an element of $\mathcal{E}[\![\tau/\rho]\!]_\eta$ evaluates to a value of type τ while possibly performing an effect in ρ and getting control-stuck finitely many times along the way. When the term is control-stuck, we do not have a handler, but we do know the return type of the operation, so we resume with all possible values in parallel.

This is a very local view of effectful computations, perhaps giving the false impression that all handlers immediately resume with a value and do nothing more (this is called the *reader* effect and is equivalent to dynamically-scoped variables). In reality, much can happen between performing the operation and it finally returning. Nevertheless, this local view suffices for our purposes and accurately reflects how much of the computation remains in the scope of a handler throughout evaluation.

More graphically (see fig. 3.2), we can associate with an element in $\mathcal{E}[\![\tau/\rho]\!]_\eta$ what we will call a *reduction trace* – a tree where nodes are expressions in $\mathcal{E}[\![\tau/\rho]\!]_\eta$, leaves are values in $[\![\tau]\!]_\eta$, and edges are either (maximal) multi-step reductions or branches from $\mathcal{S}[\![\tau/\rho]\!]_\eta$ for each element of μ, which substitute the value for the operation.[2] Different choices of values to resume with can lead to getting stuck a different number of times, so the depth of the tree is not uniform, and it can even be infinite. Still, these trees are well-founded thanks to the choice of the inductive interpretation.

To describe the construction in more detail, we temporarily overload notation

[1] Technically, the terms in $\mathcal{S}[\![\tau/\rho]\!]_\eta$ do not have to be control-stuck, since the (e, n, μ) could come from the interpretation of an effect or row variable, and the space of semantic effects allows for any e. Such a term *is* stuck if the triple comes from the interpretation of an operation. We leave it that way to make adding multiple operations easier later. An analogous definition appears in [Bie+18].

[2] This is also how we could view the structure of values of the inductive type $\mathcal{E}[\![\tau/\rho]\!]_\eta$, had we formalized the definitions in a (dependent) type theory.

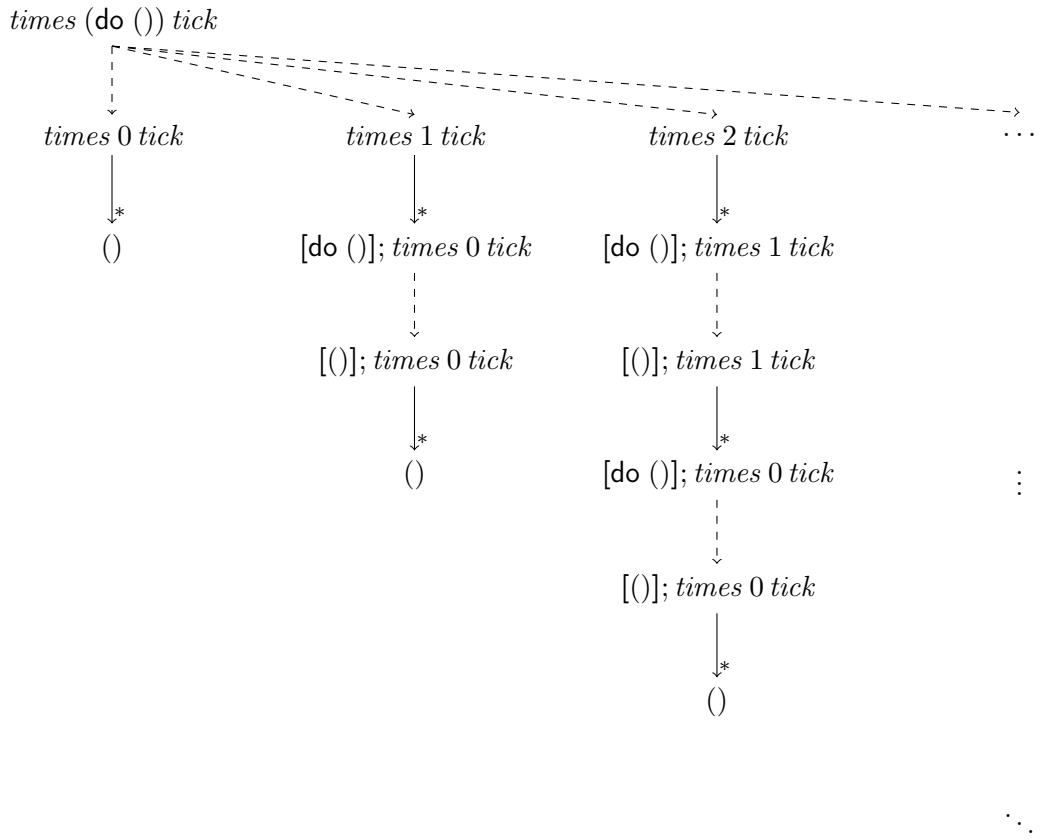

Figure 3.2: An illustration of the reduction trace for $\mathit{times}\,(\mathsf{do}\ ())\,\mathit{tick} \in \mathcal{E}[\![\mathsf{unit}/\mathsf{askNat} \cdot \mathsf{tick} \cdot \iota]\!]$, where $\mathsf{askNat} = \mathsf{unit} \Rightarrow \mathsf{nat}$ and $\mathsf{tick} = \mathsf{unit} \Rightarrow \mathsf{unit}$. For clarity, we assume the unit type, the inductive naturals, and a function *times* for invoking a thunk n times, but these can be realized using Church encodings. We set $\mathit{tick} = \lambda x.\,[\mathsf{do}\ ()]$. Here, the tick effect does not seem to do much, but a handler for it could, for instance, increment an external counter. The semicolon is standard syntax sugar: $e_1; e_2$ is expanded to $(\lambda x.\,e_2)\ e_1$ with x not free in e_2. The solid arrow is the usual reduction, while the dashed one represents substituting a value for the operation in a stuck term in $\mathcal{S}[\![\mathsf{unit}/\mathsf{askNat} \cdot \mathsf{tick} \cdot \iota]\!]$. This tree is well-founded despite having infinite depth.

and define operators $\mathcal{S}[\![\tau/\rho]\!]_\eta$ and $\mathcal{E}[\![\tau/\rho]\!]_\eta$ on sets of expressions (denoted by X).

