HANDLING ALGEBRAIC EFFECTS

GORDON D. PLOTKIN AND MATIJA PRETNAR

Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Scotland
e-mail address: gdp@inf.ed.ac.uk

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. Algebraic effects are computational effects that can be represented by an equational theory whose operations produce the effects at hand. The free model of this theory induces the expected computational monad for the corresponding effect. Algebraic effects include exceptions, state, nondeterminism, interactive input/output, and time, and their combinations. Exception handling, however, has so far received no algebraic treatment.

We present such a treatment, in which each handler yields a model of the theory for exceptions, and each handling construct yields the homomorphism induced by the universal property of the free model. We further generalise exception handlers to arbitrary algebraic effects. The resulting programming construct includes many previously unrelated examples from both theory and practice, including stream redirection, renaming and hiding in Milner's CCS, timeout, and rollback.

Introduction

In seminal work [13], Moggi proposed a uniform representation of computational effects by monads [1]. For example, working in the category of sets, a computation that returns values from a set A is modelled by an element of TA for a suitable monad T. Examples of such effects include exceptions, state, nondeterminism, interactive input/output, time, continuations, and combinations of them. Later, Plotkin and Power proposed to represent effects by

(1) a set of operations that represent the sources of effects; and
(2) an equational theory that describes their properties [19].

The basic operational intuition is that each computation either returns a value or performs an operation with an outcome that determines a continuation of the computation. The arguments of the operation represent the possible continuations. For example, using a

1998 ACM Subject Classification: D3.3, F3.3.
Key words and phrases: algebraic effects, exception handlers, generalised handlers.
A preliminary version of this work was presented at ESOP 2009, see [21].
This research was supported by EPSRC grant GR/586371/01 and by a Royal Society-Wolfson Award.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.2168/LMCS-???
© Gordon D. Plotkin and Matija Pretnar
Creative Commons

binary choice operation choose, a computation that nondeterministically chooses a boolean is:

$$\mathsf{choose}(\mathsf{return\ true}, \mathsf{return\ false})$$

The outcome of making the choice is binary: either to continue with the computation given by the first argument or else to continue with that given by the second.

A computation that returns values from a set A is modelled by an element of the free model FA, generated by the equational theory. In the case of nondeterminism, these equations state that choose is a semilattice operation. Modulo the forgetful functor, the free model functor is exactly the monad proposed by Moggi to model the corresponding effect [17]. Effects whose monad can be obtained by such an equational presentation are called *algebraic*; the relevant monads are exactly the ranked ones. With the notable exception of continuations [4, 7], all of the above effects are algebraic, and, indeed, have natural equational presentations.

The algebraic view has given ways of combining effects [8] and reasoning about them [20]. However, exception handlers provided a challenge to the algebraic approach.

In more detail, the monad $- + \mathbf{exc}$ for a given set of exceptions $\mathbf{exc}$ is presented by a nullary exception raising operation $\mathsf{raise}_e()$ for each $e \in \mathbf{exc}$ and no equations. The operation $\mathsf{raise}_e()$ takes no arguments as there is no continuation immediately after an exception has been raised.

The question then arises how to deal with exception handling. One approach would be to consider a binary exception handling construct

$$\mathsf{handle}_e(M, N)$$

which proceeds as M unless the exception e is raised, when it proceeds as N. This construct has a standard interpretation as a binary operation using the exception monad; however, as explained in [18], it lacks a certain naturality property characterising equationally specified operations. In programming terms this corresponds to the operation commuting with evaluation contexts $\mathcal{E}[-]$. For example we would expect each of the following two equations to hold:

$$\mathcal{E}[\mathsf{choose}(M, N)] = \mathsf{choose}(\mathcal{E}[M], \mathcal{E}[N]) \qquad \mathcal{E}[\mathsf{raise}_e()] = \mathsf{raise}_e()$$

but not:

$$\mathcal{E}[\mathsf{handle}_e(M, N)] = \mathsf{handle}_e(\mathcal{E}[M], \mathcal{E}[N])$$

Since the naturality property is common to all equationally specified operations, it follows that no alternative (ranked) monad will suffice either.

In this paper we give an algebraic account of exception handling. The main idea is that

(1) handlers correspond to (not necessarily free) models of the equational theory; and
(2) the semantics of handling is given using the universal property of the free model, that it induces unique homomorphisms to these models.

The usual exception handling construct corresponds to the application of the unique homomorphism that preserves returned values. We, however, adopt a more general approach, suggested by Benton and Kennedy [2], and which was an inspiration for the present work. In Benton and Kennedy's approach returned values are passed to a user-defined continuation; this amounts to an application of an arbitrary homomorphism to a computation.

As we shall see, this idea generalises to all algebraic effects, yielding a new programming concept enabling one to handle any algebraic effect. Examples include stream redirection of

shell processes, renaming and hiding in CCS [6], timeout, rollback, and many others. Conceptually, algebraic operations and effect handlers are dual: the former could be called *effect constructors* as they give rise to the effects; the latter could be called *effect deconstructors* as the computations they provide proceed according to the effects already created. Filinski's reflection and reification operations provide general effect constructors and deconstructors in the context of layered monads [3].

In Section 1 we illustrate the main semantic ideas via an informal discussion of exception handlers. Then in Section 2 we give a calculus extending Levy's call-by-push-value [10] with operations, handler definitions, and our effect handling construct, which handles computations using a given handler. In Section 3, we give some examples that demonstrate the versatility of our handlers. Next, in Section 4, we provide a denotational semantics. In Section 5, we sketch some reasoning principles for handlers. In Section 6, we describe the inclusion of recursion; to do this we switch from the category of sets and functions to that of *ω-cpos* (partial orders with suprema of increasing countable chains) and *continuous functions* (monotone functions preserving suprema of increasing countable chains). In the conclusion, we list open questions and briefly discuss some possible answers. At various points in the paper we use operational ideas to aid understanding; we do not however present a formal operational semantics of effect handlers.

1. Exception Handlers

We start our study with exception handlers, both because they are an established concept [2, 11] and also since exceptions are the simplest example of an algebraic effect. To focus on the exposition of ideas, we write this section in a rather informal style, mixing syntax and semantics.

We consider a finite set of exceptions **exc**. Computations returning values from a set A are modelled by elements of the exception monad $TA \stackrel{\text{def}}{=} A + \mathbf{exc}$. This has unit $\eta_A \stackrel{\text{def}}{=} \mathrm{in}_1 \colon A \to A + \mathbf{exc}$, and the computation $\mathsf{return}\, V$ is interpreted by $\eta_A(V) = \mathrm{in}_1(V)$, while $\mathsf{raise}_e()$ is interpreted by $\mathrm{in}_2(e)$.

1.1. Simple handling construct.

Fixing A, the simple, standard, handling construct is

$$M \;\mathsf{handled\ with}\; \{\mathsf{raise}_e \mapsto M_e\}_{e \in \mathbf{exc}}$$

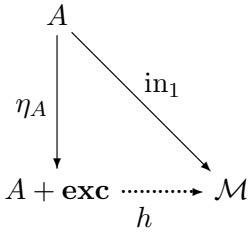
where $\{\cdots\}_{e \in \mathbf{exc}}$ represents a set of computations, one for each exception $e \in \mathbf{exc}$. The construct proceeds by carrying out the computation $M \in A + \mathbf{exc}$, intercepting raised exceptions $e \in \mathbf{exc}$ by carrying out predefined computations $M_e \in A + \mathbf{exc}$ instead. If we choose not to handle a particular exception e, we take M_e to be $\mathsf{raise}_e()$. The handling construct satisfies two equations:

$$\mathsf{return}\, V \;\mathsf{handled\ with}\; \{\mathsf{raise}_e \mapsto M_e\}_{e \in \mathbf{exc}} = \mathrm{in}_1(V)$$
$$\mathsf{raise}_{e'}() \;\mathsf{handled\ with}\; \{\mathsf{raise}_e \mapsto M_e\}_{e \in \mathbf{exc}} = M_{e'}$$

From an algebraic point of view, the computations M_e give a new model $\mathcal{M}$ for the theory of exceptions. The carrier of this model is $A + \mathbf{exc}$ as before; however, for each e, $\mathsf{raise}_e()$ is instead interpreted by M_e. We then see from the above two equations that

$$h(M) \stackrel{\text{def}}{=} M \;\mathsf{handled\ with}\; \{\mathsf{raise}_e \mapsto M_e\}_{e \in \mathbf{exc}}$$

is the unique homomorphism (a map preserving operations) from $A+\mathbf{exc}$ to $\mathcal{M}$ that extends the map $\mathrm{in}_1 \colon A \to A + \mathbf{exc}$, i.e., such that the following diagram commutes:



So the idea is to interpret a handling construct as the application to the computation being handled of a homomorphism obtained using the freeness of the model $A + \mathbf{exc}$ used for computations. This, in turn, requires a model $\mathcal{M}$ on $A + \mathbf{exc}$ for the target of the homomorphism; that algebra is supplied via the handling construct.

1.2. **Extended handling construct.** Benton and Kennedy [2] generalised the handling construct to one of the form

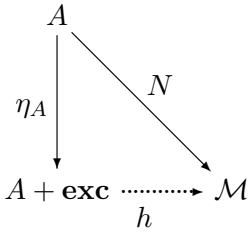
$$M \text{ handled with } \{\mathsf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \text{ to } x : A.\ N(x)$$

(written using our syntax). Here returned values are passed to a user-defined continuation, a map $N \colon A \to B + \mathbf{exc}$, and handling computations N_e return values in B, if they do not themselves raise exceptions: thus $N_e \in B + \mathbf{exc}$. These two facts can be expressed equationally:

$$\text{return } V \text{ handled with } \{\mathsf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \text{ to } x : A.\ N(x) = N(V)$$
$$\mathsf{raise}_{e'}() \text{ handled with } \{\mathsf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \text{ to } x : A.\ N(x) = N_{e'}$$

As discussed in [2], this construct captures a programming idiom that was cumbersome to write with the simpler construct, allows additional program optimisations, and has a stack-free small-step operational semantics.

Algebraically we again have a model $\mathcal{M}$, this time on $B + \mathbf{exc}$, interpreting $\mathsf{raise}_e()$ by N_e. The handling construct can be interpreted as $h(M)$, where $h \colon A + \mathbf{exc} \to \mathcal{M}$ is the unique homomorphism that extends N, i.e., such that the following diagram commutes:



Note that *all* the homomorphisms from the free model to a model on a given carrier are obtained in this way. So Benton and Kennedy's handling construct is the most general one possible from the algebraic point of view.

1.3. **Handling arbitrary algebraic effects.** We can now see how to give handlers for other algebraic effects. A model of an equational theory is given by a set and a set of maps, one for each operation, that satisfy the equations. As before, computations are interpreted in the free model and handling constructs are interpreted by the induced homomorphisms. Where exceptions were replaced by handling computations, operations are replaced by the handling maps; as computations are built from combinations of operations, handling a computation may involve several such replacements.

Importantly however, not all families of maps yield a model of the theory. Even more, for some families, it is undecidable, given their definitions, whether they yield a model or not (see Section 5.2). We can see two general approaches to this difficulty when designing programming languages with facilities to handle effects. One is to make the language designer responsible: the definable families should be restricted so that only models can be defined; this is the approach taken in [21]. Another, the one adopted in this paper, is to allow complete freedom in the language: all possible definitions are permitted. In this case not all handlers will have an interpretation, and it is not the responsibility of the language designer to ensure that all handlers defined give models. One can envisage responsibility being assigned variously to the language designer, to the compiler, to the programmer, to a protocol for establishing program correctness, or, in varying degrees, to all of them. For example, some handlers may be "built-in", and so the responsibility of the language designer, while others may be defined by the programmer, and so their responsibility.

2. Syntax

2.1. **Representing the underlying system.** *Signature types* α, β are given by:

$$\alpha, \beta ::= \mathbf{b} \mid \mathbf{1} \mid \alpha \times \beta \mid \textstyle\sum_{\ell \in L} \alpha_\ell$$

where $\mathbf{b}$ ranges over a given set of *base types*, where ℓ ranges over the set of *labels* (taken from a fixed set Lab of all possible labels), and where L ranges over finite subsets of labels; we do not specify the set of labels in detail, but assume such ones as needed are available. We specify a subset of the base types as *arity base types*, and say that a signature type is an *arity signature type* if the only base types it contains are arity ones.

To represent finite data such as booleans $\mathbf{bool} = \{\mathsf{true}, \mathsf{false}\}$, finite subsets of integers $\mathbf{n} = \{1, 2, \ldots, n\}$, the empty set $\mathbf{0}$, characters $\mathbf{chr}$, or memory locations $\mathbf{loc}$, we may confuse a finite set L with the type $\sum_{\ell \in L} \mathbf{1}$. For infinite sets, such as natural numbers $\mathbf{nat}$ or strings, we take base types as needed.

Next, we assume given a set of typed *function symbols* $\mathsf{f} : \alpha \to \beta$. These represent pure built-in functions, for example arithmetic function symbols $+ : \mathbf{nat} \times \mathbf{nat} \to \mathbf{nat}$, $\ldots$, comparison function symbols $< : \mathbf{nat} \times \mathbf{nat} \to \mathbf{bool}$, $\ldots$, logical function symbols $\neg : \mathbf{bool} \to \mathbf{bool}$, $\ldots$, or an equality on locations $= : \mathbf{loc} \times \mathbf{loc} \to \mathbf{bool}$. We may use infix notation when writing binary functions.

Finally, we assume given a finite set of typed *operation symbols* $\mathsf{op} : \alpha \twoheadrightarrow \beta$, where each such β is an arity signature type, and an *effect theory* $\mathcal{T}$. The operations represent sources of effects and the effect theory determines their properties. In order to to focus on handlers, we postpone the consideration of effect theories to Section 4.1.

The typing $\mathsf{op} : \alpha \twoheadrightarrow \beta$ indicates that the operation represented by op accepts a parameter of type α and, after performing the relevant effect, its outcome, of type β, determines

its continuation; we say that op is *parameterised* on α and has *arity* β, or is β-ary. In case $\alpha = \mathbf{1}$, we may just write $\mathsf{op} : \beta$.

The given sets of base types, typed function symbols, and typed operation symbols constitute a *signature*; the syntax is parameterised by the choice of such a signature. The choice of signature (and theory) that we take evidently depends on the effects we want to represent.

We now give some examples, taken from [8].

Examples 2.1.

Exceptions: We take a single nullary (i.e., $\mathbf{0}$-ary) operation symbol $\mathsf{raise} : \mathbf{exc} \twoheadrightarrow \mathbf{0}$, parameterised on $\mathbf{exc}$, for raising exceptions. Here $\mathbf{exc}$ is a finite set of exceptions; we could instead take it to be a base type if we wanted an infinite set of exceptions. The operation symbol is nullary as there is no continuation after raising an exception: instead the exception has to be handled.