$$\mathcal{E}[\![\tau/\rho]\!]_\eta(X) = \{e \mid \exists v \in [\![\tau]\!]_\eta.\, e \to^* v \vee \exists e' \in \mathcal{S}[\![\tau/\rho]\!]_\eta(X).\, e \to^* e'\}$$
$$\mathcal{S}[\![\tau/\rho]\!]_\eta(X) = \{K[e] \mid \exists n, \mu.\, (e, n, \mu) \in [\![\rho]\!]_\eta \wedge n\text{-free}(K) \wedge \forall u \in \mu.\, K[u] \in X\}$$

They are clearly monotone, so by the Knaster-Tarski theorem the fixed-point equation $\mathcal{E}[\![\tau/\rho]\!]_\eta(X) = X$ has a least solution.[3] Moreover, it can be characterized as the intersection of all $\mathcal{E}[\![\tau/\rho]\!]_\eta$-closed sets:

$$\mathcal{E}[\![\tau/\rho]\!]_\eta = \bigcap\{X \mid \mathcal{E}[\![\tau/\rho]\!]_\eta(X) \subseteq X\}$$
$$\mathcal{S}[\![\tau/\rho]\!]_\eta = \mathcal{S}[\![\tau/\rho]\!]_\eta(\mathcal{E}[\![\tau/\rho]\!]_\eta)$$

We immediately obtain the following principle:

Lemma 1 (Tarski induction principle)**.** *If $\mathcal{E}[\![\tau/\rho]\!]_\eta(X) \subseteq X$, then $\mathcal{E}[\![\tau/\rho]\!]_\eta \subseteq X$.*

By expanding out the definition of the function $\mathcal{E}[\![\tau/\rho]\!]_\eta$ and treating X as a predicate P, we get the more familiar principle of structural induction on $\mathcal{E}[\![\tau/\rho]\!]_\eta$.

Lemma 2 (Induction principle)**.** *Assume P is a predicate on closed expressions and*

- *if e evaluates to a value in $[\![\tau]\!]_\eta$, then $P(e)$ holds; and*
- *if e reduces to some $K[e']$ such that there exist $(e', n, \mu) \in [\![\rho]\!]_\eta$ such that n-free(K) and $P(K[u])$ holds for all $u \in \mu$, then $P(e)$ holds.*

Then $P(e)$ holds for all $e \in \mathcal{E}[\![\tau/\rho]\!]_\eta$.

We also note some straightforward but useful properties of the relations.

Lemma 3 (Value inclusion)**.** *For any τ and ρ we have $[\![\tau]\!]_\eta \subseteq \mathcal{E}[\![\tau/\rho]\!]_\eta$.*

Lemma 4 (Control-stuck inclusion)**.** *For any τ and ρ we have $\mathcal{S}[\![\tau/\rho]\!]_\eta \subseteq \mathcal{E}[\![\tau/\rho]\!]_\eta$.*

Lemma 5 (Closedness under antireduction)**.** *If $e \to^* e' \in \mathcal{E}[\![\tau/\rho]\!]_\eta$, then $e \in \mathcal{E}[\![\tau/\rho]\!]_\eta$.*

Lemma 6 (Weakening)**.** *If η' extends η, then $\mathcal{E}[\![\tau/\rho]\!]_\eta = \mathcal{E}[\![\tau/\rho]\!]_{\eta'}$.*

Lemma 7 (Monotonicity in types)**.** *If $[\![\tau_1]\!]_\eta \subseteq [\![\tau_2]\!]_\eta$ and $[\![\rho_1]\!]_\eta \subseteq [\![\rho_2]\!]_\eta$, then $\mathcal{S}[\![\tau_1/\rho_1]\!]_\eta \subseteq \mathcal{S}[\![\tau_2/\rho_2]\!]_\eta$ and $\mathcal{E}[\![\tau_1/\rho_1]\!]_\eta \subseteq \mathcal{E}[\![\tau_2/\rho_2]\!]_\eta$.*

3.2 Compatibility lemmas

We want to establish that $\vdash e : \tau \,/\, \iota$ implies $e \in \mathcal{E}[\![\tau/\iota]\!]$. For this purpose we will prove a semantic counterpart of each typing rule. First, we need to define a counterpart to the typing judgment. Unlike typing judgments, our relations are on closed terms only, so we get around that by using substitution. We define semantic entailment as follows:

$$\Delta; \Gamma \models e : \tau \,/\, \rho \iff \forall \eta \in [\![\Delta]\!].\, \forall \gamma \in [\![\Gamma]\!]_\eta.\, \gamma(e) \in \mathcal{E}[\![\tau/\rho]\!]_\eta,$$

where $[\![\Gamma]\!]_\eta = \{\gamma \mid \mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \wedge \forall x : \tau \in \Gamma.\, \gamma(x) \in [\![\tau]\!]_\eta\}$ contains expression-level variable substitutions.

[3] The operators are not Scott-continuous because of potentially infinite semantic types μ. This is why the construction from the Kleene fixed-point theorem (union of iterations of the operator on the empty set) would fail to be a solution. With this definition the reduction traces have bounded depth. The application compatibility lemma would fail: it grafts different reduction traces to the leaves of a potentially infinitely branching reduction trace, so the bounded depth is not preserved.

Lemma 8 (Variable compatibility)**.**

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \models x : \tau \,/\, \iota}$$

Proof. Assume $x : \tau \in \Gamma$. We want to prove $\Delta; \Gamma \models x : \tau \,/\, \iota$. Take any $\eta \in [\![\Delta]\!]$ and $\gamma \in [\![\Gamma]\!]_\eta$. We want to show $\gamma(x) \in \mathcal{E}[\![\tau/\iota]\!]_\eta$. From the definition of $[\![\Gamma]\!]_\eta$ we know that $\gamma(x) \in [\![\tau]\!]_\eta$, so by lemma 3 we have $\gamma(x) \in \mathcal{E}[\![\tau/\iota]\!]_\eta$. □

Lemma 9 (Abstraction compatibility)**.**

$$\frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta; \Gamma, x : \tau_1 \models e : \tau_2 \,/\, \rho}{\Delta; \Gamma \models \lambda x.\, e : \tau_1 \to_\rho \tau_2 \,/\, \iota}$$

Proof. Assume $\Delta \vdash \tau_1 :: \mathsf{T}$ and $\Delta; \Gamma, x : \tau_1 \models e : \tau_2 \,/\, \rho$. We want to prove $\Delta; \Gamma \models \lambda x.\, e : \tau_1 \to_\rho \tau_2 \,/\, \iota$. Take any $\eta \in [\![\Delta]\!]$ and $\gamma \in [\![\Gamma]\!]_\eta$. By lemma 3 it suffices to show $\gamma(\lambda x.\, e) = \lambda x.\, \gamma(e) \in [\![\tau_1 \to_\rho \tau_2]\!]_\eta$. So take any $v \in [\![\tau_1]\!]_\eta$. We need to show $\gamma(e)\{v/x\} \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$. Let $\gamma' = \gamma'[x \mapsto v]$. Then $\gamma' \in [\![\Gamma, x : \tau_1]\!]_\eta$, so $\gamma(e)\{v/x\} = \gamma'(e) \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$. □

For clarity of presentation, in the following we will assume Γ empty. The lemmas in full generality can then be proven simply by substituting an interpretation of Γ.

Lemma 10 (Lift compatibility)**.**

$$\frac{\Delta \vdash \varepsilon :: \mathsf{E} \qquad \Delta; \models e : \tau \,/\, \rho}{\Delta; \models [e] : \tau \,/\, \varepsilon \cdot \rho}$$

Proof. Assume $\Delta \vdash \tau :: \mathsf{T}$, $\Delta \vdash \varepsilon :: \mathsf{E}$, and $\Delta \vdash \rho :: \mathsf{R}$. Take any $\eta \in [\![\Delta]\!]$. We will show by induction on $e \in \mathcal{E}[\![\tau/\rho]\!]_\eta$ that $[e] \in \mathcal{E}[\![\tau/\varepsilon \cdot \rho]\!]_\eta$.

If $e \to^* K[e']$ and there exists $(e', n, \mu) \in [\![\rho]\!]_\eta$ such that n-free(K) and for all $u \in \mu$ the induction hypothesis holds for $K[u]$, then we have $(e', n+1, \mu) \in [\![\varepsilon \cdot \rho]\!]_\eta$, $n+1$-free$([K])$, and $\forall u \in \mu.\, [K[u]] \in \mathcal{E}[\![\tau/\varepsilon \cdot \rho]\!]_\eta$. So $[K[e']] \in \mathcal{S}[\![\tau/\varepsilon \cdot \rho]\!]_\eta$ and $[e] \in \mathcal{E}[\![\tau/\varepsilon \cdot \rho]\!]_\eta$ by antireduction.

If $e \to^* v \in [\![\tau]\!]_\eta$, then $[e] \to^* [v] \to v$, so $[e] \in \mathcal{E}[\![\tau/\varepsilon \cdot \rho]\!]_\eta$. □

Lemma 11 (Application compatibility)**.**

$$\frac{\Delta; \models e_1 : \tau_1 \to_\rho \tau_2 \,/\, \rho \qquad \Delta; \models e_2 : \tau_1 \,/\, \rho}{\Delta; \models e_1\, e_2 : \tau_2 \,/\, \rho}$$

Proof. Fix any well-formed Δ and $\tau_1 \to_\rho \tau_2$. Take any $\eta \in [\![\Delta]\!]$ and $e_2 \in \mathcal{E}[\![\tau_1/\rho]\!]_\eta$. We will show by induction on $e_1 \in \mathcal{E}[\![\tau_1 \to_\rho \tau_2/\rho]\!]_\eta$ that $e_1\, e_2 \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$.

If $e_1 \to^* K_1[e_1']$ and there exists $(e_1', n, \mu) \in [\![\rho]\!]_\eta$ such that n-free(K_1) and for all $u \in \mu$ the inductive hypothesis holds for $K_1[u]$, then $K_1[e_1']\, e_2 \in \mathcal{S}[\![\tau_2/\rho]\!]_\eta$, since n-free$(K_1\, e_2)$. By antireduction $e_1\, e_2 \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$.

Now assume $e_1 \to^* (\lambda x.\, e) \in [\![\tau_1 \to_\rho \tau_2/\rho]\!]_\eta$. We will show by induction on $e_2 \in \mathcal{E}[\![\tau_1/\rho]\!]_\eta$ that $(\lambda x.\, e)\, e_2 \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$ and the claim will follow by antireduction.

If $e_2 \to^* K_2[e_2']$ and there exists $(e_2', n, \mu) \in [\![\rho]\!]_\eta$ such that n-free(K_2) and for all $u \in \mu$ the inductive hypothesis holds for $K_2[u]$, then $(\lambda x.\, e)\, K_2[e_2'] \in \mathcal{S}[\![\tau_2/\rho]\!]_\eta$, since n-free$((\lambda x.\, e)\, K_2)$. By antireduction $(\lambda x.\, e)\, e_2 \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$.

If $e_2 \to^* v \in [\![\tau_1]\!]_\eta$, then $(\lambda x.\, e)\, e_2 \to^* (\lambda x.\, e)\, v \to e\{v/x\} \in \mathcal{E}[\![\tau_2/\rho]\!]_\eta$. □

Lemma 12 (Handle compatibility)**.**

$$\frac{\Delta; \models e : \tau \,/\, (\Delta'.\, \tau_1 \Rightarrow \tau_2) \cdot \rho \qquad \Delta, \Delta'; x : \tau_1, r : \tau_2 \to_\rho \tau_r \models e_h : \tau_r \,/\, \rho \qquad \Delta; x : \tau \models e_r : \tau_r \,/\, \rho}{\Delta; \models \mathsf{handle}\ e\ \{x, r.\, e_h; x.\, e_r\} : \tau_r \,/\, \rho}$$

Proof. Assume $\Delta, \Delta'; x : \tau_1, r : \tau_2 \to_\rho \tau_r \models e_h : \tau_r \,/\, \rho$ and $\Delta; x : \tau \models e_r : \tau_r \,/\, \rho$. Let h stand for $\{x, r.\, e_h;\ x.\, e_r\}$. Take any $\eta \in [\![\Delta]\!]$. We will show by induction on $e \in \mathcal{E}[\![\tau/(\Delta'.\, \tau_1 \Rightarrow \tau_2) \cdot \rho]\!]_\eta$ that $\mathsf{handle}\ e\, h \in \mathcal{E}[\![\tau_r/\rho]\!]_\eta$. Note that only τ_1 and τ_2 require Δ' to be in context.

If $e \to^* v \in [\![\tau]\!]_\eta$, then $\mathsf{handle}\ e\, h \to^* e_r\{v/x\} \in \mathcal{E}[\![\tau_r/\rho]\!]_\eta$, so the claim follows by antireduction.