State: We take an arity base type $\mathbf{nat}$ for natural numbers and read and write operation symbols $\mathsf{get} : \mathbf{loc} \twoheadrightarrow \mathbf{nat}$ and $\mathsf{set} : \mathbf{loc} \times \mathbf{nat} \twoheadrightarrow \mathbf{1}$, where $\mathbf{loc}$ is a finite set of *locations*. The idea is that there is a state holding natural numbers in the locations, and get retrieves the number from a given location, while set sets a given location to a given number and returns nothing.

Read-only state: Here we only take an arity base type $\mathbf{nat}$ and a read operation symbol $\mathsf{get} : \mathbf{loc} \twoheadrightarrow \mathbf{nat}$.

(Binary) nondeterminism: We take a binary (i.e., $\mathbf{2}$-ary) operation symbol choose for nondeterministic choice. The operation symbol is binary as the outcome of making a choice is to decide between one of two choices.

Interactive input and output (I/O): We take a finite set $\mathbf{chr}$ of characters, and operation symbols $\mathsf{read} : \mathbf{chr}$, for reading characters, and $\mathsf{write} : \mathbf{chr} \twoheadrightarrow \mathbf{1}$, for writing them. The operation symbol read is $\mathbf{chr}$-ary, as the outcome of reading is to obtain a character; the operation symbol write is unary as there will be just one continuation after writing a character.

2.2. **Types.** Our language follows Levy's call-by-push-value approach [10] and so has a strict separation between *value types* A, B and *computation types* $\underline{C}$. These types are given by:

$$A, B ::= \mathbf{b} \mid \mathbf{1} \mid A \times B \mid \textstyle\sum_{\ell \in L} A_\ell \mid U\underline{C}$$
$$\underline{C} ::= FA \mid \textstyle\prod_{\ell \in L} \underline{C}_\ell \mid A \to \underline{C}$$

The value types extend the signature types, as there is an additional type constructor $U-$. The type $U\underline{C}$ classifies computations of type $\underline{C}$ that have been *thunked* (or frozen) into values; such computations can be passed around and later *forced* back into evaluation.

The computation type FA classifies the computations that return values of type A. The product computation type $\prod_{\ell \in L} \underline{C}_\ell$ classifies finite indexed products of computations, of types $\underline{C}_\ell$, for $\ell \in L$. These tuples are not evaluated sequentially as in a call-by-value setting; instead, a component of a tuple is evaluated only once it is selected by a projection. Finally, the function type $A \to \underline{C}$ classifies computations of type $\underline{C}$ parametric on values of type A.

2.3. **Terms.** The terms of our language consist of *value* terms V, W, *computation* terms M, N, and *handler* terms H. They are given by:

$$\begin{aligned} V, W &::= x \mid \mathsf{f}(V) \mid \langle\rangle \mid \langle V, W\rangle \mid \ell(V) \mid \mathsf{thunk}\ M \\ M, N &::= \mathsf{match}\ V\ \mathsf{with}\ \langle x, y\rangle \mapsto M \mid \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto M_\ell\}_{\ell\in L} \mid \mathsf{force}\ V \mid \\ &\quad \mathsf{return}\ V \mid M\ \mathsf{to}\ x : A.\ N \mid \langle M_\ell\rangle_{\ell\in L} \mid \mathsf{prj}_\ell\ M \mid \lambda x : A.\ M \mid M\ V \mid \\ &\quad \mathsf{op}_V(x : \beta.\ M) \mid k(V) \mid M\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x : A.\ N \\ H &::= \{\mathsf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}:\alpha\twoheadrightarrow\beta} \end{aligned}$$

where $x, y, \ldots$ range over an assumed set of *value variables*, and k ranges over an assumed set of *continuation variables*. Here $\{\cdots\}_{\ell\in L}$ represents a set of computations, one for each label $\ell \in L$; similarly, $\{\cdots\}_{\mathsf{op}:\alpha\twoheadrightarrow\beta}$ represents a set of computations, one for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$.

We may omit type annotations in bindings if it does not cause ambiguity; we may also speak of values, computations, or handlers instead of value terms, computation terms, or handler terms, respectively.

All the syntax is standard from call-by-push-value, other than that for the handlers and the last line of the computation terms. Value terms are built from value variables, the usual constructs for finite products, constructs for indexed sums, and thunked computations. Note that value terms only involve constructors, for example, pairing $\langle V, W\rangle$, while the corresponding destructor terms, for example matching $\mathsf{match}\ V\ \mathsf{with}\ \langle x, y\rangle \mapsto M$, are computations.

Next, there are constructor and destructor computation terms for computation types. Perhaps the most interesting ones are the ones for the type FA. The constructor computation $\mathsf{return}\ V$ returns the value V, while the sequencing construct $M\ \mathsf{to}\ x : A.\ N$ evaluates M, binds the result to x, and proceeds as N.

Next, there is an *operation application* computation term:

$$\mathsf{op}_V(x : \beta.\ M)$$

This first triggers the operation op with parameter V and then binds the outcome to x, proceeding as the *continuation* M.

Examples 2.2.

- The computation
$$\mathsf{read}_{\langle\rangle}(c : \mathbf{chr}.\ \mathsf{write}_c(x : \mathbf{1}.\ \mathsf{write}_c(y : \mathbf{1}.\ \mathsf{return}\ \langle\rangle)))$$
reads a character c, entered by the user, prints it out twice, and returns the unit value.
- The computation
$$\mathsf{get}_\ell(n : \mathbf{nat}.\ \mathsf{set}_{\langle\ell, n+1\rangle}(x : \mathbf{1}.\ \mathsf{return}\ n))$$
increments the number x, stored in location ℓ, and returns the old value.

Next, a handler term

$$\{\mathsf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}:\alpha\twoheadrightarrow\beta}$$

is given by a finite set of *handling operation definitions*

$$\mathsf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathsf{op}}$$

one for each operation symbol op. The *handling terms* M_{op} are dependent on their parameters, captured in their *parameter variables* x, and on the continuations of the handled operations, captured in their continuation variables k. Note that continuation variables never appear independently, only in the form $k(V)$, where they are applied to a value V.

Examples 2.3.

- An exception handler dependent on a variable $f : U(\mathbf{exc} \to \underline{C})$, which determines how to handle exceptions, is written as:
$$H_{\text{exc}} = \{\mathsf{raise}_{e:\mathbf{exc}}(k : \mathbf{0} \to \underline{C}) \mapsto (\mathsf{force}\, f)\, e\}$$
Note that we do not use the continuation k in the handling term. Indeed, raise is a nullary operation symbol and there are no values of type $\mathbf{0}$ we could feed to the continuation, hence we cannot use it.
- Even though we cannot modify read-only state, we can still evaluate a computation with the state temporarily set to a different value. To do so, we use the *temporary-state* handler that is dependent on a variable $n : \mathbf{nat}$; it is
$$H_{\text{temporary}} = \{\mathsf{get}_{\ell:\mathbf{loc}}(k : \mathbf{nat} \to \underline{C}) \mapsto k(n)\}$$

Finally, we have the *handling* computation term

$$M \;\mathsf{handled\ with}\; H \;\mathsf{to}\; x : A.\; N$$

This evaluates the computation M, handling all operation application computations according to H, binds the result to x and proceeds as N. (Sequencing is equivalent to the special case of handling in which H handles all operations by themselves: see the discussion of this point in section 5.1.) In more detail, handling works as follows. Assume that M triggers an operation application $\mathsf{op}_V(y.\, M')$ and that the corresponding handling term is $\mathsf{op}_z(k) \mapsto M_{\mathsf{op}} \in H$. Then, the operation is handled by evaluating M_{op} instead, with the parameter variable z bound to V and with each occurrence of a term of the form $k(W)$ in M_{op} replaced by

$$M'[W/y] \;\mathsf{handled\ with}\; H \;\mathsf{to}\; x : A.\; N$$

Thus, the continuation k receives an outcome W, determined by M_{op}, and is handled in the same way as M. The handling term M_{op} may use the continuation k any number of times and the behaviour of the handling construct can be very involved. Note that while continuations are handled by H, the handling term M_{op} itself is not. Any operations it triggers or values it returns escape the handler. They could however be handled by an enclosing handler.

In the next example we use a standard let binding abbreviation, defined as follows:

$$\mathsf{let}\; x : A \;\mathsf{be}\; V \;\mathsf{in}\; M \stackrel{\text{def}}{=} (\lambda x : A.\; M)\, V$$

Example 2.4. The simplest use for the handling construct is handling exceptions. Using a handler H_{exc} from Example 2.3, we would write the computation

$$M \;\mathsf{handled\ with}\; \{\mathsf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \;\mathsf{to}\; x : A.\; N$$

given in Section 1, as

$$\begin{array}{l}\mathsf{let}\; f : U(\mathbf{exc} \to \underline{C}) \;\mathsf{be\ thunk}\; M' \;\mathsf{in}\\ M \;\mathsf{handled\ with}\; H_{\text{exc}} \;\mathsf{to}\; x : A.\; N\end{array}$$

where $M' : \mathbf{exc} \to \underline{C}$ is given by:

$$\lambda e : \mathbf{exc}.\ \mathsf{match}\ e\ \mathsf{with}\ \{e(x) \mapsto N_e\}_{e \in \mathbf{exc}}$$

In this case, the behaviour matches the one given by Benton and Kennedy. If the computation M returns a value V then $N[V/x]$ is evaluated. If, instead, the computation M triggers an exception e then the replacement term $M'\, e$ is evaluated instead. (So, if a value is returned that value is the final result of the entire computation and is not bound in N.)

Remark 2.5. The syntax of our handling construct differs from that of Benton and Kennedy:

$$\mathsf{try}\ x : A \Leftarrow M\ \mathsf{in}\ N\ \mathsf{unless}\ \{e \Rightarrow M_e\}_{e \in \mathbf{exc}}$$

They noted some programming concerns regarding their syntax [2]. In particular, is not obvious that M is handled but N is not; this is especially the case when N is large and the handler is obscured. An alternative they propose is:

$$\mathsf{try}\ x : A \Leftarrow M\ \mathsf{unless}\ \{e \Rightarrow M_e\}_{e \in \mathbf{exc}}\ \mathsf{in}\ N$$

but then it is not obvious that x is bound in N but not in the handler. The syntax of our construct $M\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x : A.\ N$ addresses both those issues. It also clarifies the order of evaluation: M is handled with H and its results are bound to x and then used in N.

Example 2.6. Consider the temporary state handler $H_{\mathrm{temporary}}$ given in Example 2.3. Then the computation

$$\begin{array}{l}\mathsf{let}\ n : \mathbf{nat}\ \mathsf{be}\ 20\ \mathsf{in}\\ \mathsf{get}_\ell(x : \mathbf{nat}.\ \mathsf{return}\ x + 1)\\ \mathsf{handled\ with}\ H_{\mathrm{temporary}}\ \mathsf{to}\ y : A.\ \mathsf{return}\ y \cdot 2\end{array}$$

involving the stateful computation of Example 2.2 evaluates as follows: first, the get operation is handled, with an outcome 20; then $\mathsf{return}\ 20 + 1$ is handled with result 21; this is substituted for y in $\mathsf{return}\ y \cdot 2$, and the final result is 42.

As remarked in Section 1.3, the language considered in [21] has restricted facilities for defining handlers. In more detail, two levels of language are considered there. In the first there are no handlers, and so no handling constructs. The first level is used to define handlers, which, if they give models, are then used in handling constructs in the second level (which has no facilities for the further definition of handlers). This can be considered a *minimal* approach in contrast to that considered here, which can rather be considered *maximal* as handlers and handling can be nested arbitrarily deeply. The advantage of the maximal approach is that it accommodates all possible ways of treating the problem of ensuring that handlers give models.

2.4. **Typing judgements.** All typing judgements are made in *value contexts*

$$\Gamma = x_1 : A_1, \ldots, x_m : A_m$$

of value variables x_i bound to value types A_i and *continuation contexts*

$$K = k_1 : \alpha_1 \to \underline{C}_1, \ldots, k_n : \alpha_n \to \underline{C}_n$$

of continuation variables k_j bound to continuation types $\alpha_j \to \underline{C}_j$. Here, *continuation types* have the form $\alpha \to \underline{C}$ with α an arity signature type: they type continuations k

that accept a value of type α and proceed as a computation of type $\underline{C}$. Values are typed as $\Gamma \mid K \vdash V : A$, computations are typed as $\Gamma \mid K \vdash M : \underline{C}$, and handlers are typed as $\Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler}$. Values are typed according to the following rules:

$$\Gamma \mid K \vdash x : A \quad (x : A \in \Gamma) \qquad \frac{\Gamma \mid K \vdash V : \alpha}{\Gamma \mid K \vdash \mathsf{f}(V) : \beta} \quad (\mathsf{f} : \alpha \to \beta) \qquad \Gamma \mid K \vdash \langle\rangle : \mathbf{1}$$

$$\frac{\Gamma \mid K \vdash V : A \qquad \Gamma \mid K \vdash W : B}{\Gamma \mid K \vdash \langle V, W \rangle : A \times B} \qquad \frac{\Gamma \mid K \vdash V : A_\ell}{\Gamma \mid K \vdash \ell(V) : \sum_{\ell \in L} A_\ell} \quad (\ell \in L)$$

$$\frac{\Gamma \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathsf{thunk}\, M : U\underline{C}}$$

Next, computations are typed according to the following rules:

$$\frac{\Gamma \mid K \vdash V : A \times B \qquad \Gamma, x : A, y : B \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathsf{match}\ V\ \mathsf{with}\ \langle x, y \rangle \mapsto M : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : \sum_{\ell \in L} A_\ell \qquad \Gamma, x_\ell : A_\ell \mid K \vdash M_\ell : \underline{C} \quad (\ell \in L)}{\Gamma \mid K \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto M_\ell\}_{\ell \in L} : \underline{C}} \qquad \frac{\Gamma \mid K \vdash V : U\underline{C}}{\Gamma \mid K \vdash \mathsf{force}\, V : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : A}{\Gamma \mid K \vdash \mathsf{return}\, V : FA} \qquad \frac{\Gamma \mid K \vdash M : FA \qquad \Gamma, x : A \mid K \vdash N : \underline{C}}{\Gamma \mid K \vdash M\ \mathsf{to}\ x : A.\ N : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash M_\ell : \underline{C}_\ell \quad (\ell \in L)}{\Gamma \mid K \vdash \langle M_\ell \rangle_{\ell \in L} : \prod_{\ell \in L} \underline{C}_\ell} \qquad \frac{\Gamma \mid K \vdash M : \prod_{\ell \in L} \underline{C}_\ell}{\Gamma \mid K \vdash \mathsf{prj}_\ell\, M : \underline{C}_\ell} \quad (\ell \in L)$$

$$\frac{\Gamma, x : A \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \lambda x : A.\ M : A \to \underline{C}} \qquad \frac{\Gamma \mid K \vdash M : A \to \underline{C} \qquad \Gamma \mid K \vdash V : A}{\Gamma \mid K \vdash M\, V : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : \alpha \qquad \Gamma, x : \beta \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathsf{op}_V(x : \beta.\ M) : \underline{C}} \quad (\mathsf{op} : \alpha \twoheadrightarrow \beta) \qquad \frac{\Gamma \mid K \vdash V : \alpha}{\Gamma \mid K \vdash k(V) : \underline{C}} \quad (k : \alpha \to \underline{C} \in K)$$

$$\frac{\Gamma \mid K \vdash M : FA \qquad \Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler} \qquad \Gamma, x : A \mid K \vdash N : \underline{C}}{\Gamma \mid K \vdash M\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x : A.\ N : \underline{C}}$$

Finally, handlers are typed according to the following rule:

$$\frac{\Gamma, x : \alpha \mid K, k : \beta \to \underline{C} \vdash M_{\mathsf{op}} : \underline{C} \quad (\mathsf{op} : \alpha \twoheadrightarrow \beta)}{\Gamma \mid K \vdash \{\mathsf{op}_{x : \alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathsf{op}}\}_{\mathsf{op} : \alpha \twoheadrightarrow \beta} : \underline{C}\ \mathbf{handler}}$$

Observe that K may contain more than one continuation variable when the handler being defined is used in handling definitions of other handlers.