Now assume $e \to^* K[e']$ and we have $(e', n, \mu) \in [\![(\Delta'.\, \tau_1 \Rightarrow \tau_2) \cdot \rho]\!]_\eta$ such that n-free(K) and for all $u \in \mu$ the induction hypothesis holds for $K[u]$.

If $n = 0$, then $(e', n, \mu) \in [\![\tau_1 \Rightarrow \tau_2]\!]_{\eta\eta'}$ for some $\eta' \in [\![\Delta']\!]$. More specifically, $e' = \mathsf{do}\ v$, $v \in [\![\tau_1]\!]_{\eta\eta'}$ and $\mu = [\![\tau_2]\!]_{\eta\eta'}$. We have $\mathsf{handle}\ e\, h \to^* \mathsf{handle}\ K[\mathsf{do}\ v]\, h \to e_h\{v/x\}\{v_c/r\}$, where $v_c = \lambda z.\, \mathsf{handle}\ K[z]\, h$. To show $v_c \in [\![\tau_2 \to_\rho \tau_r]\!]_{\eta\eta'}$, take any $u \in [\![\tau_2]\!]_{\eta\eta'}$ and show $\mathsf{handle}\ K[u]\, h \in \mathcal{E}[\![\tau_r/\rho]\!]_{\eta\eta'} = \mathcal{E}[\![\tau_r/\rho]\!]_\eta$. Which holds by induction hypothesis. Therefore, $e_h\{v/x\}\{v_c/r\}$ is in $\mathcal{E}[\![\tau_r/\rho]\!]_\eta$ and so is $\mathsf{handle}\ e\ h$.

If $n > 0$, then $\mathsf{handle}\ K[e']\, h \in \mathcal{S}[\![\tau_r/\rho]\!]_\eta$, since $n - 1$-free$(\mathsf{handle}\ K\, h)$, $(e', n - 1, \mu) \in [\![\rho]\!]_\eta$, and $\forall u \in \mu.\, \mathsf{handle}\ K[u]\, h \in \mathcal{E}[\![\tau_r/\rho]\!]_\eta$. Again, the claim follows by antireduction.

□

Lemma 13. *Syntactic type substitutions are compatible with semantic type environments: If*

- Δ *and* Δ' *have disjoint domains; and*
- $\Delta \vdash \delta :: \Delta'$*; and*
- $\Delta, \Delta' \vdash \tau :: \kappa$*; and*
- $\eta \in [\![\Delta]\!]$*; and*
- η' *extends* η *by mappings* $\alpha \mapsto [\![\delta(\alpha)]\!]_\eta$*;*

then $[\![\tau]\!]_{\eta'} = [\![\delta(\alpha)]\!]_\eta$.

Proof. By induction on the kinding rules.

If $\tau = \alpha \in \Delta$, then both sides are equal to $\eta(\alpha)$.

If $\tau = \alpha \in \Delta'$, then equality follows from the definition of η'.

If $\tau = \iota$, then both sides are empty.

If $\tau = \forall \alpha :: \kappa.\, \tau'$, then $[\![\tau]\!]_{\eta'} = \bigcap\{[\![\tau']\!]_{\eta'[\alpha \mapsto \mu]} \mid \mu \in [\![\kappa]\!]\}$ and $[\![\delta(\tau)]\!]_\eta = \bigcap\{[\![\delta(\tau')]\!]_{\eta[\alpha \mapsto \mu]} \mid \mu \in [\![\kappa]\!]\}$, which are equal by the inductive hypothesis (taking $\Delta, \alpha :: \kappa$ as Δ in the statement).

If $\tau = \Delta''.\, \tau_1 \Rightarrow \tau_2$, then $[\![\tau]\!]_{\eta'} = \{(e, 0, [\![\tau_2]\!]_{\eta'\eta''}) \mid \eta'' \in [\![\Delta'']\!] \wedge v \in [\![\tau_1]\!]_{\eta'\eta''}\}$ and $[\![\delta(\tau)]\!]_\eta = \{(e, 0, [\![\delta(\tau_2)]\!]_{\eta\eta''}) \mid \eta'' \in [\![\Delta'']\!] \wedge v \in [\![\delta(\tau_1)]\!]_{\eta\eta''}\}$, which are equal q'by induction (taking Δ, Δ'' as Δ in the statement).

If $\tau = \varepsilon \cdot \rho$, then $[\![\tau]\!]_{\eta'} = [\![\varepsilon]\!]_{\eta'} \cup \{(e, n+1, \mu) \mid (e, n, \mu) \in [\![\rho]\!]_{\eta'}\}$ and $[\![\delta(\tau)]\!]_\eta = [\![\delta(\varepsilon)]\!]_\eta \cup \{(e, n+1, \mu) \mid (e, n, \mu) \in [\![\delta(\rho)]\!]_\eta\}$, which are equal by the inductive hypothesis.

□

Lemma 14 (Do compatibility)**.**

$$\frac{\Delta; \models v : \delta(\tau_1) \,/\, \iota \qquad \Delta \vdash \delta :: \Delta' \qquad \Delta \vdash \Delta'.\, \tau_1 \Rightarrow \tau_2 :: \mathsf{E}}{\Delta; \models \mathsf{do}\ v : \delta(\tau_2) \,/\, (\Delta'.\, \tau_1 \Rightarrow \tau_2)}$$

Proof. Assume $\Delta \vdash \delta :: \Delta'$, $\Delta \vdash \Delta'.\, \tau_1 \to \tau_2 :: \mathsf{E}$. Take any $\eta \in [\![\Delta]\!]$. Assume $v \in \mathcal{E}[\![\delta(\tau_1)/\iota]\!]_\eta$. Clearly, $v \in [\![\delta(\tau_1)]\!]_\eta$. We want to show $\mathsf{do}\ v \in \mathcal{E}[\![\delta(\tau_2)/(\Delta'.\, \tau_1 \Rightarrow \tau_2)]\!]_\eta$.