2.5. **Abbreviations.** Before we continue, let us introduce a few abbreviations to help make examples more readable. First, we obtain arbitrary finite products from binary products:

$$A_1 \times \cdots \times A_n \stackrel{\text{def}}{=} (A_1 \times \cdots \times A_{n-1}) \times A_n \quad (n \geq 3)$$

$$\langle V_1, \ldots, V_n \rangle \stackrel{\text{def}}{=} \langle \langle V_1, \ldots, V_{n-1} \rangle, V_n \rangle \quad (n \geq 3)$$

understanding binary product, where $n = 2$, as before, the unit product, where $n = 1$ as simply A_1, and the empty product, where $n = 0$, as $\mathbf{1}$.

The main use of products is to pass around multiple values as one, so we set:

$$\mathsf{f}(V_1, \ldots, V_n) \stackrel{\text{def}}{=} \mathsf{f}(\langle V_1, \ldots, V_n \rangle)$$

$$\ell(V_1, \ldots, V_n) \stackrel{\text{def}}{=} \ell(\langle V_1, \ldots, V_n \rangle)$$

$$\mathsf{op}_{V_1,\ldots,V_n}(x : \beta.\ M) \stackrel{\text{def}}{=} \mathsf{op}_{\langle V_1,\ldots,V_n \rangle}(x : \beta.\ M)$$

$$k(V_1, \ldots, V_n) \stackrel{\text{def}}{=} k(\langle V_1, \ldots, V_n \rangle)$$

Further, where possible, we omit empty parentheses in values and write:

$$\mathsf{f} \stackrel{\text{def}}{=} \mathsf{f}()$$

$$\ell \stackrel{\text{def}}{=} \ell()$$

$$k \stackrel{\text{def}}{=} k()$$

We also adapt the tuple destructor to tuples of arbitrary finite size. Using this destructor, we allow multiple variables in binding constructs such as sequencing or handler definitions. For example, we set

$$\mathsf{op}_{x_1,\ldots,x_n}(k) \mapsto M \stackrel{\text{def}}{=} \mathsf{op}_x(k) \mapsto (\mathsf{match}\ x\ \mathsf{with}\ \langle x_1, \ldots, x_n \rangle \mapsto M)$$

Similarly, we omit empty parentheses in binding constructs that bind no variables. For the set of booleans, we set:

$$\mathsf{if}\ V\ \mathsf{then}\ M\ \mathsf{else}\ N \stackrel{\text{def}}{=} \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{true} \mapsto M, \mathsf{false} \mapsto N\}$$

For operation symbols $\mathsf{op} : \alpha \twoheadrightarrow \mathbf{n}$, we define the usual finitary operation applications by:

$$\mathsf{op}_V(M_1, \ldots, M_n) \stackrel{\text{def}}{=} \mathsf{op}_V(x : \mathbf{n}.\ \mathsf{match}\ x\ \mathsf{with}\ \{i \mapsto M_i\}_{i \in \mathbf{n}})$$

and write handling definitions as:

$$\mathsf{op}_x(k_1, \ldots, k_n) \mapsto M_{\mathsf{op}}$$

where in M_{op}, we write k_i instead of $k(i)$ for $1 \leq i \leq n$. In particular, we have:

$$\mathsf{op}_V() \stackrel{\text{def}}{=} \mathsf{op}_V(x : \mathbf{0}.\ \mathsf{match}\ x\ \mathsf{with}\ \{\})$$

and the handling definitions for nullary operations do not contain the corresponding continuation variable. This agrees with the discussion given in Examples 2.3.

When a handler term contains handling terms only for operation symbols from a subset Θ of the set of operation symbols, we assume that the remaining operations are handled

by themselves (so they are "passed through"). Such a handler is defined by:

$$\{\mathsf{op}_x(k) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}\in\Theta} \stackrel{\text{def}}{=} \left\{ \mathsf{op}_x(k) \mapsto \begin{cases} M_{\mathsf{op}} & (\mathsf{op} \in \Theta) \\ \mathsf{op}_x(y : \beta.\ k(y)) & (\mathsf{op} \notin \Theta) \end{cases} \right\}_{\mathsf{op}}$$

Sometimes we do not wish to write a value continuation in handlers. Then, we use the following abbreviation:

$$M \text{ handled with } H \stackrel{\text{def}}{=} M \text{ handled with } H \text{ to } x : A.\ \mathsf{return}\ x$$

which employs a standard value continuation — the identity one. This abbreviation can be considered as a generalisation to arbitrary algebraic effects of the simple exception handling construct discussed in Section 1.

There is a difference between

$$M \text{ handled with } H \text{ to } x : A.\ N$$

which is the full handling construct,

$$(M \text{ handled with } H) \text{ to } x : A.\ N$$

which takes the result of the handled computation and binds it to x in N, and

$$(M \text{ to } x : A.\ N) \text{ handled with } H$$

which handles the computation that evaluates M and binds the result to x in N. Both the first and the second computation handle only effects triggered by M, while the third computation handles effects triggered by both M and N. Furthermore, in the first computation, x binds the value returned by M, while in the second computation, x binds the value returned by M once handled with H.

Example 2.7. To see the difference between the first two computations and the third, set:

$$H \stackrel{\text{def}}{=} \{\mathsf{raise}_e() \mapsto \mathsf{return}\ 10\}$$
$$M \stackrel{\text{def}}{=} \mathsf{return}\ 5$$
$$N \stackrel{\text{def}}{=} \mathsf{raise}_e()$$

Then the first two computations raise exception e, while the third one returns 10. To see the difference between the second computation and the other two, set:

$$H \stackrel{\text{def}}{=} \{\mathsf{raise}_e() \mapsto \mathsf{return}\ 10\}$$
$$M \stackrel{\text{def}}{=} \mathsf{raise}_e()$$
$$N \stackrel{\text{def}}{=} \mathsf{return}\ 5$$

Then the second computation returns 5, while the other two return 10.

3. Examples

3.1. **Stream redirection.** Let us begin with a practical example, that of processes in a Unix-like operating system. The main philosophy behind these processes is that they read their input and write their output through standard input and output channels. These channels are usually connected to a keyboard and a terminal window. However, an output of one process can be piped to the input of another one, allowing multiple simple processes to be combined into more powerful ones.

For our simplified model we take a base type **file** to represent files and two operation symbols: $\mathsf{read} : \mathbf{channel} \twoheadrightarrow \mathbf{chr}$ and $\mathsf{write} : \mathbf{chr} \times \mathbf{channel} \twoheadrightarrow \mathbf{1}$. Here, **channel** is an abbreviation for the sum $\mathbf{1}_{\mathrm{std}} + \mathbf{file}_{\mathrm{file}}$, where std represents the standard input and output channel, and $\mathrm{file}(f)$ represents the file f.

Our first examples concern using handlers to achieve various redirections of output and input streams. We begin with the redirection `p > out` which takes the output stream of a process `p` and writes it to a file `out`. This is used to either automatically generate files or to log the activity of processes. The redirection is written as:

$$\Gamma \mid K \vdash \mathsf{let}\ f : \mathbf{file}\ \mathsf{be}\ \mathtt{out}\ \mathsf{in}\ \mathtt{p}\ \mathsf{handled\ with}\ H_{\mathtt{>}} : \underline{C}$$

where the handler $H_{\mathtt{>}}$ is given by:

$$\begin{aligned}
\Gamma, f : \mathbf{file} \mid K \vdash \{ & \\
& \mathsf{write}_{c,ch}(k) \mapsto \mathsf{match}\ ch\ \mathsf{with}\ \{ \\
& \quad \mathrm{std} \mapsto \mathsf{write}_{c,\mathrm{file}(f)}(k), \\
& \quad \mathrm{file}(f') \mapsto \mathsf{write}_{c,\mathrm{file}(f')}(k) \\
& \} \\
\} : \underline{C}\ & \mathbf{handler}
\end{aligned}$$

A few special files also exist that allow one to treat different devices as ordinary files. An example is the null device `/dev/null`, which discards everything that is written to it. The command `p > /dev/null` hence effectively suppresses the output of the process `p`. The same behaviour can be achieved using the handler $H_{\mathtt{>\ /dev/null}}$, given by:

$$\begin{aligned}
\Gamma \mid K \vdash \{ & \\
& \mathsf{write}_{c,ch}(k) \mapsto \mathsf{match}\ ch\ \mathsf{with}\ \{ \\
& \quad \mathrm{std} \mapsto k, \\
& \quad \mathrm{file}(f') \mapsto \mathsf{write}_{c,\mathrm{file}(f')}(k) \\
& \} \\
\} : \underline{C}\ & \mathbf{handler}
\end{aligned}$$

A similar redirection `p < in` reads the file `in` and passes its contents to the process `p`. This redirection can be represented using a handler that now replaces the standard input

with a given file in all read operations. It is given by:

$$\begin{aligned}\Gamma, f : \mathbf{file} \mid K \vdash \{ & \\ & \mathsf{read}_{ch}(k) \mapsto \mathsf{match}\ ch\ \mathsf{with}\ \{ \\ & \quad \mathrm{std} \mapsto \mathsf{read}_{\mathrm{file}(f)}(c : \mathbf{chr}.\ k(c)), \\ & \quad \mathrm{file}(f') \mapsto \mathsf{read}_{\mathrm{file}(f')}(c : \mathbf{chr}.\ k(c))) \\ & \} \\ \} : \underline{C}\ & \mathbf{handler}\end{aligned}$$

Both redirections can be combined so `(p < in) > out` reads the input file and writes the processed contents to the output file.

We next consider UNIX pipes `p1 | p2`, where the output of `p1` is fed to the input of `p2`. Using handlers we can express simple cases of the pipe combinator `|`. For example, consider the pipe `yes | p`, where the process `yes` outputs an infinite stream made of a predetermined character (the default one being `y`). Such a pipe then gives a way of routinely confirming a series of actions, for example deleting a large number of files. (This is not always the best way, since processes usually provide a safer means of doing the same thing, but is often useful when they do not.) This pipe may be written using the following handler:

$$\Gamma, c : \mathbf{chr} \mid K \vdash \{\mathsf{read}_{\mathrm{std}}(k) \mapsto k(c)\} : \underline{C}\ \mathbf{handler}$$

We do however, not know how to use a handler to represent `p1 | p2` in general. The difficulty is that the pipe combinator is a *binary* deconstructor in that it reacts to actions of *both* its arguments. This contrasts with the above examples of redirection and the simple case of the pipe combinator, which are all *unary* deconstructors.

3.2. **Explicit Nondeterminism.** The evaluation of a nondeterministic computation usually takes only one of all the possible paths. An alternative is to take all the paths in some order and allow the possibility of a path's failing. This kind of nondeterminism is represented slightly differently from binary nondeterminism. In addition to the binary operation symbol choose, we take a nullary operation symbol fail, representing a path that failed. This interpretation of nondeterminism corresponds to Haskell's nondeterminism monad [14].

We consider a handler which extracts the results of a computation into a list (which can then be operated on by other computations). Since our calculus has no polymorphic lists — although they could easily be added — we limit ourselves to lists of a single base type $\mathbf{b}$. We take a base type $\mathbf{list_b}$ and the appropriate function symbols: $\mathsf{nil} : \mathbf{1} \to \mathbf{list_b}$, $\mathsf{cons} : \mathbf{b} \times \mathbf{list_b} \to \mathbf{list_b}$, $\mathsf{append} : \mathbf{list_b} \times \mathbf{list_b} \to \mathbf{list_b}$, and others.

Then, all the results of a computation $\Gamma \mid K \vdash M : F\mathbf{b}$ can be extracted into a returned value of type $F\mathbf{list_b}$ by:

$$\Gamma \mid K \vdash M\ \mathsf{handled\ with}\ H_{\mathrm{list}}\ \mathsf{to}\ x : \mathbf{b}.\ \mathsf{return}\ \mathsf{cons}(x, \mathsf{nil}) : F\mathbf{list_b}$$

where H_{list} is the handler given by:

$$\begin{aligned}\Gamma \mid K \vdash \{ & \\ & \mathsf{fail}() \mapsto \mathsf{return}\ \mathsf{nil}, \\ & \mathsf{choose}(k_1, k_2) \mapsto k_1\ \mathsf{to}\ \ell_1 : \mathbf{list_b}.\ k_2\ \mathsf{to}\ \ell_2 : \mathbf{list_b}.\ \mathsf{return}\ \mathsf{append}(\ell_1, \ell_2) \\ \} : F\mathbf{list_b}\ & \mathbf{handler}\end{aligned}$$

3.3. **CCS.** To represent (the finitary part of) Milner's CCS [12] we take a type **act** of actions, together with equality, and three operation symbols representing combinators which we consider as effect constructors: deadlock $\mathsf{nil} : \mathbf{1} \twoheadrightarrow \mathbf{0}$, action prefix $\mathsf{prefix} : \mathbf{act} \twoheadrightarrow \mathbf{1}$, and sum $\mathsf{choose} : \mathbf{1} \twoheadrightarrow \mathbf{2}$. We use the usual notation and write P instead of M for processes, $a.P$ instead of $\mathsf{prefix}_a(P)$, and $P_1 + P_2$ instead of $\mathsf{choose}(P_1, P_2)$. Processes P do not terminate normally, only in deadlock, hence we represent them as computations $P : F\mathbf{0}$.

We consider the other CCS combinators as effect deconstructors. Both renaming and hiding can be represented using handlers. Renaming $P[b/a]$ replaces all a actions in P by b actions; it can be represented using the following handler:

$$\Gamma, a : \mathbf{act}, b : \mathbf{act} \mid K \vdash \{a'.k \mapsto \mathsf{if}\ a' = a\ \mathsf{then}\ b.k\ \mathsf{else}\ a'.k\} : F\mathbf{0}\ \mathbf{handler}$$

Hiding $P \backslash a$ removes all a actions from P; it can be represented using the following handler:

$$\Gamma, a : \mathbf{act} \mid K \vdash \{a'.k \mapsto \mathsf{if}\ a' = a\ \mathsf{then}\ k\ \mathsf{else}\ a'.k\} : F\mathbf{0}\ \mathbf{handler}$$

One can deal with more general versions involving hiding sets of actions similarly.

The final CCS combinator — parallel, written $P \mid Q$ — is also an effect deconstructor. However, like the UNIX pipe combinator, it is a binary one and, for the same reasons as in that case, we do not know how to represent it using handlers. (Note that renaming and hiding are unary effect deconstructors.)

3.4. **Parameter-passing handlers.** We sometimes wish to handle different instances of the same operation differently. For example, we may wish to suppress the output after a certain number of characters have been printed out. Although each handler prescribes a fixed handling term for each operation, we can use handlers on function types $P \to \underline{C}$ to obtain $\underline{C}$ handlers that pass around parameters of value type P. Each handling term then has type $P \to \underline{C}$, rather than $\underline{C}$, and captured continuations k have type $\alpha \to (P \to \underline{C})$, rather than $\alpha \to \underline{C}$.

For any computation type $\underline{C}$, we define a handler H_{suppress}, which behaves as above, by:

$$\begin{aligned}\Gamma, n_{\max} : \mathbf{nat} \mid K \vdash \{ & \\ & \mathsf{write}_c(k : \mathbf{nat} \to \underline{C}) \mapsto \lambda n : \mathbf{nat}. \\ & \quad \mathsf{if}\ n < n_{\max}\ \mathsf{then}\ \mathsf{write}_c(k\,(n+1))\ \mathsf{else}\ k\,n_{\max} \\ & \} : \mathbf{nat} \to \underline{C}\ \mathbf{handler}\end{aligned}$$

The handling term for write is dependent on $n : \mathbf{nat}$, the number of character printed out. If this number is less than $n_{\max}$, the maximum number of characters we want to print out, we write out the character and then handle the continuation, but now passing an incremented parameter to it. But if $n \geq n_{\max}$, we do not perform the write operation, but continue with the handled continuation. It does not matter exactly which parameter we pass to it, as long as it at least $n_{\max}$. Still, we call the continuation as it may return a value or trigger other operations.

Since the type of the handler is not of the form $FA\,\mathbf{handler}$, we need to specify a term for "handling" values in the handling construct, also dependent on the current value of the parameter. For example, if we wish to handle $M : FA$ with H_{suppress}, we write

$$\Gamma \mid K \vdash M\ \mathsf{handled\ with}\ H_{\text{suppress}}\ \mathsf{to}\ x : A.\ \lambda n : \mathbf{nat}.\ \mathsf{return}\ x : \mathbf{nat} \to FA$$

This means than no matter what the value of parameter n is, we return the value x. The handled computation has type $\mathbf{nat} \to FA$ and so, in order to obtain a computation of type FA, we need to apply it to the initial parameter $0 : \mathbf{nat}$ as:

$$\Gamma \mid K \vdash (M \text{ handled with } H_{\text{suppress}} \text{ to } x : A.\ \lambda n : \mathbf{nat}.\ \text{return } x)\, 0 : FA$$

Remark 3.1. In the presence of parameters, the convention of handling omitted operations with themselves is still valid. What one wishes to do in the case of an operation that is not handled is to pass an unchanged parameter to the continuation, that is:

$$\mathsf{op}_y(k) \mapsto \lambda p : P.\ \mathsf{op}_y(x : \beta.\ k(x)\, p)$$

Since operations are defined pointwise on the function type, this is equivalent to

$$\mathsf{op}_y(k) \mapsto \mathsf{op}_y(x : \beta.\ \lambda p : P.\ k(x)\, p)$$

which, by η-equivalence, is equivalent to

$$\mathsf{op}_y(k) \mapsto \mathsf{op}_y(x : \beta.\ k(x))$$

and is exactly what our convention assumes.

Although we do not give an equational logic in this paper, the two equivalences we have used here are standard in call-by-push-value and are discussed further in Section 5.

3.5. **Timeout.** An example of a parameter-passing handler is the *timeout* handler. This runs a computation and waits for a given amount of time. If the computation does not complete in given time, it terminates it, returning a default value x_0 instead.

We represent time using a single operation $\mathsf{delay} : \mathbf{nat} \twoheadrightarrow \mathbf{1}$, where $\mathsf{delay}_t(M)$ is a computation that stalls for t units of time, and then proceeds as M. For any value type A, the timeout handler H_{timeout} is given by:

$$\begin{aligned}
\Gamma, x_0 : A, t_{\text{wait}} : \mathbf{nat} \mid K \vdash \{ & \\
& \mathsf{delay}_t(k : \mathbf{nat} \to FA) \mapsto \lambda t_{\text{spent}} : \mathbf{nat}. \\
& \quad \mathsf{if}\ t + t_{\text{spent}} \leq t_{\text{wait}} \\
& \quad \mathsf{then}\ \mathsf{delay}_t(k\,(t + t_{\text{spent}})) \\
& \quad \mathsf{else}\ \mathsf{delay}_{t_{\text{wait}} - t_{\text{spent}}}(\mathsf{return}\ x_0) \\
& \} : \mathbf{nat} \to FA\ \mathbf{handler}
\end{aligned}$$

and is used on a computation $M : FA$ as:

$$\Gamma, x_0 : A, t_{\text{wait}} : \mathbf{nat} \mid K \vdash (M \text{ handled with } H_{\text{timeout}} \text{ to } x : A.\ \lambda t : \mathbf{nat}.\ \text{return } x)\, 0 : FA$$

Note that the handling term preserves the time spent during the evaluation of the handled computation.

3.6. **Rollback.** When a computation raises an exception while modifying the memory, for example, when a connection drops halfway through a database transaction, we may want to revert all modifications made during the computation. This behaviour is termed *rollback*.

Assuming, for the sake of simplicity, that there is only a single location ℓ, an appropriate rollback handler H_{rollback} is given by:

$$\Gamma, n_{\text{init}} : \mathbf{nat} \mid K \vdash \{\mathsf{raise}_e(k) \mapsto \mathsf{set}_{\ell, n_{\text{init}}}(M'\, e)\} : \underline{C}\ \mathbf{handler}$$

where $M' : \mathbf{exc} \to \underline{C}$. To evaluate a computation M, rolling back to the initial state n_{init} in the case of exceptions, we write:

$$\Gamma \mid K \vdash \mathsf{get}(n_{\text{init}} : \mathbf{nat}.\ M\ \mathsf{handled\ with}\ H_{\text{rollback}}) : \underline{C}$$

An alternative is a parameter-passing handler, which does not modify the memory, but keeps track of all the changes to the location ℓ in the parameter n. Then, once the handled computation has returned a value, meaning that no exceptions have been raised, the parameter is committed to the memory. This handler $H_{\text{param-rollback}}$ is:

$$\begin{aligned}\Gamma \mid K \vdash \{ & \\ & \mathsf{get}(k : \mathbf{nat} \to (\mathbf{nat} \to \underline{C})) \mapsto \lambda n : \mathbf{nat}.\ k(n)\, n \\ & \mathsf{set}_{n'}(k : \mathbf{1} \to (\mathbf{nat} \to \underline{C})) \mapsto \lambda n : \mathbf{nat}.\ k(\langle\rangle)\, n' \\ & \mathsf{raise}_e() \mapsto \lambda n : \mathbf{nat}.\ M'\, e \\ \} & : \mathbf{nat} \to \underline{C}\ \mathbf{handler}\end{aligned}$$

It is used on a computation M as follows:

$$\mathsf{get}(n_{\text{init}} : \mathbf{nat}.\ (M\ \mathsf{handled\ with}\ H_{\text{param-rollback}}\ \mathsf{to}\ x : A.\ \lambda n : \mathbf{nat}.\ \mathsf{set}_n(\mathsf{return}\ x))\, n_{\text{init}})$$

Here the initial state is read, obtaining n_{init}, which is passed to the handler as the initial parameter value. If no exception is raised and a value x is returned, the state is updated to reflect the final value n of the parameter.

4. Semantics

To give denotational semantics, we roughly need a cartesian closed category with finite products and coproducts, just like for any language with first-class functions, finite products, and finite sums. More specific to algebraic effects, we need the free model construction and parametric lifting of maps to homomorphisms from the free model; these are used to model both the sequencing and the handling construct. For the sake of simplicity, we largely limit ourselves to sets, but show in Section 6 how everything adapts straightforwardly to ω-cpos. However it may be possible to work over any category $\mathbf{V}$ that is locally countably presentable as a cartesian closed category; presumably one would use a more abstract notion of the effect theories of Section 4.1 below, based on Lawvere $\mathbf{V}$-theories (see [19]).

4.1. **Effect theories.** We describe properties of effects with equations between *templates* T. These describe the general shape of all computations, regardless of their type. Assuming a given signature, templates are given by:

$$T ::= z(V) \mid \mathsf{match}\ V\ \mathsf{with}\ \langle x, y\rangle \mapsto T \mid \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell\}_{\ell \in L} \mid \mathsf{op}_V(x : \beta.\ T)$$

where z ranges over a given set of *template variables*.

In templates, we limit ourselves to *signature values*. These are values that can be typed as $\Gamma \vdash V : \alpha$, where Γ is a context of value variables bound to signature types; the typing rules for this judgement are, with the omission of continuation contexts and the rule for typing thunks, the same as those for values.

We build templates in a context of value variables, bound to signature types, and a *template context*

$$Z = z_1 : \alpha_1, \ldots, z_n : \alpha_n$$

of template variables z_j, bound to arity signature types α_j. Note: $z_j : \alpha_j$ does not represent a value of type α_j, but a computation dependent on such a value. As templates describe common properties of computations, we do not assign them a type, we only check whether they are well-formed, relative to a value context and a template context. The typing judgement for being a well-formed template is $\Gamma \mid Z \vdash T$; it is given by the following rules:

$$\frac{\Gamma \vdash V : \alpha}{\Gamma \mid Z \vdash z(V)} \quad (z : \alpha \in Z) \qquad \frac{\Gamma \vdash V : \alpha \times \beta \qquad \Gamma, x : \alpha, y : \beta \mid Z \vdash T}{\Gamma \mid Z \vdash \mathsf{match}\ V\ \mathsf{with}\ \langle x, y \rangle \mapsto T}$$

$$\frac{\Gamma \vdash V : \sum_{\ell \in L} \alpha_\ell \qquad \Gamma, x_\ell : \alpha_\ell \mid Z \vdash T_\ell \quad (\ell \in L)}{\Gamma \mid Z \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell\}_{\ell \in L}}$$

$$\frac{\Gamma \vdash V : \alpha \qquad \Gamma, x : \beta \mid Z \vdash T}{\Gamma \mid Z \vdash \mathsf{op}_V(x : \beta.\, T)} \quad (\mathsf{op} : \alpha \rightarrowtriangle \beta)$$

An *effect theory* $\mathcal{T}$ is a finite set of equations $\Gamma \mid Z \vdash T_1 = T_2$, where T_1 and T_2 are well-formed relative to Γ and Z.

Some examples follow, continuing on from the corresponding signatures given in Example 2.1 above. We make use of abbreviations for handling tuples, etc., in templates that are analogous to those introduced above for the other kinds of term.

Examples 4.1.

Exceptions: The effect theory is the empty set, as exceptions satisfy no equations — even more, as raise is nullary (its outcome type is $\mathbf{0}$), there are no nontrivial equations we can write.

State: The effect theory consists of the following equations (for readability, we write this and other theories without contexts):

$$\begin{aligned}
\mathsf{get}_\ell(x.\, \mathsf{set}_{\ell,x}(z)) &= z \\
\mathsf{set}_{\ell,x}(\mathsf{set}_{\ell,x'}(z)) &= \mathsf{set}_{\ell,x'}(z) \\
\mathsf{set}_{\ell,x}(\mathsf{get}_\ell(x'.\, z(x')) &= \mathsf{set}_{\ell,x}(z(x)) \\
\mathsf{get}_\ell(x.\, \mathsf{get}_\ell(x'.\, z(x, x'))) &= \mathsf{get}_\ell(x.\, z(x, x)) \\
\mathsf{set}_{\ell,x}(\mathsf{set}_{\ell',x'}(z)) &= \mathsf{set}_{\ell',x'}(\mathsf{set}_{\ell,x}(z)) && (\ell \neq \ell') \\
\mathsf{set}_{\ell,x}(\mathsf{get}_{\ell'}(x'.\, z(x')) &= \mathsf{get}_{\ell'}(x'.\, \mathsf{set}_{\ell,x}(z(x'))) && (\ell \neq \ell') \\
\mathsf{get}_\ell(x.\, \mathsf{get}_{\ell'}(x'.\, z(x, x'))) &= \mathsf{get}_{\ell'}(x'.\, \mathsf{get}_\ell(x.\, z(x, x')) && (\ell \neq \ell')
\end{aligned}$$

We find it convenient to write the last three equations with a side condition $\ell \neq \ell'$. This still remains in the scope of our definition of an effect theory, as inequality is a function symbol and we regard $T_1 = T_2$ $(\ell \neq \ell')$ as an abbreviation for

$$T_1 = (\mathsf{if}\ \ell \neq \ell'\ \mathsf{then}\ T_2\ \mathsf{else}\ T_1)$$

Read-only state: The effect theory consists of the following equations:

$$\begin{aligned}
\mathsf{get}_\ell(x.\, z) &= z \\
\mathsf{get}_\ell(x.\, \mathsf{get}_\ell(x'.\, z(x,x'))) &= \mathsf{get}_\ell(x.\, z(x,x)) \\
\mathsf{get}_\ell(x.\, \mathsf{get}_{\ell'}(x'.\, z(x,x'))) &= \mathsf{get}_{\ell'}(x'.\, \mathsf{get}_\ell(x.\, z(x,x')) \qquad (\ell \neq \ell')
\end{aligned}$$

Nondeterminism: The effect theory consists of the following equations:

$$\begin{aligned}
\mathsf{choose}(z, z) &= z \\
\mathsf{choose}(z_1, z_2) &= \mathsf{choose}(z_2, z_1) \\
\mathsf{choose}(\mathsf{choose}(z_1, z_2), z_3) &= \mathsf{choose}(z_1, \mathsf{choose}(z_2, z_3))
\end{aligned}$$

I/O: The effect theory is (again) the empty set.

It is often convenient to give theories as combinations of other, simpler ones. There are two main ways to do so: *sum* and *tensor* [8]. Suppose we are given two theories over two signatures. The signature of their sum is obtained by taking the union of each of their sets of base types and arity base types, their function symbols and typings, and their operation symbols and typings (where we assume that the two sets of operation symbols are disjoint). The theory of the sum is then simply the union of the two theories. For example, the standard combination of exceptions and state is their sum.