By lemma 4 it suffices to show $\mathsf{do}\ v \in \mathcal{S}[\![\delta(\tau_2)/(\Delta'.\, \tau_1 \Rightarrow \tau_2)]\!]_\eta$. By taking the empty context in the definition of $\mathcal{S}$ and lemma 3, it suffices to show $(\mathsf{do}\ v, 0, [\![\delta(\tau_2)]\!]_\eta) \in [\![\Delta'.\, \tau_1 \Rightarrow \tau_2]\!]_\eta$. By the interpretation of polymorphic effects, it would be enough to show $(\mathsf{do}\ v, 0, [\![\delta(\tau_2)]\!]_\eta) \in [\![\tau_1 \Rightarrow \tau_2]\!]_{\eta'}$, where η' extends η by mappings $\alpha \mapsto [\![\delta(\alpha)]\!]_\eta$ for all $\alpha \in \Delta'$. In other words, we need $v \in [\![\tau_1]\!]_{\eta'}$ and $[\![\delta(\tau_2)]\!]_\eta = [\![\tau_2]\!]_{\eta'}$, which follows immediately from lemma 13. □

Lemma 15 (Subtyping compatibility)**.**

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \rho_1 <: \rho_2 \qquad \Delta; \models e : \tau_1 \,/\, \rho_1}{\Delta; \models e : \tau_2 \,/\, \rho_2}$$

Proof. By induction on subtyping rules we will show that if $\Delta \vdash \sigma_1 <: \sigma_2$, then $[\![\sigma_1]\!]_\eta \subseteq [\![\sigma_2]\!]_\eta$ for all $\eta \in [\![\Delta]\!]$. Then, by lemma 7, from $\Delta \vdash \tau_1 <: \tau_2$, $\Delta \vdash \rho_1 <: \rho_2$ and $\Delta; \Gamma \models e : \tau_1 \,/\, \rho_1$ we will be able to conclude $\Delta; \Gamma \models e : \tau_2 \,/\, \rho_2$.

For the case of the reflexivity rule, we obviously have $[\![\sigma]\!]_\eta \subseteq [\![\sigma]\!]_\eta$.

For the case of the function type rule, assume $[\![\tau_2^1]\!]_\eta \subseteq [\![\tau_1^1]\!]_\eta$, $[\![\rho_1]\!]_\eta \subseteq [\![\rho_2]\!]_\eta$, and $[\![\tau_1^2]\!]_\eta \subseteq [\![\tau_2^2]\!]_\eta$. We want to show $[\![\tau_1^1 \to_{\rho_1} \tau_1^2]\!]_\eta \subseteq [\![\tau_2^1 \to_{\rho_2} \tau_2^2]\!]_\eta$. So take any $(\lambda x.\, e)$ in the former and any $v \in [\![\tau_2^1]\!]$. Since $v \in [\![\tau_1^1]\!]_\eta$, we have $e\{v/x\} \in \mathcal{E}[\![\tau_1^2/\rho_1]\!]_\eta$. By lemma 7 we obtain $e\{v/x\} \in \mathcal{E}[\![\tau_2^2/\rho_2]\!]_\eta$ as desired.

For the case of the universal quantifier rule, assume $[\![\tau_1]\!]_{\eta[\alpha \mapsto \mu]} \subseteq [\![\tau_2]\!]_{\eta[\alpha \mapsto \mu]}$ for all $\mu \in [\![\kappa]\!]$. The claim holds since

$$\begin{aligned}[\![\forall \alpha :: \kappa.\, \tau_1]\!]_\eta &= \{v \mid \forall \mu \in [\![\kappa]\!].\, v \in [\![\tau_1]\!]_{[\alpha \mapsto \mu]\eta}\} \\ &\subseteq \{v \mid \forall \mu \in [\![\kappa]\!].\, v \in [\![\tau_2]\!]_{[\alpha \mapsto \mu]\eta}\} = [\![\forall \alpha :: \kappa.\, \tau_2]\!]_\eta.\end{aligned}$$

The case of the empty row rule holds trivially, since $[\![\iota]\!]_\eta = \emptyset$.

For the case of the row extension rule, assume $[\![\rho_1]\!]_\eta \subseteq [\![\rho_2]\!]_\eta$. We clearly have

$$\{(e, n+1, \mu) \mid (e, n, \mu) \in [\![\rho_1]\!]_\eta\} \subseteq \{(e, n+1, \mu) \mid (e, n, \mu) \in [\![\rho_2]\!]_\eta\},$$

so $[\![\varepsilon \cdot \rho_1]\!]_\eta \subseteq [\![\varepsilon \cdot \rho_2]\!]_\eta$ as well.

□

Lemma 16 (Polymorphism introduction compatibility)**.**

$$\frac{\Delta, \alpha :: \kappa; \models e : \tau \,/\, \iota}{\Delta; \models e : \forall \alpha :: \kappa.\, \tau \,/\, \iota}$$

Proof. Assume $\Delta, \alpha :: \kappa; \models e : \tau \,/\, \iota$. Take $\eta \in [\![\Delta]\!]$. We know e evaluates to a value in $[\![\tau]\!]_{\eta[\alpha \mapsto \mu]}$ for any $\mu \in [\![\kappa]\!]$. Therefore this value is in $[\![\forall \alpha :: \kappa.\, \tau]\!]_\eta$, and by antireduction $e \in \mathcal{E}[\![\forall \alpha :: \kappa.\, \tau/\iota]\!]_\eta$. □

Lemma 17 (Polymorphism elimination compatibility)**.**

$$\frac{\Delta \vdash \sigma :: \kappa \qquad \Delta; \models e : \forall \alpha :: \kappa.\, \tau \,/\, \rho}{\Delta; \models e : \tau\{\sigma/\alpha\} \,/\, \rho}$$

Proof. Assume $\Delta \vdash \sigma :: \kappa$ and $\Delta; \models e : \forall \alpha :: \kappa.\, \tau \,/\, \rho$. By lemma 13 we have $[\![\tau\{\sigma/\alpha\}]\!]_\eta = [\![\tau]\!]_{\eta[\alpha \mapsto [\![\sigma]\!]_\eta]}$, which is a superset of $[\![\forall \alpha :: \kappa.\, \tau]\!]$. So we have $\Delta; \models e : \tau\{\sigma/\alpha\} \,/\, \rho$ by lemma 7. □

Theorem 1 (Termination of evaluation)**.** *If $\vdash e : \tau \,/\, \iota$, then evalution of e terminates.*

Proof. Take a derivation of $\vdash e : \tau \,/\, \iota$. After replacing each typing rule by the corresponding compatibility lemma, we obtain $\models e : \tau \,/\, \iota$, so $e \in \mathcal{E}[\![\tau/\iota]\!]$. Therefore e has to terminate to a value, since $[\![\iota]\!]$ is empty and hence $\mathcal{S}[\![\tau/\iota]\!]$ is empty. □

This is also an alternative proof of soundness ("well-typed programs do not go wrong"), which was already shown using the technique of progress and preservation in [PPS19].