The tensor product represents a commuting combination of two effects, for example, of two independent states. Given two theories over two signatures (again assuming that their two sets of operation symbols are disjoint), their tensor product has the same signature as their sum and its theory is the union of the two theories, together with an equation

$$y : \alpha, y' : \alpha' \mid z : \beta \times \beta' \vdash \mathsf{op}_y(x : \beta.\, \mathsf{op}'_{y'}(x' : \beta'.\, z(x,x'))) = \mathsf{op}'_{y'}(x' : \beta'.\, \mathsf{op}_y(x : \beta.\, z(x,x')))$$

for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$ from the first theory and operation symbol $\mathsf{op}' : \alpha' \twoheadrightarrow \beta'$ from the second.

Examples 4.2.

Explicit nondeterminism: The effect theory consists of following equations:

$$\begin{aligned}
\mathsf{choose}(\mathsf{choose}(z_1, z_2), z_3) &= \mathsf{choose}(z_1, \mathsf{choose}(z_2, z_3)) \\
\mathsf{choose}(z, \mathsf{nil}()) &= z \\
\mathsf{choose}(\mathsf{nil}(), z) &= z
\end{aligned}$$

CCS: The effect theory is the union of the effect theories for nondeterminism and explicit nondeterminism.

Time: The effect theory consists of the following equations:

$$\begin{aligned}
\mathsf{delay}_0(z) &= z \\
\mathsf{delay}_{t_1}(\mathsf{delay}_{t_2}(z)) &= \mathsf{delay}_{t_1+t_2}(z)
\end{aligned}$$

Destructive exceptions: The effect theory (discussed as that of "rollback" in [8]) is the tensor product of the theories for state and exceptions. Hence, it consists of all the equations for state and the following two equations:

$$\begin{aligned}
\mathsf{get}_\ell(x.\, \mathsf{raise}_e()) &= \mathsf{raise}_e() \\
\mathsf{set}_{\ell,x}(\mathsf{raise}_e()) &= \mathsf{raise}_e()
\end{aligned}$$

As raise is nullary, we have written $\mathsf{raise}_e()$ instead of $\mathsf{raise}_e(x : \mathbf{0}.\, T)$, emphasising that the continuation never gets evaluated. The equations imply that a memory operation followed by raising an exception is the same as just raising the exception. Effectively, this implies that all memory operations are moot if an exception occurs, hence the term destructive exceptions. Observe that the first equation is an instance of one occurring in the effect theory for state.

4.2. **Interpreting effect theories.** We assume given an effect theory $\mathcal{T}$. We begin by interpreting each signature type α by a set $[\![\alpha]\!]$.

Suppose we are given an assignment $[\![\mathbf{b}]\!]$ of a set to each base type $\mathbf{b}$, which is countable in the case that it is an arity base type. Finite products and sums of signature types are then interpreted using the corresponding set operations, and base types are interpreted by the assigned sets. Note that $[\![\beta]\!]$ is then a countable set for any arity signature type β; this will allow us to interpret effects in terms of countable equational theories.

An *interpretation* of $\mathcal{T}$ then consists of such an assignment together with a map $[\![\mathsf{f}]\!] \colon [\![\alpha]\!] \to [\![\beta]\!]$ for each function symbol $\mathsf{f} : \alpha \to \beta$. We assume given such an interpretation. We can then interpret each well-typed signature value $\Gamma \vdash V : \alpha$ in an evident way by a map $[\![V]\!] \colon [\![\Gamma]\!] \to [\![\alpha]\!]$, where value contexts Γ are interpreted componentwise by setting $[\![x_1 : \alpha_1, \ldots, x_m : \alpha_m]\!] = [\![\alpha_1]\!] \times \cdots \times [\![\alpha_m]\!]$.

Take a set $|\mathcal{M}|$ and an *operation*

$$\mathsf{op}_{\mathcal{M}} \colon [\![\alpha]\!] \times |\mathcal{M}|^{[\![\beta]\!]} \to |\mathcal{M}|$$

for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$. Then, we can interpret a well-formed template $\Gamma \mid Z \vdash T$ by a map $[\![T]\!] \colon [\![\Gamma]\!] \times [\![Z]\!] \to |\mathcal{M}|$, where template contexts Z are interpreted componentwise by $[\![z_1 : \alpha_1, \ldots, z_n : \alpha_n]\!] = |\mathcal{M}|^{[\![\alpha_1]\!]} \times \cdots \times |\mathcal{M}|^{[\![\alpha_n]\!]}$.

We can then interpret templates:

$$\begin{aligned}
[\![\Gamma \mid Z \vdash z_j(V)]\!] &= \mathrm{ev} \circ \langle \mathrm{pr}_j \circ \mathrm{pr}_2, [\![V]\!] \circ \mathrm{pr}_1 \rangle \quad (1 \leq j \leq n) \\
[\![\Gamma \mid Z \vdash \mathsf{match}\ V\ \mathsf{with}\ \langle x, y \rangle \mapsto T]\!] &= [\![T]\!] \circ (\langle \mathrm{id}_{[\![\Gamma]\!]}, [\![V]\!] \rangle \times \mathrm{id}_{[\![Z]\!]}) \\
[\![\Gamma \mid Z \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell\}_{\ell \in L}]\!] &= [[\![T_\ell]\!]]_{\ell \in L} \circ \mathrm{dist} \circ (\langle \mathrm{id}_{[\![\Gamma]\!]}, [\![V]\!] \rangle \times \mathrm{id}_{[\![Z]\!]}) \\
[\![\Gamma \mid Z \vdash \mathsf{op}_V(x : \beta.\, T)]\!] &= \mathsf{op}_{\mathcal{M}} \circ \langle [\![V]\!] \circ \mathrm{pr}_1, \mathrm{tr}([\![\Gamma, x : \beta \mid Z \vdash T]\!]) \rangle
\end{aligned}$$

where on the right, we have abbreviated the obvious typing judgements.

In the above, we use the following maps: $\mathrm{pr}_i \colon A_1 \times \cdots \times A_n \to A_i$ is the i-th projection; $\mathrm{ev} \colon B^A \times A \to A$ is evaluation; $\mathrm{id}_A \colon A \to A$ is the identity; $\langle f_1, f_2 \rangle \colon A \to B_1 \times B_2$ is the tuple of $f_1 \colon A \to B_1$ and $f_2 \colon A \to B_2$; then $f_1 \times f_2 = \langle f_1 \circ \mathrm{pr}_1, f_2 \circ \mathrm{pr}_2 \rangle \colon A_1 \times A_2 \to B_1 \times B_2$ is the product of $f_1 \colon A_1 \to B_1$ and $f_2 \colon A_2 \to B_2$; $\mathrm{dist} \colon B \times (\sum_{i \in I} A_i) \to \sum_{i \in I} B \times A_i$ is the distributivity isomorphism; $[f_i]_{i \in I} \colon \sum_{i \in I} A_i \to B$ is the co-tuple of a family of functions $f_i \colon A_i \to B$, $i \in I$; and $\mathrm{tr}(f) \colon A \to C^B$ is the transpose of $f \colon A \times B \to C$. For the sake of readability, we have also omitted associativity and commutativity isomorphisms (and we do so again below).

Definition 4.3. A *model* $\mathcal{M}$ of the effect theory $\mathcal{T}$ is a set $|\mathcal{M}|$, called the *carrier* of the model, together with an *operation*

$$\mathsf{op}_{\mathcal{M}} \colon [\![\alpha]\!] \times |\mathcal{M}|^{[\![\beta]\!]} \to |\mathcal{M}|$$

for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$, such that $[\![T_1]\!] = [\![T_2]\!]$ holds for all the equations $\Gamma \mid Z \vdash T_1 = T_2$ in $\mathcal{T}$.

A *homomorphism* $h\colon \mathcal{M}_1 \multimap \mathcal{M}_2$ is a map $h\colon |\mathcal{M}_1| \to |\mathcal{M}_2|$ such that

$$h \circ \mathsf{op}_{\mathcal{M}_1} = \mathsf{op}_{\mathcal{M}_2} \circ (\mathrm{id}_{[\![\alpha]\!]} \times h^{[\![\beta]\!]})$$

holds for all operation symbols $\mathsf{op} : \alpha \twoheadrightarrow \beta$.

The models of $\mathcal{T}$ and the homomorphisms between them form a category $\mathbf{Mod}_{\mathcal{T}}$. This category is equipped with a forgetful functor $U\colon \mathbf{Mod}_{\mathcal{T}} \to \mathbf{Set}$, where $U\mathcal{M} \stackrel{\text{def}}{=} |\mathcal{M}|$ and $Uh \stackrel{\text{def}}{=} h$.

Proposition 4.4. *The functor $U\colon \mathbf{Mod}_{\mathcal{T}} \to \mathbf{Set}$ has a left adjoint $F\colon \mathbf{Set} \to \mathbf{Mod}_{\mathcal{T}}$.*

Proof. The existence of the adjoint functor follows from the corresponding result in the context of countable equational theories [5].

First we obtain a countable equational theory $\mathcal{T}_\omega$ from $\mathcal{T}$. The signature of $\mathcal{T}_\omega$ consists of operations op_a of arity $|[\![\beta]\!]|$ for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$ and each $a \in [\![\alpha]\!]$. As β is an arity signature type, $[\![\beta]\!]$ is a countable set, hence each operation has a countable arity.

Next, for each well-formed template

$$\Gamma \mid z_1 : \alpha_1, \ldots, z_n : \alpha_n \vdash T$$

and each $c \in [\![\Gamma]\!]$, we get a $\mathcal{T}_\omega$ term $Z' : T^c$, where the context Z' consists of variables z_i^a for each $1 \le i \le n$ and each $a \in [\![\alpha_i]\!]$. Since all the α_i are again arity signature types, the sets $[\![\alpha_i]\!]$ are countable, and so Z has countably many variables. The term T^c is defined recursively by:

$$(z_i(V))^c = z_i^{[\![V]\!](c)}$$

$$(\mathsf{match}\ V\ \mathsf{with}\ \langle x, y\rangle \mapsto T)^c = T^{\langle c, \mathrm{pr}_1([\![V]\!](c)), \mathrm{pr}_2([\![V]\!](c))\rangle}$$

$$(\mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell\}_{\ell \in L})^c = T_{\ell'}^{\langle c, a\rangle} \quad ([\![V]\!](c) = \mathrm{in}_{\ell'}(a))$$

$$\mathsf{op}_V(x : \beta.\, T)^c = \mathsf{op}_{[\![V]\!](c)}(T^b)_{b \in [\![\beta]\!]}$$

We then take $\mathcal{T}_\omega$ to be the equational theory generated by all equations $Z' \vdash T_1^c = T_2^c$, where $\Gamma \mid Z \vdash T_1 = T_2$ is a $\mathcal{T}$ equation and $c \in [\![\Gamma]\!]$. It is not difficult to see that the category of models of $\mathcal{T}_\omega$ is equivalent to $\mathbf{Mod}_{\mathcal{T}}$, with the equivalence being consistent with the forgetful functors; the result follows. □

The left adjoint F is called the *free model functor*; for a set A, the model FA is called the *free model over* A; and for a map $f\colon A \to U\mathcal{M}$, the adjoint homomorphism $\bar{f}\colon FA \multimap \mathcal{M}$ is called the *homomorphism induced by* f. We write $\eta_A\colon A \to FA$ for the unit, as usual.

4.3. **Interpreting values and computations.** We assume given a signature (so that value and computation terms are determined), an effect theory $\mathcal{T}$ over that signature, and an interpretation of the effect theory (so that the category of models, etc., is determined).

We can then interpret value types A by sets $[\![A]\!]$ and computation types $\underline{C}$ by models $[\![\underline{C}]\!]$ of the given effect theory $\mathcal{T}$. Value types are interpreted in the same way as signature types, except for $U\underline{C}$, which is interpreted as the carrier of $[\![\underline{C}]\!]$. The computation type FA is interpreted by the free model on $[\![A]\!]$. Products of computation types are interpreted by product models and function types are interpreted by exponent models. These are defined as follows:

Definition 4.5. For any family of models $\mathcal{M}_i$, where i ranges over the index set I, the *product model* $\prod_{i\in I}\mathcal{M}_i$ is the model with carrier $\prod_{i\in I}|\mathcal{M}_i|$ and componentwise defined operations. For any model $\mathcal{M}$ and any set A, the *exponent model* $\mathcal{M}^A$ is the model with carrier $|\mathcal{M}|^A$ and pointwise defined operations.

Contexts are again interpreted componentwise, with $x_1 : A_1, \ldots, x_m : A_m$ being interpreted by $[\![A_1]\!] \times \cdots \times [\![A_m]\!]$, and $k_1 : \beta_1 \to \underline{C}_1, \ldots, k_n : \beta_n \to \underline{C}_n$ being interpreted by $U[\![\underline{C}_1]\!]^{[\![\beta_1]\!]} \times \cdots \times U[\![\underline{C}_n]\!]^{[\![\beta_n]\!]}$.

We wish to interpret well-typed values $\Gamma \mid K \vdash V : A$ by maps $[\![V]\!] \colon [\![\Gamma]\!] \times [\![K]\!] \to [\![A]\!]$, well-typed computations $\Gamma \mid K \vdash M : \underline{C}$ by maps $[\![M]\!] \colon [\![\Gamma]\!] \times [\![K]\!] \to U[\![\underline{C}]\!]$, and well-typed handlers $\Gamma \mid K \vdash H : \underline{C}$ **handler** by maps

$$[\![H]\!] \colon [\![\Gamma]\!] \times [\![K]\!] \to \prod_{\mathsf{op} : \alpha \to \beta} [\![\alpha]\!] \times U[\![\underline{C}]\!]^{[\![\beta]\!]} \to U[\![\underline{C}]\!]$$

such that, for every $c \in [\![\Gamma]\!] \times [\![K]\!]$, the set $U[\![\underline{C}]\!]$, together with the operations $\mathrm{pr}_{\mathsf{op}}([\![H]\!](c))$, forms a model of $\mathcal{T}$.

However, as discussed informally above, not all handlers receive an interpretation. As handler terms occur in computation terms and computation terms occur in value terms, computation terms and value terms may also not receive an interpretation. Consequently we only have that if, the interpretation $[\![V]\!]$ of a well-typed value $\Gamma \mid K \vdash V : A$ is defined, then it is a map $[\![\Gamma]\!] \times [\![K]\!] \to [\![A]\!]$, and similarly for computation terms and handler terms.

When giving interpretations it is then convenient to use *Kleene equality* $\simeq$ where two sides are equal if either they are both defined and equal or they are both undefined. Fix value and continuation contexts $\Gamma = x_1 : \alpha_1, \ldots, x_m : \alpha_m$ and $K = k_1 : \beta_1 \to \underline{C}_1, \ldots, k_n : \beta_n \to \underline{C}_n$.