3.3 Extensions

The calculus considered so far is fairly simple. We will discuss extensions that bring it closer to other calculi with algebraic effects. Adapting the proofs to the changes poses no difficulty and is not interesting, so we will omit that.

A simple extension we can make is analogous to multiple prompts for delimited continuations: annotate each operation ($\mathsf{do}_l\ v$), handler ($\mathsf{handle}_l\ e\ h$), and lift ($[e]_l$) with a label l from a fixed set of labels $\mathcal{L}$. As a result, we obtain sets of completely independent effectful operators for each label. We would also need a notion of freeness per label, which would naturally be reflected in the type system as having different rows of effects per label.[4] The typing rule for an effectful operation with label l would only look at the row for l. The numbers representing freeness in the logical relations would also have to be indexed by labels, i.e., changed into functions from labels to natural numbers.

Another common feature in languages with algebraic effects is grouping multiple operations into one effect. We argue that this is not a significant change and is not deeply related to effects, as it could be simulated if the language featured generalized algebraic data types (GADTs) with pattern matching. If we want an effect ε with operations $\Delta'_i.\, op_i : \tau_i \Rightarrow \sigma_i$, we can define a GADT $e\ \alpha$ with constructors $op_i : \forall \Delta'_i.\, \tau_i \to e\ \sigma_i$ and set $\varepsilon = \alpha :: \mathsf{T}.\, e\ \alpha \Rightarrow \alpha$. A handler for this effect would have to pattern match on a value of type $e\ \alpha$ to refine the type variable α.

Nevertheless, we can sketch how we could accommodate grouping operations into one effect natively. We fix a set of operation names ranged over by op. Effects are now sets of distinct operation names together with their types:

$$\frac{\text{for each } i \in \{1 \dots n\} \qquad \Delta, \Delta'_i \vdash \tau_i :: \mathsf{T} \qquad \Delta, \Delta'_i \vdash \sigma_i :: \mathsf{T}}{\Delta \vdash \{op_i : \Delta'_i.\, \tau_i \Rightarrow \sigma_i \mid i \in \{1 \dots n\}\} :: \mathsf{E}}$$

Handlers would take on the form $\mathsf{handle}\ e\ \{op_1\ x\ r.\, e_1; \dots; op_n\ x\ r.\, e_n; x.\, e_r\}$ and $\mathsf{do}\ v$ would be generalized to $op\ v$. The typing rule for operations would also have to "guess"

[4] Alternatively: one row in which effects are tagged with labels and consecutive effects with different tags can be swapped using subsumption rules.

the other operations in an effect (in the original rule it already guesses the types in the operation and a type-level substitution). The rule for handle would naturally need a suitable premise for each operation clause.

When we have both features and identify labels with effects (groups of operations),[5] we arrive at a calculus very similar to the one in [Bie+18]. A crucial difference still remains: our effect types are structural. In particular, effects cannot be recursive. In the work cited it is conjectured that a restriction on recursive occurrences of effects would make their calculus terminating. Therefore, we claim that our method settles this conjecture positively. The interpretations of effects would be essentially the same as in the cited work (but unary) and the definitions of $\mathcal{E}$ and $\mathcal{S}$ would be inductive as in this work.

We consider the language in [Bie+18] to be a good representative calculus for algebraic effects – we want to stress that our normalization result is not just a peculiarity of the calculus chosen. In particular, it features polymorphism and allows duplicate instances of effects, both of which are sometimes omitted in the literature.

3.4 Related work

One thing that was not mentioned about [Bie+18], where the interpretation of effects and the relation on control-stuck terms come from, is that their relation is *biorthogonal* – there is an additional relation on (full program) evaluation contexts. Their relation on expressions is defined extensionally: an expression is "good" if it behaves "well" in all "good" contexts. In our work, noticing that this is unnecessary led to simpler definitions and proofs. More importantly, our intensional direct-style definition of good expressions leads to better intuition, in particular, allows us to consider reduction traces. However, opting for direct-style logical relations would possibly not simplify the work in [Bie+18] dramatically, as the relations are binary.

A work which is eerily similar to ours, albeit in a very different setting, is [Hur+12]. To reason about program equivalence, they define a relation $\mathcal{E}$ on expressions and a relation $\mathcal{S}$ on stuck terms by solving a recursive equation. But the terms are "stuck" for a different reason: they are applications to "external" functions about which we in a sense do not know enough, a situation reminiscent of normal form bisimulations. Furthermore, they interpret the recursive definition *coinductively*, i.e., as the greatest fixed point.

In our setting, a coinductive definition would theoretically allow expressions in $\mathcal{E}[\![\tau/\rho]\!]_\eta$ to have reduction traces that are ill-founded, i.e., contain infinite paths. The compatibility lemmas can be proven using strong coinduction[6] at the type in the rule conclusion, except for the one for handle. The problem is that we substitute a resumption (which contains a handler) for r, but to show that it is in $[\![\tau_2 \to_\rho \tau_r]\!]_\eta$ we would need exactly the claim we are trying to prove in the first place. The failure of handle compatibility should not be very surprising, as this is the rule responsible for eliminating effects – in particular, it should be able to take any expression in $\mathcal{E}[\![\tau/\tau_1 \Rightarrow \tau_2 \cdot \iota]\!]_\eta$ and show that wrapping it in a suitable handler produces an expression in $\mathcal{E}[\![\tau/\iota]\!]_\eta$, which cannot get stuck and simply evaluates to a value of type

[5] Technically, effects are formed in type contexts. Therefore, we also forbid effects to have free type variables, so that they can be formed at the "top-level", where the set of labels is also fixed. This is similar to effects in [Bie+18]. We also note that, with this restriction, effect rows could be represented simply as mappings from labels to natural numbers.

[6] Strong coinduction is the principle that $X \subseteq F(X \cup \nu F)$ implies $X \subseteq \nu F$, where F is a monotone operator on a complete lattice and νF is its greatest fixed point.

τ. This seems impossible, as theoretically the expression's reduction trace could have no leaves, so by definition of $\mathcal{E}$ the type τ could be arbitrary.