Value terms are interpreted as follows:

$$\begin{aligned}
[\![\Gamma \mid K \vdash x_i : A_i]\!] &\simeq \mathrm{pr}_i \circ \mathrm{pr}_1 \quad (1 \le i \le m) \\
[\![\Gamma \mid K \vdash \mathsf{f}(V) : \alpha]\!] &\simeq [\![\mathsf{f}]\!] \circ [\![V]\!] \quad (\mathsf{f} : \alpha \to \beta) \\
[\![\Gamma \mid K \vdash \langle\rangle : \mathbf{1}]\!] &\simeq \mathrm{t} \\
[\![\Gamma \mid K \vdash \langle V, W\rangle : A \times B]\!] &\simeq \langle [\![V]\!], [\![W]\!] \rangle \\
[\![\Gamma \mid K \vdash \ell(V) : \textstyle\sum_{\ell\in L} A_\ell]\!] &\simeq \mathrm{in}_\ell \circ [\![V]\!] \\
[\![\Gamma \mid K \vdash \mathsf{thunk}\, M : U\underline{C}]\!] &\simeq [\![\Gamma \mid K \vdash M : \underline{C}]\!]
\end{aligned}$$

In the above, typing judgements are abbreviated on the right-hand side, $\mathrm{t} \colon A \to \mathbb{1}$ is the map to the one-element set, and $\mathrm{in}_\ell \colon A_\ell \to \sum_{\ell\in L} A_\ell$ is the ℓ-th injection into a disjoint sum.

Next, computation terms, other the handling construct (for which see below) are interpreted as follows:

$$\begin{aligned}
[\![\Gamma \mid K \vdash \mathsf{match}\ V\ \mathsf{with}\ \langle x, y\rangle \mapsto M : \underline{C}]\!] &\simeq [\![M]\!] \circ \langle \mathrm{id}_{[\![\Gamma]\!]\times[\![K]\!]}, [\![V]\!]\rangle \\
[\![\Gamma \mid K \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto M_\ell\}_{\ell\in L} : \underline{C}]\!] &\simeq [[\![M_\ell]\!]]_{\ell\in L} \circ \mathrm{dist} \circ \langle \mathrm{id}_{[\![\Gamma]\!]\times[\![K]\!]}, [\![V]\!]\rangle \\
[\![\Gamma \mid K \vdash \mathsf{force}\ V : \underline{C}]\!] &\simeq [\![\Gamma \mid K \vdash V : U\underline{C}]\!] \\
[\![\Gamma \mid K \vdash \mathsf{return}\ V : FA]\!] &\simeq \eta_A \circ [\![V]\!] \\
[\![\Gamma \mid K \vdash M\ \mathsf{to}\ x : A.\ N : \underline{C}]\!] &\simeq [\![N]\!]^{\dagger_{[\![\underline{C}]\!]}} \circ \langle \mathrm{id}_{[\![\Gamma]\!]\times[\![K]\!]}, [\![M]\!]\rangle \\
[\![\Gamma \mid K \vdash \langle M_\ell\rangle_{\ell\in L} : \textstyle\prod_{\ell\in L}\underline{C}_\ell]\!] &\simeq \langle [\![M_\ell]\!]\rangle_{\ell\in L} \\
[\![\Gamma \mid K \vdash \mathsf{prj}_\ell\, M : \underline{C}_\ell]\!] &\simeq \mathrm{pr}_\ell \circ [\![M]\!] \\
[\![\Gamma \mid K \vdash \lambda x : A.\ M : A \to \underline{C}]\!] &\simeq \mathrm{tr}([\![\Gamma, x : A \mid K \vdash M : \underline{C}]\!]) \\
[\![\Gamma \mid K \vdash M\, V : \underline{C}]\!] &\simeq \mathrm{ev} \circ \langle [\![M]\!], [\![V]\!]\rangle \\
[\![\Gamma \mid K \vdash \mathsf{op}_V(x : \beta.\ M) : \underline{C}]\!] &\simeq \mathsf{op}_{[\![\underline{C}]\!]} \circ \langle [\![V]\!], \mathrm{tr}([\![\Gamma, x : \beta \mid K \vdash M : \underline{C}]\!])\rangle \\
[\![\Gamma \mid K \vdash k_j(V) : \underline{C}_i]\!] &\simeq \mathrm{ev} \circ \langle \mathrm{pr}_j \circ \mathrm{pr}_2, [\![V]\!]\rangle \quad (1 \le j \le n)
\end{aligned}$$

In the above, typing judgements are again abbreviated on the right-hand side, *parameterised lifting* $f^{\dagger_{\mathcal{M}}} \colon A \times UFB \to U\mathcal{M}$ of a map $f \colon A \times B \to U\mathcal{M}$ is given by

$$f^{\dagger_{\mathcal{M}}} \stackrel{\mathrm{def}}{=} U\bar{f} \circ \mathrm{st} \colon A \times UFB \to U\mathcal{M}$$

where $\bar{f}$ is the homomorphism induced by f, and $\mathrm{st} : A \times UFB \to UF(A \times B)$ is the *strength* of the monad UF; we have also used the evident tupling and projections associated to labelled products.

Consider a handler $\Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler}$ where

$$H = \{\mathsf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}:\alpha\to\beta}$$

If defined, the interpretations of the handling terms M_{op} in H yield maps

$$[\![\Gamma, x : \alpha \mid K, k : \beta \to \underline{C} \vdash M_{\mathsf{op}} : \underline{C}]\!] \colon ([\![\Gamma]\!] \times [\![\alpha]\!]) \times ([\![K]\!] \times U[\![\underline{C}]\!]^{[\![\beta]\!]}) \to U[\![\underline{C}]\!]$$

and by suitable rearrangement and transposition, one obtains maps

$$\mathsf{op}_H \colon [\![\Gamma]\!] \times [\![K]\!] \to ([\![\alpha]\!] \times U[\![\underline{C}]\!]^{[\![\beta]\!]} \to U[\![\underline{C}]\!])$$

Hence, for any $c \in [\![\Gamma]\!] \times [\![K]\!]$, we obtain operations $\mathsf{op}_H(c)$ on $U[\![\underline{C}]\!]$.

Definition 4.6. A handler $\Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler}$ is *correct* if the op_H are all defined and if, for all $c \in [\![\Gamma]\!] \times [\![K]\!]$, the set $U[\![\underline{C}]\!]$, together with the $\mathsf{op}_H(c)$, is a model of $\mathcal{T}$.

We can then give the interpretation of handlers:

$$[\![\Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler}]\!] \simeq \begin{cases} c \mapsto \langle \mathsf{op}_H(c)\rangle_{\mathsf{op}} & (\text{if } \Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler} \text{ is correct}) \\ \text{undefined} & (\text{otherwise}) \end{cases}$$

The interpretation of the handling construct

$$\Gamma \mid K \vdash M\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x : A.\ N : \underline{C}$$

is a bit complicated because of the possibility of free variables in H. Suppose that the interpretations of M, H, and $\underline{C}$ are all defined (necessary for that of the handling construct to be defined). Then for every $c \in [\![\Gamma]\!] \times [\![K]\!]$ define $\mathcal{M}_c$ to be the model of $\mathcal{T}$ with carrier $U[\![\underline{C}]\!]$ and operations $\mathrm{pr}_{\mathsf{op}}([\![H]\!](c))$. Taking the product of all these models we obtain

a model $\mathcal{M}_H$ of $\mathcal{T}$ with carrier $U[\![\underline{C}]\!]^{[\![\Gamma]\!]\times[\![K]\!]}$. Next, transposing $[\![\Gamma, x : A \mid K \vdash N : \underline{C}]\!]$ (modulo a rearrangement of factors) we obtain a map $[\![A]\!] \to U[\![\underline{C}]\!]^{[\![\Gamma]\!]\times[\![K]\!]}$. The adjoint homomorphism of this map has type $F[\![A]\!] \multimap [\![\underline{C}]\!]^{[\![\Gamma]\!]\times[\![K]\!]}$ and this, in turn, yields a map $[\![\Gamma]\!] \times [\![K]\!] \times UF[\![A]\!] \to U[\![\underline{C}]\!]$. Combining this last map with the interpretation of M we finally obtain the desired map $[\![\Gamma]\!] \times [\![K]\!] \to U[\![\underline{C}]\!]$.

Writing this out we obtain the interpretation of the handling construct:

$$[\![\Gamma \mid K \vdash M \text{ handled with } H \text{ to } x : A.\ N : \underline{C}]\!] \simeq$$
$$\mathrm{tr}^{-1}(\mathrm{tr}([\![\Gamma, x : A \mid K \vdash N : \underline{C}]\!])^{\dagger_{\mathcal{M}_H}}) \circ \langle \mathrm{id}_{[\![\Gamma]\!]\times[\![K]\!]}, [\![\Gamma \mid K \vdash M : FA]\!] \rangle$$

The correctness of handlers is clearly of central concern. If the effect theory is empty, then any handler is correct, but, in general — and unsurprisingly — correctness is undecidable. Indeed, as will be discussed in Section 5.2, even the decidability of the correctness of very simple handlers is a Π_2-complete problem.

Example 4.7. All the handlers, given in Examples 2.3 and Section 3 are correct. In particular, the exception handler and the stream redirection handlers are trivially correct as the corresponding effect theories are empty.

Both rollback handlers are also correct in the presence of destructive exceptions, given in Examples 4.2. The standard exception handler, however, is not correct for destructive exceptions, as it intercepts an exception and provides a replacement computation instead, but does not correct any modifications made to the state. In particular, the handler does not respect the equation

$$\mathsf{set}_{\ell,x}(\mathsf{raise}_e()) = \mathsf{raise}_e()$$

5. Reasoning about handlers

The semantics we have just given allows us to reason about handlers. We are interested in two questions in particular: what computations are equivalent, and which handlers are correct? We content ourselves in this section with some initial results.

5.1. **Equations.** We fix a signature, and an effect theory $\mathcal{T}$ and its interpretation. As interpretations of value terms, computation terms, or handlers may not be defined, we need somewhat non-standard equational judgements, as well as definedness ones. Given two computations $\Gamma \mid K \vdash M : \underline{C}$ and $\Gamma \mid K \vdash N : \underline{C}$, the *weak equality* judgement

$$\Gamma \mid K \vdash M \eqsim N : \underline{C}$$

holds if $[\![\Gamma \mid K \vdash M : \underline{C}]\!]$ and $[\![\Gamma \mid K \vdash N : \underline{C}]\!]$ are equal in case they are both defined. The *definedness* judgement

$$\Gamma \mid K \vdash M \downarrow$$

holds if $[\![\Gamma \mid K \vdash M : \underline{C}]\!]$ is defined.

The judgements

$$\Gamma \mid K \vdash V \eqsim W : A$$

$$\Gamma \mid K \vdash V \downarrow$$

$$\Gamma \mid K \vdash H \downarrow$$

are understood analogously; the final one can equivalently be understood as asserting handler correctness.

The weak equality and definedness judgements interact. For example (ignoring contexts and types) we have the rule:

$$\frac{M_1 \eqsim M_2 \quad M_2 \downarrow \quad M_2 \eqsim M_3}{M_1 \eqsim M_3}$$

Our calculus extends Levy's call-by-push-value, hence standard equations [10] continue to hold. For example we have β-equivalence for the sequencing construct:

$$\Gamma \mid K \vdash \mathsf{return}\ V\ \mathsf{to}\ x : A.\ M \eqsim M[V/x] : \underline{C}$$

and η-equivalence for functions:

$$\Gamma \mid K \vdash \lambda x : A.\ Mx \eqsim M : A \to \underline{C}$$

Our choice of weak equality was motivated by the desire for an equational version of these and other equivalences. If, instead, we had chosen strong equality (which holds if both sides are defined and equal) we would have needed additional definedness assumptions.

Next, we have equations that describe the pointwise and componentwise behaviour of operations on function and product types, respectively:

$$\Gamma \mid K \vdash \mathsf{op}_V(x : \alpha.\ \lambda y : A.\ M) \eqsim \lambda y : A.\ \mathsf{op}_V(x : \alpha.\ M) : A \to \underline{C}$$
$$\Gamma \mid K \vdash \mathsf{op}_V(x : \alpha.\ \langle M_\ell \rangle_{\ell \in L}) \eqsim \langle \mathsf{op}_V(x : \alpha.\ M_\ell) \rangle_{\ell \in L} : \textstyle\prod_{\ell \in L} \underline{C}_\ell$$

We also have an equation describing the commutativity between operations and sequencing:

$$\Gamma \mid K \vdash \mathsf{op}_V(x : \alpha.\ M)\ \mathsf{to}\ y : A.\ N \eqsim \mathsf{op}_V(x : \alpha.\ M\ \mathsf{to}\ y : A.\ N) : \underline{C}$$

This equation holds as sequencing is defined using the homomorphism induced by the universality of the free model, thus it maps an operation on FA to an operation on $\underline{C}$. The equations given by Levy for two specific operations: printing and divergence (divergence is considered in Section 6), are instances of the above three equations.

We further have a rule for inheritance of equations from the effect theory $\mathcal{T}$. First we can obtain computations from templates by substituting values for value variables and abstracted computations for template variables:

Definition 5.1. The *substitution* of a template

$$x_1 : \alpha_1, \ldots, x_m : \alpha_m \mid z_1 : \beta_1, \ldots, z_n : \beta_n \vdash T$$

by values $\Gamma \mid K \vdash V_i : \alpha_i$ $(1 \le i \le m)$ and computations $\Gamma, y_j : \beta_j \mid K \vdash M_j : \underline{C}$ $(1 \le j \le n)$ is the computation written

$$\Gamma \mid K \vdash T[(V_i/x_i)_i, (y_j : \beta_j.\ M_j/z_j)_j] : \underline{C}$$

and obtained by a standard capture-avoiding substitution of values V_i for the value variables x_i and of abstracted computations $y_j : \beta_j.\ M_j$ for the template variables z_j. In particular, each occurrence $z_j(V)$ of a template variable is replaced by the computation $M_j[V/y_j]$. Substitution behaves structurally on the three other template constructors.

The inheritance rule is then:

$$\frac{x_1 : \alpha_1, \ldots, x_m : \alpha_m \mid z_1 : \beta_1, \ldots, z_n : \beta_n \vdash T_1 = T_2 \in \mathcal{T}}{\Gamma \mid K \vdash T_1[(V_i/x_i)_i, (y_j.\ M_j/z_j)_j] \eqsim T_2[(V_i/x_i)_i, (y_j.\ M_j/z_j)_j] : \underline{C}}$$

where $\Gamma \mid K \vdash V_i : \alpha_i$ $(1 \le i \le m)$ and $\Gamma, y_j : \beta_j \mid K \vdash M_j : \underline{C}$ $(1 \le j \le n)$.

The equations for the handling construct state the universal properties of the induced homomorphism: first, that on values, it extends the inducing map and second, that it acts homomorphically on operations. For any handler $H = \{\mathsf{op}_y(k) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}:\alpha\to\beta}$, the equations are:

$$\Gamma \mid K \vdash \mathsf{return}\ V\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x:A.\ N \eqsim N[V/x] : \underline{C}$$
$$\Gamma \mid K \vdash \mathsf{op}_V(x':\beta.\ M)\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x:A.\ N \eqsim M_{\mathsf{op}}[V/y, x':\beta.\ M\ \mathsf{handled\ with}\ H\ \mathsf{to}\ x:A.\ N/k] : \underline{C}$$

where substitutions of the form $x':\beta.\ M'/k$ replace each occurrence of computations $k(W)$ by $M'[W/x']$ (recall that continuation variables always appear only in the form $k(W)$).

Observe that these last two equations generalise the two given above for the sequencing construct (β-equivalence and commutativity of operations and sequencing). Indeed we have:

$$\Gamma \mid K \vdash M\ \mathsf{to}\ x:A.\ N \eqsim M\ \mathsf{handled\ with}\ \{\}\ \mathsf{to}\ x:A.\ N : \underline{C}$$

stating that sequencing is equivalent to the special case of handling in which all operations are handled by themselves.

There are other equations for exception handlers, given by Benton and Kennedy [2], and Levy [11], and one would wish to generalise these to our general handlers. It turns out that these equations fail for general handlers, but do hold for particular classes of handlers [22]. We do not consider this issue further here.

Finally we give a rule for showing handler correctness. First we need to be able to replace operations in templates by their corresponding definitions in handlers. So, consider a handler $\Gamma \mid K \vdash H : \underline{C}$ **handler**, where H is $\{\mathsf{op}_{x:\alpha}(k:\beta \to \underline{C}) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}:\alpha\to\beta}$, and a template variable context Z. Let K' be the continuation context with a continuation variable $k_z : \alpha \to \underline{C}$ for each template variable $z:\alpha \in Z$.

Then, for every template term $\Gamma' \mid Z \vdash T$ we inductively define a computation term $\Gamma, \Gamma' \mid K, K' \vdash T^H : \underline{C}$ by:

$$z(V)^H = k_z(V)$$
$$(\mathsf{match}\ V\ \mathsf{with}\ \langle x_1, x_2\rangle \mapsto T)^H = \mathsf{match}\ V\ \mathsf{with}\ \langle x_1, x_2\rangle \mapsto T^H$$
$$(\mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell\}_{\ell\in L})^H = \mathsf{match}\ V\ \mathsf{with}\ \{\ell(x_\ell) \mapsto T_\ell^H\}_{\ell\in L}$$
$$\mathsf{op}_V(y:\beta.\ T)^H = M_{\mathsf{op}}[V/x, y:\beta.\ T^H/k]$$

Then, we have:

$$\frac{\Gamma, x:\alpha \mid K, k:\beta \to \underline{C} \vdash M_{\mathsf{op}} \downarrow \quad (\mathsf{op}:\alpha \twoheadrightarrow \beta) \qquad \Gamma, \Gamma' \mid K, K' \vdash T_1^H \eqsim T_2^H : \underline{C} \quad (\Gamma' \mid Z \vdash T_1 = T_2 \in \mathcal{T})}{\Gamma \mid K \vdash H \downarrow}$$

The rule states that a handler is correct when its operation definitions respect the equations of the effect theory.

5.2. **Handler correctness.** The decidability of handler correctness is an interesting question and one potentially pertinent for compiler writers. We now give some results on the decidability of some natural classes of handlers; their proofs are given in Appendix A.

Definition 5.2. A handler $\Gamma \mid K \vdash H : \underline{C}$ **handler** over a given signature is *simple* if there are value variable typings $x_1 : \alpha_1, \ldots, x_m : \alpha_m$ and $f_1 : U(\beta_1 \to \underline{C}), \ldots, f_n : U(\beta_n \to \underline{C})$, where the β_j are arity signature types, such that, up to reordering, Γ is

$$x_1 : \alpha_1, \ldots, x_m : \alpha_m, f_1 : U(\beta_1 \to \underline{C}), \ldots, f_n : U(\beta_n \to \underline{C})$$

and there are continuation typings $k'_1 : \beta'_1 \to \underline{C}, \ldots, k'_p : \beta'_p \to \underline{C}$ such that, up to reordering, K is

$$k'_1 : \beta'_1 \to \underline{C}, \ldots, k'_p : \beta'_p \to \underline{C}$$

and for each $\mathsf{op} : \alpha \rightarrowtriangle \beta$ there is a template

$$x_1 : \alpha_1, \ldots, x_m : \alpha_m, x : \alpha \mid z'_1 : \beta'_1, \ldots, z'_p : \beta'_p, z_1 : \beta_1, \ldots, z_n : \beta_n, z : \beta \vdash T$$

such that the handling term

$$\Gamma, x : \alpha \mid K, k : \beta \to \underline{C} \vdash M_{\mathsf{op}} : \underline{C}$$

is obtained by the substitution of T that replaces x and each x_i by itself, and that replaces z by $(y : \beta.\, k(y))$, each z_j by $(y_j : \beta_j.\, (\mathsf{force}\ f_j) y_j)$, and each z'_l by $(y'_l : \beta'_l.\, k'_l(y'_l))$.

In essence, simple handlers define handling computations in terms of polymorphic constructs only. The exception handler, temporary-state handler, stream redirection handlers, and the CCS renaming and hiding handlers are all simple; none of the parameter-passing handler are simple as they all contain lambda abstractions in their handling terms.

A signature is *simple* if it has no base types or function symbols. In that case there is a unique interpretation, the trivial one, and we will omit mention of it. Simple signatures and theories are equivalent to ones in which all operation symbols are **n**-ary for some n. The signatures given above for exceptions (over a finite set), read-only state, I/O, and nondeterminism are all simple, but none of the others are.

Theorem 5.3. *The problem of deciding, given a simple signature, effect theory, and closed simple handler* $\vdash H : F\mathbf{0}$ **handler** *over the given signature, whether the handler is correct, is* Π_2*-complete.*

The polymorphic nature of template variables means that a template can define a whole family of handlers, one for each computation type. This leads one to the definition of a uniformly simple family of handlers:

Definition 5.4. A family of handlers $\{\Gamma_{\underline{C}} \mid K_{\underline{C}} \vdash H_{\underline{C}} : \underline{C}\,\mathbf{handler}\}_{\underline{C}}$ over a given signature, where $\underline{C}$ ranges over all computation types, is *uniformly simple* if there are value variables $x_1, \ldots, x_m$ and $f_1, \ldots, f_n$, continuation variables $k'_1, \ldots, k'_p$, signature types $\alpha_1, \ldots, \alpha_m$, arity signature types $\beta_1, \ldots, \beta_n$ and $\beta'_1, \ldots, \beta'_p$, and, for each $\mathsf{op} : \alpha \rightarrowtriangle \beta$, there is a template

$$x_1 : \alpha_1, \ldots, x_m : \alpha_m, x : \alpha \mid z'_1 : \beta'_1, \ldots, z'_p : \beta'_p, z_1 : \beta_1, \ldots, z_n : \beta_n, z : \beta \vdash T_{\mathsf{op}}$$

such that for each computation type $\underline{C}$, up to reordering, $\Gamma_{\underline{C}}$ is

$$x_1 : \alpha_1, \ldots, x_m : \alpha_m, f_1 : U(\beta_1 \to \underline{C}), \ldots, f_n : U(\beta_n \to \underline{C})$$

up to reordering, $K_{\underline{C}}$ is

$$k'_1 : \beta'_1 \to \underline{C}, \ldots, k'_p : \beta'_p \to \underline{C}$$

and the handling term

$$\Gamma_{\underline{C}}, x : \alpha \mid K_{\underline{C}}, k : \beta \to \underline{C} \vdash M_{\mathsf{op}} : \underline{C}$$

is obtained by the substitution of T_{op} that replaces x and each x_i by itself, z by $(y : \beta.\, k(y))$, each z_j by $(y_j : \beta_j.\, (\mathsf{force}\ f_j) y_j)$, and each z'_l by $(y'_l : \beta'_l.\, k'_l(y'_l))$.

If a family of handlers is uniformly simple then, as is evident, any member of the family is simple. Conversely, because of the polymorphic nature of templates, any simple handler can be generalised to obtain a uniformly simple one. Of the simple handlers given above, the exception handler, the temporary-state handler, and the stream redirection handlers are also uniformly simple; the CCS renaming and hiding handlers are not.

Note that a uniformly simple family of handlers is determined by the various variables, types and template terms discussed in the above definition. As there are finitely many of these altogether, uniformly simple families of handlers can be finitely presented and so it is proper to ask decidability questions about them.

Definition 5.5. A family of handlers $\{\Gamma_{\underline{C}} \mid K_{\underline{C}} \vdash H_{\underline{C}} : \underline{C}\mathbf{handler}\}_{\underline{C}}$ over a given signature, where $\underline{C}$ ranges over all computation types, is *correct* if each handler in the family is correct.

Since a uniformly simple family of handlers cannot use properties of a specific computation type, it cannot be as contrived as an arbitrary family of handlers; we may therefore expect it to be easier to decide its correctness. As we now see, correctness can become semidecidable:

Theorem 5.6. *The problem of deciding, given a simple signature, effect theory, and a uniformly simple family of closed handlers* $\{\vdash H_{\underline{C}} : \underline{C}\ \mathbf{handler}\}_{\underline{C}}$ *over the given signature, whether the family of handlers is correct, is* Σ_1*-complete.*

Effect theories with a simple signature correspond (see the proof of Proposition 4.4) to ordinary finite equational theories, i.e., those with finitely many finitary function symbols and finitely many axioms. Consequently notions such as decidability can be transferred to them. (All the above examples of simple signatures and theories are decidable in this sense.) With this understanding, we have:

Theorem 5.7. *The problem of deciding, given a simple signature, decidable effect theory, and a uniformly simple family of handlers* $\{\vdash H_{\underline{C}} : \underline{C}\ \mathbf{handler}\}_{\underline{C}}$ *over the given signature, whether the family of handlers is correct, is decidable.*

6. Recursion

Beginning with syntax, signatures are as before except that we always assume an additional operation symbol $\Omega : \mathbf{1} \to \mathbf{0}$, representing nontermination. To provide a recursion facility, we follow [10] and add a fixed-point constructor $\mathsf{rec}\, x : U\underline{C}.\, M$, typed as follows:

$$\frac{\Gamma, x : U\underline{C} \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathsf{rec}\, x : U\underline{C}.\, M : \underline{C}}$$

Turning to semantics, templates are defined as before, but effect theories now consist of *inequations* $\Gamma \mid Z \vdash T_1 \leq T_2$ rather than equations. We always assume that the effect theory contains the inequation $\cdot \mid z : \mathbf{1} \vdash \Omega() \leq z$, stating that Ω is the least element. See [8] for examples.

In order to, eventually, interpret recursion, we move from the category of sets to $\omega\mathbf{Cpo}$, the category of ω-cpos and continuous functions. First we assume given an assignment of ω-cpos to base types, which are flat and countable in the case of arity base types; this yields interpretations $[\![\alpha]\!]$ of signature types as ω-cpos, which are flat and countable in the case of arity signature types. (A ω-cpo is flat if no two distinct elements are comparable.) An

interpretation of an effect theory then consists of such an assignment and an assignment $[\![\mathsf{f}]\!] : [\![\alpha]\!] \to [\![\beta]\!]$ of continuous functions to function symbols $\mathsf{f} : \alpha \to \beta$, and well-formed base values $\Gamma \vdash V : \alpha$ are interpreted as continuous functions $[\![V]\!] : [\![\Gamma]\!] \to [\![\alpha]\!]$ much as before.

Similarly, given an ω-cpo $|\mathcal{M}|$ together with operations, that is, continuous functions $\mathsf{op}_{\mathcal{M}} : [\![\alpha]\!] \times |\mathcal{M}|^{[\![\beta]\!]} \to |\mathcal{M}|$, for each operation symbol $\mathsf{op} : \alpha \twoheadrightarrow \beta$, well-formed templates $\Gamma \mid Z \vdash T$ can be interpreted as continuous functions $[\![T]\!] : [\![\Gamma]\!] \times [\![Z]\!] \to |\mathcal{M}|$.

With that one can define models $\mathcal{M}$ of effect theories $\mathcal{T}$ as ω-cpos together with operations for each operation symbol such that such that $[\![T_1]\!] \leq [\![T_2]\!]$ holds for all the inequations $\Gamma \mid Z \vdash T_1 \leq T_2$ in $\mathcal{T}$. Defining homomorphisms in the evident way, one then obtains a category of models $\mathbf{Mod}_{\mathcal{T}}$, equipped with a forgetful functor $U : \mathbf{Mod}_{\mathcal{T}} \to \omega\mathbf{Cpo}$. As before, the forgetful functor has a left adjoint $F : \omega\mathbf{Cpo} \to \mathbf{Mod}_{\mathcal{T}}$. Both functors have continuous strengths and are locally continuous, by which is meant that they act continuously on the hom ω-cpos. It is also important to note that the unique continuous homomorphism $\bar{f} : FA \multimap \mathcal{M}$ induced by a continuous map $f : A \to U\mathcal{M}$ is itself a continuous function of f. As effect theories assumes least elements, carriers of models are pointed ω-cpos (i.e., ω-cpos with a least element) and homomorphisms are strict continuous functions (i.e., continuous functions preserving least elements).

The monads UF obtained in this way are the standard ones that occur in semantics. For example, the monad obtained from the theory for exceptions together with a least element is $P \mapsto (P + \mathbf{exc})_\perp$, where the *lifting* $Q_\perp$ is the ω-cpo obtained from Q by adding a new least element; the monad for nondeterminism, together with a least element, is $P \mapsto \mathcal{P}(P_\perp)$, where $\mathcal{P}$ is the convex powerdomain monad; and the monad for state, together with a least element and inequations sating that the state operations are strict, is $P \mapsto (S \times P)^S_\perp$. See [8] for further discussion and references.

To show the existence of the free model functor and its properties, one can proceed along related lines to before. Instead of working with a signature of operation symbols of countable arity, one switches to families g_a of operation symbols of countable arity, parameterised by elements a of given ω-cpos P. And, instead of considering equations between the resulting infinitary terms, one considers inequations. Function symbols are interpreted by continuous functions varying continuously over the parameter ω-cpos ; this corresponds to the fact that one is working with continuous functions $\mathsf{op}_{\mathcal{M}} : [\![\alpha]\!] \times |\mathcal{M}|^{[\![\beta]\!]} \to |\mathcal{M}|$ with $[\![\alpha]\!]$ an ω-cpo.

One thereby obtains continuously parameterised countably infinitary inequational; theories; these can be shown to be equivalent to the discrete countable Lawvere $\omega\mathbf{Cpo}$-theories: see [9], and see too [15] for further discussion of parameterised equational logic. The existence of the free model functor and its continuity properties follow from the fact that $\omega\mathbf{Cpo}$ is locally countably presentable as a cartesian closed category (see [9]).