We have found only sketches of normalization proofs for algebraic effects in the literature, so we cannot be sure of how they work. Such a mention appears in [KLO13]. We speculate that their proof might have a combinatorial flavor, rather than use only logical relations. This is because multiple such arguments appear in a paper they cite [Lin07] and because of the mention that the handler is reapplied on a strict subterm of the original computation during each handling reduction. This is possible thanks to an unusual calculus, where the operation carries a continuation alongside the argument and there is a reduction that moves frames surrounding the expression to the continuation until the operation is directly inside the handler. It is feasible that such an argument could also work in our semantics by using a suitable term size function.

A calculus with effect handlers appears with a mention of a termination proof in [For+19]. As it is even terser than the previous one, we cannot give any comment.

A mechanized proof of strong normalization for the calculus defined in [For+19] is reported in [Ham19]. A tool for proof search is used and the proof is not spelled out in detail, but it also relies on combinatorial arguments and strikes us as more elaborate than ours.

We did not consider recursive effects, as they are known to introduce nontermination [Bie+18]. Likely, we could allow effects where all recursive occurrences are positive – but note that the left side of an effect arrow ($\Rightarrow$) is positive and the right is negative, as opposed to function arrows ($\to$). Needless to say, a least fixed point construction would have to be used in the interpretation of effects. This is discussed from the perspective of a translation into a System F-like calculus in [Xie+20].

The possibility of defining logical relations by solving fixed-point equations has been known for a long time. This is most natural when the language considered has features directly concerning fixed points, such as (co)inductive types [Alt98; PS98]. In different settings, this technique appears rare, and our work is one example.

Chapter 4

Implementation

We will present an expository implementation of the language, including a rewriting-based evaluator and an evaluator based on the proof in the previous chapter.

One can see logical relations as interpreting the object language into a metalanguage. Indeed, we have not been using the words "interpretation" and "semantic" baselessly. In the previous chapter, our metalanguage was informal mathematics – set theory, we could say. The proof was constructive and it should also be readily formalizable in a dependent type theory. If we care only for the computational content of the proof, a simpler metalanguage will suffice – we no longer have to consider proof terms with complex types but which are operationally unit types. Here, we will use the OCaml language.

4.1 Syntax and rewriting-based evaluator

To be faithful to the presentation in chapter 2, we use a representation with named variables (rather than de Bruijn indices or higher-order abstract syntax). For simplicity, we do not have a separate type for values.

```
let assoc, deassoc = List.assoc, List.remove_assoc

type exp =
  | Var of string
  | Lam of string * exp
  | App of exp * exp
  | Do of exp
  | Handle of exp * handler
  | Lift of exp
and handler = string * string * exp * string * exp

let is_value = function
  | Var _ | Lam _ -> true
  | App _ | Do _ | Handle _ | Lift _ -> false

let rec subst s : exp -> exp = function
  | Var x as e -> (try assoc x s with Not_found -> e)
  | Lam (x,t) -> Lam(x, subst (deassoc x s) t)
  | App (t1,t2) -> App(subst s t1, subst s t2)
  | Lift t -> Lift(subst s t)
  | Do t -> Do(subst s t)
  | Handle (t1,(x,r,t2,y,t3)) ->
    Handle (subst s t1,
            (x, r, subst (deassoc x (deassoc r s)) t2,
             y, subst (deassoc y s) t3))
```

We define evaluation contexts as lists of evaluation context frames (the hole is at the front) and define a function for calculating freeness:

```
type frame =
  | FArg of exp
  | FFun of exp
  | FLift
  | FHandle of handler

type econt = frame list

let frame e : frame -> exp = function
  | FArg e' -> App (e, e')
  | FFun l -> App (l, e)
  | FLift -> Lift e
  | FHandle h -> Handle (e, h)

let plug e k = List.fold_left frame e k

let ffree n = function
  | FArg _ | FFun _ -> n
  | FLift -> n+1
  | FHandle _ -> if n > 0 then n-1 else failwith "nonfree"

let free e = List.fold_left ffree 0 e
```

We implement decomposing terms into evaluation contexts and redexes:

```
type redex =
  | RBeta of string * exp * exp
  | RLift of exp
  | RReturn of exp * handler
  | RDo of econt * exp * handler


let rec find_handler n rev_res = function
  | FHandle h :: fs when n = 0 -> (List.rev rev_res, h, fs)
  | f :: fs -> find_handler (ffree n f) (f :: rev_res) fs
  | [] -> failwith "handler␣not␣found"

let rec decomp k = function
  | App (e1, e2) when is_value e1 && is_value e2 ->
    (match e1 with
    | Lam (x, e1) -> RBeta (x, e1, e2), k
    | _ -> failwith "app␣nonvalue")
  | App (e1, e2) when is_value e1 -> decomp (FFun e1 :: k) e2
  | App (e1, e2) -> decomp (FArg e2 :: k) e1
  | Lift e when is_value e -> RLift e, k
  | Lift e -> decomp (FLift :: k) e
  | Do v -> let r, h, k = find_handler 0 [] k in (RDo (r, v, h)), k
  | Handle (e, h) when is_value e -> RReturn (e, h), k
  | Handle (e, h) -> decomp (FHandle h :: k) e
  | Lam _ | Var _ -> failwith "no␣redex"
```

Finally, we can implement reductions and multi-step reductions:

```
let contract = function
  | RBeta (x, e, v) -> subst [x, v] e
  | RLift v -> v
  | RReturn (v, (_, _, _, x, e)) -> subst [x, v] e
  | RDo (k, v, ((x, r, e, _, _) as h)) ->
    subst [x, v; r, Lam ("z", Handle (plug (Var "z") k, h))] e
```

```
let step e =
  match decomp [] e with
  | r, k -> plug (contract r) k
  | exception _ -> failwith "no␣step"

let rec multistep e =
  match step e with
  | e' -> multistep e'
  | exception _ -> e
```

4.2 Evaluator based on the proof of termination

Now, we demonstrate the computational content of the proof of termination. The code may appear simpler than the code performing multi-step reductions, in part because we leverage the metalanguage's features to realize the object language's features. For instance, functions are realized as OCaml functions and we do not have to consider variable binding ourselves.