Value types are now interpreted by ω-cpos and computation types are interpreted by models whose carriers are pointed ω-cpos, again making use of product and exponentiation models. Values, computations and handlers are interpreted by continuous maps as before, and such interpretations may not be defined. The fixed-point constructor is interpreted using the usual least fixed-point interpretation:

$$[\![\Gamma \mid K \vdash \mathsf{rec}\, x : U\underline{C}.\, M : \underline{C}]\!] \simeq \lambda a : [\![\Gamma]\!].\, \lambda\kappa : [\![K]\!].\, \mu x : [\![U\underline{C}]\!].\, [\![M]\!]((a, x), \kappa)$$

where, as usual, $\mu x : P.f(x)$ is the least fixed-point of a continuous function f on a pointed ω-cpo P. Note that the interpretation is defined if, and only if, $[\![\Gamma, x : U\underline{C} \mid K \vdash M : \underline{C}]\!]$ is.

The interpretation of the handling construct proceeds as before except that, rather than taking $\mathcal{M}_H$ to be the product of all models $\mathcal{M}_c$, for $c \in [\![\Gamma]\!] \times [\![K]\!]$, one takes it to be the

submodel of that product whose carrier is the ω-cpo $U[\![\underline{C}]\!]^{[\![\Gamma]\!]\times[\![K]\!]}$ of continuous functions from $[\![\Gamma]\!] \times [\![K]\!]$ to $U[\![\underline{C}]\!]$. In this context, note that correct handlers cannot redefine Ω as the theory of Ω fixes it uniquely.

Conclusion

The current work opens some immediate questions. The most important is how to simultaneously handle two computations to describe parallel combinators, e.g., that of CCS or the Unix pipe combinator. Understanding this would bring parallelism within the ambit of the algebraic theory of effects.

Next, the logical ideas of Section 5.1 should be worked out more fully and merged with the general logic for algebraic effects [20, 22]. There is a close correspondence between the handling construct and the free model principle in the logic, which should be examined in detail.

It would be worthwhile to extend the results of Section 5.2 further, whether to wider classes of signatures, handlers, or theories. One would also like to have analogous results for (the interpretation over) ω-cpos.

Only correct handlers can be interpreted, but, as we have seen, obtaining mechanisms that ensure correctness is hard. One option would be to drop equations altogether, when handlers are interpreted as models of absolutely free theories (i.e., those with no equations). This would be correctly implementable if one removed the connection with the equivalences expected for the real effects between handled computations.

More routinely, perhaps, the work done on combinations of effects in [8] should be extended to combinations of handlers, and there should be a general operational semantics [16] which includes that of Benton and Kennedy [2].

Finally, one should develop the programming language aspects further. For example, while call-by-push-value serves well as a fundamental calculus and as an intermediate language, it may be more realistic, or at least more in accordance with current practice, to find a formulation of handlers in a call-by-value context (and call-by-name also has some interest). It would also be important to increase ease of programming, for example by allowing abstraction on handlers, rather than, as above, making use of their global variables; again one would like syntactic support for parametric handlers; some ideas along these lines can be found in [21]. In general, perhaps handlers could become more first-class entities.

Acknowledgments

We thank Andrej Bauer, Andrzej Filinski, Ohad Kammar, Paul Levy, John Power, Mojca Pretnar, and Alex Simpson for their insightful comments and support.

References

[1] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *APPSEM 2000*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122, 2000.

[2] Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, 2001.

[3] Andrzej Filinski. Representing layered monads. In *26th Symposium on Principles of Programming Languages*, pages 175–188, 1999.

[4] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, pages 237–247, 1993.
[5] George A. Grätzer. *Universal Algebra*. Springer, 2nd edition, 1979.
[6] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.
[7] Martin Hyland, Paul Blain Levy, Gordon D. Plotkin, and John Power. Combining algebraic effects with continuations. *Theoretical Computer Science*, 375(1-3):20–40, 2007.
[8] Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.
[9] Martin Hyland and John Power. Discrete Lawvere theories and computational effects. *Theor. Comput. Sci.*, 366(1-2):144–162, 2006.
[10] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.
[11] Paul Blain Levy. Monads and adjunctions for global exceptions. *Electronic Notes in Theoretical Computer Science*, 158:261–287, 2006.
[12] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.
[13] Eugenio Moggi. Notions of computation and monads. *Information And Computation*, 93(1):55–92, 1991.
[14] Simon L. Peyton Jones. Haskell 98. *Journal of Functional Programming*, 13(1):0–255, 2003.
[15] Gordon D. Plotkin. Some varieties of equational logic. In *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 150–156, 2006.
[16] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *4th International Conference on Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24, 2001.
[17] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356, 2002.
[18] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
[19] Gordon D. Plotkin and John Power. Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163, 2004.
[20] Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. In *23rd Symposium on Logic in Computer Science*, pages 118–129, 2008.
[21] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94, 2009.
[22] Matija Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, School of Informatics, University of Edinburgh, 2010.
[23] Terese. *Term Rewriting systems*. Cambridge University Press, 2003.

Appendix A. Decidability of handler correctness

We prove our results on handler correctness by reducing correctness to related questions in equational logic. An interpretation of an equational theory $\mathcal{T}$ in another $\mathcal{T}'$ is given by an assignment of a $\mathcal{T}'$-term with free variables included in $x_1, \ldots, x_n$ to every function symbol of arity n of $\mathcal{T}$. This results in an interpretation of every $\mathcal{T}$-term by a $\mathcal{T}'$-term, and so in an interpretation of every $\mathcal{T}$-equation by a $\mathcal{T}'$-equation.

Lemma A.1. *Given a finitary equational theory with finite signature and finitely many axioms, and an interpretation of this theory in itself, it is a Π_2-complete problem to decide whether the interpretation holds in the initial model of the theory.*

Proof. The problem is clearly in Π_2 as an axiom holds in the initial model of such an equational theory if, and only if, there exists a proof of all its closed instances. Conversely, take a Π_2 sentence of Peano Arithmetic. Without loss of generality, we can assume this to

be of the form $\forall x.\, \exists y.\, \varphi(x, y)$, where $\varphi(x, y)$ defines a primitive recursive relation $R(m, n)$. We now define a finitary equational theory $\mathcal{T}$ with finite signature and finitely many axioms, and an interpretation of it in itself, such that that $\forall x.\, \exists y.\, \varphi(x, y)$ is true if, and only if, the interpretation of the axioms holds in the initial model of the theory.

Changing to $\forall x.\, \exists y.\, \exists y' \leq y.\, \varphi(x, y')$ if necessary, we can assume that if $R(m, n)$ holds and $n \leq n'$, then $R(m, n')$ holds too. Let $f_1, \ldots, f_k$ be a sequence of primitive recursive functions, including a function (coding) disjunction, each definable in terms of the previous ones by composition or primitive recursion, such that f_k is the characteristic function of R. Take a constant symbol zero and a unary function symbol succ. Next, for all the primitive recursive functions f_i take: a function symbol f_i of the same arity, and axioms corresponding to the primitive recursive definition of f_i, written using zero, succ, and the f_j with $j < i$.

Next, take a binary function symbol try and a unary function symbol exists, together with the following two axioms:

$$\begin{aligned} \mathsf{try}(x, y) &= \mathsf{f}_k(x, y) \vee \mathsf{try}(x, \mathsf{succ}(y)) \\ \mathsf{exists}(x) &= \mathsf{try}(x, \underline{0}) \end{aligned}$$

where $\vee$ is the function symbol corresponding to disjunction and $\underline{n}$ is the n-th numeral, defined using zero and succ. All this defines the theory $\mathcal{T}$.

Finally, take the interpretation of the theory $\mathcal{T}$ in itself where each function symbol f other than exists is interpreted by itself — more precisely, the term $\mathsf{f}(x_1, \ldots, x_n)$ — and exists is interpreted by the term $\underline{1}$. This interpretation of the axioms evidently holds in the initial model of $\mathcal{T}$ if, and only if, $\mathsf{try}(t, \underline{0}) = \underline{1}$ is provable for all closed terms t.

Next, assume that the sentence $\forall x.\, \exists y.\, \varphi(x, y)$ is true and choose m. Then there exists an n such that $\mathsf{f}_k(\underline{m}, \underline{n}) = \underline{1}$ is provable in $\mathcal{T}$. The following sequence of equations is then provable in $\mathcal{T}$:

$$\begin{aligned} \mathsf{try}(\underline{m}, \underline{0}) &= \mathsf{f}_k(\underline{m}, \underline{0}) \vee \mathsf{try}(\underline{m}, \underline{1}) \\ &= \mathsf{f}_k(\underline{m}, \underline{0}) \vee \mathsf{f}_k(\underline{m}, \underline{1}) \vee \mathsf{try}(\underline{m}, \underline{2}) \\ &\vdots \\ &= \mathsf{f}_k(\underline{m}, \underline{0}) \vee \cdots \vee \mathsf{f}_k(\underline{m}, \underline{n}) \vee \mathsf{try}(\underline{m}, \underline{n+1}) \\ &= \underline{1} \end{aligned}$$

(with the last holding as $\mathsf{f}_k(\underline{m}, \underline{n}) = \underline{1}$ is provable). More generally, as $\mathsf{f}_k(\underline{m}, \underline{n'}) = \underline{1}$ whenever $n' \geq n$, a similar argument shows that $\mathsf{try}(\underline{m}, \underline{n}) = \underline{1}$ is provable for any m and n. It follows that all closed terms of $\mathcal{T}$ are provably equal to numerals. Therefore, as we have also shown that $\mathsf{try}(\underline{m}, \underline{0}) = \underline{1}$ is provable for all m, the interpretation of the axioms holds in the initial model.

Conversely, assume that $\mathsf{try}(t, \underline{0}) = \underline{1}$ is provable in $\mathcal{T}$ for all closed terms t. Then, in particular, $\mathsf{try}(\underline{m}, \underline{0}) = \underline{1}$ is provable in $\mathcal{T}$ for all m. We analyse proofs in $\mathcal{T}$ via the first-order term rewriting system obtained by orienting all the axioms of $\mathcal{T}$ from left to right. As the reduction rules are left-linear and there are no overlaps between them, we have a Church-Rosser system [23]. Hence, for any two terms t and u, the equation $t = u$ is provable if, and only if, t and u reduce to a common term.

In particular, by our assumption, for every m, there is some number of steps s such that $\mathsf{try}(\underline{m}, \underline{0}) \to^s \underline{1}$. We first observe that $t \vee u \to^* \underline{1}$ if, and only if, $t \to^* \underline{1}$ or $u \to^* \underline{1}$. Using this observation, it follows by induction on s that there exists an n such that $\mathsf{f}_k(\underline{m}, \underline{n}) \to^* \underline{1}$. Therefore the sentence $\forall x.\, \exists y.\, \varphi(x, y)$ is true, concluding the proof. □

Proof of Theorem 5.3. Given a simple signature and theory $\mathcal{T}$, following the proof of Proposition 4.4 one recursively constructs a corresponding finitary equational theory $\mathcal{T}_f$ with finite signature and finitely many axioms whose models are in 1-1 correspondence with those of the given theory. Handlers $\vdash H : F\mathbf{0}$ **handler** then correspond to interpretations of $\mathcal{T}_f$ in itself, and are correct iff the interpretation of $\mathcal{T}_f$ in itself holds in the model of $\mathcal{T}_f$ corresponding to the model $F[\![\mathbf{0}]\!] = [\![F\mathbf{0}]\!]$ of $\mathcal{T}$. As $F[\![\mathbf{0}]\!]$ is the initial model of $\mathcal{T}$ (being its free model on the empty set), that corresponding model of $\mathcal{T}$ is its initial model.

Conversely, given a finitary equational theory $\mathcal{T}$ with finite signature and finitely many axioms, one immediately constructs a simple signature and theory $\mathcal{T}_s$ such that the above construction yields back the original finitary equational theory $\mathcal{T}$ (up to a bijection of its function symbols). Further, given an interpretation of $\mathcal{T}$ in itself one immediately constructs a handler $\vdash H : F\mathbf{0}$ **handler** to which the interpretation of $\mathcal{T}$ in itself corresponds.

The result then follows by combining these facts with Lemma A.1.

□

Lemma A.2. *Given a finitary equational theory with finite signature and finitely many axioms, and an interpretation of this theory in itself, it is a Σ_1-complete problem to decide whether the interpretation is provable in the theory.*

Proof. The proof is very similar to that of Lemma A.1. The problem is clearly in Σ_1. Conversely, consider a Σ_1 sentence of Peano Arithmetic. This can be taken, without loss of generality, to be of the form $\exists y.\, \varphi(y)$, where $\varphi(y)$ defines a primitive recursive predicate R.

One then defines a finitary equational theory $\mathcal{T}$ with finite signature and finitely many axioms, and an interpretation of it in itself, such that that $\exists y.\, \varphi(y)$ is true if, and only if, the interpretation of the axioms holds in all free models of the theory over finite sets. The theory $\mathcal{T}$ is much like before, except that try is unary and exists is a constant, and the corresponding axioms are

$$\mathsf{try}(y) = \mathsf{f}_k(y) \vee \mathsf{try}(\mathsf{succ}(y))$$
$$\mathsf{exists}() = \mathsf{try}(\underline{0})$$

and one considers the interpretation where each function symbol other than exists is interpreted by itself and exists is interpreted by $\underline{1}$. This interpretation of the axioms evidently holds in all free models of $\mathcal{T}$ over finite sets if, and only if, $\mathsf{try}(\underline{0}) = \underline{1}$ is provable.

The rest of the proof then proceeds entirely analogously to, but a little more simply than, that of Lemma A.1.

□

Proof of Theorem 5.6. The proof proceeds analogously to that of Theorem 5.3 except that one considers uniformly simple handlers $\{\vdash H_{\underline{C}} : \underline{C}\ \mathbf{handler}\}_{\underline{C}}$. In particular there is, as before, a finitary equational theory $\mathcal{T}_f$ corresponding to any given theory $\mathcal{T}$ over a given simple signature. As before, a given uniformly simple handler $\{\vdash H_{\underline{C}} : \underline{C}\ \mathbf{handler}\}_{\underline{C}}$ then corresponds to an interpretation of $\mathcal{T}_f$ in itself. The handler is correct if that interpretation holds in all models of $\mathcal{T}_f$ corresponding to those of $\mathcal{T}$ of the form $[\![\underline{C}]\!]$. Since the $[\![C]\!]$ include all free models of $\mathcal{T}$ of the form $F\mathbf{n}$, this last is equivalent to the interpretation holding in all free models of $\mathcal{T}_f$ over finite sets, and so to its being provable in $\mathcal{T}_f$.

□

Proof of Theorem 5.7. Following the proof of Theorem 5.6, we see that, given a theory $\mathcal{T}$ over a given simple signature, determining the correctness of a given uniformly simple handler $\{\vdash H_{\underline{C}} : \underline{C}\mathbf{handler}\}_{\underline{C}}$ amounts to determining whether the corresponding interpretation of $\mathcal{T}_f$ in itself is provable in $\mathcal{T}_f$. However that is decidable since $\mathcal{T}_f$ is.

□