First, we define analogs of the interpretations of values and the relation $\mathcal{E}$. We can see that effect types do not play a role in computation.

```
type goodval = Fun of (goodval -> goodexp)
and goodexp =
  | Val of goodval
  | Stk of goodval * int * (goodval -> goodexp)
```

Now, we define functions which closely follow the compatibility lemmas:

```
let eval_lam f = Val (Fun f)

let rec eval_lift = function
  | Val v -> Val v
  | Stk (v, n, k) -> Stk (v, n+1, fun u -> eval_lift (k u))

let rec eval_app e1 e2 =
  match e1 with
  | Stk (v, n, k) -> Stk (v, n, fun u -> eval_app (k u) e2)
  | Val f ->
  match e2 with
  | Stk (v, n, k) -> Stk (v, n, fun u -> eval_app e1 (k u))
  | Val v ->
  match f with Fun f -> f v

let rec eval_handle e eh er =
  match e with
  | Val v -> er v
  | Stk (v, n, k) ->
    if n = 0
    then eh v (Fun (fun u -> eval_handle (k u) eh er))
    else Stk (v, n-1, fun u -> eval_handle (k u) eh er)

let eval_do = function
  | Val v -> Stk (v, 0, fun u -> Val v)
  | _ -> failwith "do␣nonval"
```

We finish by performing something akin to a fold on the syntax tree.

```
let rec eval (e : exp) ctx : goodexp = match e with
  | App (e1, e2) -> eval_app (eval e1 ctx) (eval e2 ctx)
  | Var x -> Val (assoc x ctx)
  | Lam (x, e) -> eval_lam (fun v -> eval e ((x,v) :: ctx))
```

```
| Do v -> eval_do (eval v ctx)
| Handle (e, (x, r, eh, y, er)) ->
  eval_handle
    (eval e ctx)
    (fun v vc -> eval eh ((x,v) :: (r,vc) :: ctx))
    (fun v -> eval er ((y,v) :: ctx))
| Lift e -> eval_lift (eval e ctx)
```

Because OCaml functions are opaque, we cannot observe much using this function. To test that it is adequate, we can add integers to the language. Then, we will have an integer constructor of `goodval`, and we can compare outputs with the rewriting-based evaluator. We omit these straightforward modifications.

Our proof relied on the type system, yet here types (of the object language) are nowhere to be seen. This evaluator is only guaranteed to work if the term is typeable. It is not hard to find a term on which it disagrees with our operational semantics:

$$\mathsf{handle}\ (\mathsf{do}\ v)\ \Omega\ \{x, r.\, x; x.\, x\},$$

where Ω is a nonterminating term, for instance $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. (Such a term is untypeable, as we have proven.) The semantics are left-to-right, so in this program an effect will be performed and the handler will just return the value, ignoring the resumption. Evaluation will never reach the Ω.

Contrarily, the evaluator extracted from the proof will walk this term and try to recursively evaluate subterms. In particular, it will try to evaluate Ω, where it will loop forever.

We leave it for future work to determine if this evaluator can form part of a normalization by evaluation algorithm, i.e., if there exists a function `reify : goodexp -> exp` that can reconstruct a term in normal form given its semantics.

Bibliography

[Alt98] Thorsten Altenkirch. "Logical Relations and Inductive/Coinductive Types". In: *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*. Ed. by Georg Gottlob, Etienne Grandjean, and Katrin Seyr. Vol. 1584. Lecture Notes in Computer Science. Springer, 1998, pp. 343–354. DOI: 10.1007/10703163_23. URL: https://doi.org/10.1007/10703163_23.

[Bie+18] Dariusz Biernacki et al. "Handle with care: relational interpretation of algebraic effects and handlers". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 8:1–8:30. DOI: 10.1145/3158096. URL: https://doi.org/10.1145/3158096.

[For+19] Yannick Forster et al. "On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control". In: *J. Funct. Program.* 29 (2019), e15. DOI: 10.1017/S0956796819000121. URL: https://doi.org/10.1017/S0956796819000121.

[Ham19] Makoto Hamana. "Modular Termination Checking Theorems for Second-Order Computation". In: *CoRR* abs/1912.03434 (2019). arXiv: 1912.03434. URL: http://arxiv.org/abs/1912.03434.

[Hur+12] Chung-Kil Hur et al. "The marriage of bisimulations and Kripke logical relations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 59–72. DOI: 10.1145/2103656.2103666. URL: https://doi.org/10.1145/2103656.2103666.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 145–158. DOI: 10.1145/2500365.2500590. URL: https://doi.org/10.1145/2500365.2500590.

[KS07] Oleg Kiselyov and Chung-chieh Shan. "A Substructural Type System for Delimited Continuations". In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 223–239. DOI: 10.1007/978-3-540-73228-0_17. URL: https://doi.org/10.1007/978-3-540-73228-0_17.

[Lei14] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 100–126. DOI: 10.4204/eptcs.153.8. URL: https://doi.org/10.4204%2Feptcs.153.8.

[Lin07] Sam Lindley. "Extensional Rewriting with Sums". In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 255–271. DOI: 10.1007/978-3-540-73228-0_19. URL: https://doi.org/10.1007/978-3-540-73228-0_19.

[PP13] Gordon Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4 (Dec. 2013). Ed. by Andrzej Tarlecki. DOI: 10.2168/lmcs-9(4:23)2013. URL: https://doi.org/10.2168%2Flmcs-9%284%3A23%292013.

[PPS19] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. "Typed Equivalence of Effect Handlers and Delimited Control". In: *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*. Ed. by Herman Geuvers. Vol. 131. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 30:1–30:16. DOI: 10.4230/LIPIcs.FSCD.2019.30. URL: https://doi.org/10.4230/LIPIcs.FSCD.2019.30.

[PS98] Andrew Pitts and Ian Stark. "Operational Reasoning for Functions with Local State". In: *Higher Order Operational Techniques in Semantics*. Ed. by Andrew Gordon and Andrew Pitts. Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273. URL: http://www.inf.ed.ac.uk/~stark/operfl.html.

[Sko19] Lau Skorstengaard. "An Introduction to Logical Relations". In: *CoRR* abs/1907.11133 (2019). arXiv: 1907.11133. URL: http://arxiv.org/abs/1907.11133.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. USA: Elsevier Science Inc., 2006. ISBN: 0444520775.

[Wad90] Philip Wadler. "Comprehending Monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. Ed. by Gilles Kahn. ACM, 1990, pp. 61–78. DOI: 10.1145/91556.91592. URL: https://doi.org/10.1145/91556.91592.

[Xie+20] Ningning Xie et al. *Effect Handlers, Evidently (Extended Version)*. Tech. rep. MSR-TR-2020-23. Extended version of the ICFP'20 article. Microsoft, July 2020. URL: https://www.microsoft.com/en-us/research/publication/effect-handlers-evidently-extended-version/.