Handlers of Algebraic Effects

Gordon Plotkin[⋆] and Matija Pretnar[⋆⋆]

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Scotland

Abstract. We present an algebraic treatment of exception handlers and, more generally, introduce handlers for other computational effects representable by an algebraic theory. These include nondeterminism, interactive input/output, concurrency, state, time, and their combinations; in all cases the computation monad is the free-model monad of the theory. Each such handler corresponds to a model of the theory for the effects at hand. The handling construct, which applies a handler to a computation, is based on the one introduced by Benton and Kennedy, and is interpreted using the homomorphism induced by the universal property of the free model. This general construct can be used to describe previously unrelated concepts from both theory and practice.

1 Introduction

In seminal work, Moggi proposed a uniform representation of computational effects by monads [14,15,1]. The computations that return values from a set X are modelled by elements of TX, for a suitable monad T. Examples include exceptions, nondeterminism, interactive input/output, concurrency, state, time, continuations, and combinations thereof. Plotkin and Power later proposed to focus on *algebraic* effects, that is, effects that allow a representation by operations and equations [18,20,21]; the operations give rise to the effects at hand. All of the effects mentioned above are algebraic, with the notable exception of continuations [6], which have to be treated differently: see [9] for initial ideas.

In the algebraic approach, an operation gives rise to an occurrence of an effect and its arguments are the possible computations after that occurrence. For example, using a binary choice operation $\mathsf{or}\colon 2$, a nondeterministically chosen boolean is given by the term $\mathsf{or}(\mathsf{return\,true}, \mathsf{return\,false})\colon F\mathbf{bool}$, where $F\sigma$ stands for the type of computations that return values of type σ (we are working in Levy's call-by-push-value (CBPV) framework [11]). The equations of the theory, for example the ones stating that or is a semi-lattice operation, generate the free-model functor, which is used to interpret the type $F\sigma$.

Modulo the forgetful functor, the free model functor is exactly the monad proposed by Moggi to model the corresponding effect [19]. When operations are viewed as a family of functions parametric in X, e.g., $\mathsf{or}_X\colon TX^2 \to TX$, one

[⋆] Supported by EPSRC grant GR/586371/01 and a Royal Society-Wolfson Award Fellowship.

[⋆⋆] Supported by EPSRC grant GR/586371/01.

obtains the so-called *algebraic operations*; such families are characterised by a certain naturality condition [20].

Although the algebraic approach has given ways of constructing, combining [10], and reasoning [22] about effects, it has not yet accounted for their handling. The difficulty is that exception handlers, a well-known programming concept, fail to be algebraic operations [20]. 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 they depend on the effects already created. Filinski's reflection and reification operations provide general effect constructors and deconstructors in the context of layered monads [5].

This paper presents an account of deconstructors for arbitrary algebraic effects, and introduces a handling construct for them. The central new semantic idea is that deconstructing a computation amounts to applying to it a unique homomorphism guaranteed by universality. The domain of this homomorphism is a free model of the algebraic theory of the effects at hand; its range is a programmer-defined model of the algebraic theory; and it extends a programmer-defined map on values. Our new handling construct generalises the exception-handling construct of Benton and Kennedy [2]. It also includes many other, previously unrelated, examples, such as: stream redirection of shell processes, renaming and hiding in CCS [8], timeout, and rollback.

In Section 2, we illustrate the use of homomorphisms via an informal discussion of exception handlers. Then in Sections 3, 4, and 5, we develop a formal calculus in the call-by-push-value framework. This framework includes both call-by-value and call-by-name and proved convenient for the logic of effects in [22]. Section 3 describes the algebraic theory of effects over a given base signature and interpretation. A natural need for two languages arises: one describing handlers, given in Section 4, and one using them to handle computations, given in Section 5. The second parts of these sections give the relevant denotational semantics; readers may wish to omit them at first reading. We give examples in Section 6, where CBPV enables us to define handlers using non-free algebras.

We outline a version of a logic for algebraic effects [22] with handlers in Section 7. In Section 8, we sketch the inclusion of recursion: until then we work only with sets and functions, but everything adapts straightforwardly to ω-cpos (partial orders with sups of increasing sequences) and continuous functions (monotone functions preserving sups of increasing sequences). We conclude in Section 9 with a discussion of some open questions and possible future work.

2 Exception Handlers

We start our study with exception handlers, both because they are an established concept [2,12] and also because exceptions provide the simplest example of algebraic effects. To focus on the exposition of ideas, we write this section in a rather informal style, mixing syntax and semantics.

Taking a set of exceptions E, the computations that return values from a set X are modelled by elements γ of the monad $TX =_{\text{def}} X + E$ with unit

$\eta_X = \mathsf{inl}_X : X \to X + E$. Algebraically, one may take a nullary operation, i.e., a constant, $\mathsf{raise}_e : 0$ for each $e \in E$ and no equations, and then FX has carrier TX with raise_e interpreted as $\mathsf{inr}(e)$.

Fixing X, an exception handler $\gamma\,\mathsf{handle}\,\{e \mapsto \gamma_e\}_{e \in E}$ takes a computation $\gamma \in X + E$ and intercepts raised exceptions $e \in E$, carrying out predefined computations $\gamma_e \in X + E$ instead (if one chooses not to handle a particular exception e one takes $\gamma_e = \mathsf{raise}_e$). So we have the two equations:

$$\eta_X(x)\,\mathsf{handle}\,\{e \mapsto \gamma_e\}_{e \in E} = \mathsf{inl}_X(x) \ ,$$
$$\mathsf{raise}_e\,\mathsf{handle}\,\{e \mapsto \gamma_e\}_{e \in E} = \gamma_e \ .$$

From an algebraic point of view, the γ_e provide a model $\overline{X + E}$ for the theory of exceptions. This model has carrier $X + E$ and, for each e, raise_e is interpreted by γ_e. We then see from the above two equations that

$$\theta(\gamma) =_{\mathrm{def}} \gamma\,\mathsf{handle}\,\{e \mapsto \gamma_e\}_{e \in E}$$

is the unique homomorphism $\theta\colon X + E \to \overline{X + E}$ extending $\mathsf{inl}_X : X \to X + E$ along η_X (we confuse the free model on X with its carrier).

Benton and Kennedy [2] generalised the handling construct to one of the form

$$\mathsf{try}\,x \Leftarrow \gamma\,\mathsf{in}\,g(x)\,\mathsf{unless}\,\{e \mapsto \gamma_e\}_{e \in E} \ ,$$

where exceptions e may be handled by computations γ_e of any given type M (here a model of the theory) and returned values are "handled" with a map $g\colon X \to M$. (This construct is actually a bit more general than in [2] as E may be infinite and as we are in a call-by-push-value framework rather than a call-by-value one.) We now have:

$$\mathsf{try}\,x \Leftarrow \eta_X(x)\,\mathsf{in}\,g(x)\,\mathsf{unless}\,\{e \mapsto \gamma_e\}_{e \in E} = g(x) \ ,$$
$$\mathsf{try}\,x \Leftarrow \mathsf{raise}_e\,\mathsf{in}\,g(x)\,\mathsf{unless}\,\{e \mapsto \gamma_e\}_{e \in E} = \gamma_e \ .$$

As remarked in [2], this handling construct allows a more concise programming style, program optimisations, and a stack-free small-step operational semantics.

Algebraically we now have a model $\overline{M}$ on (the carrier of) M, interpreting raise_e by γ_e, and the handling construct corresponds to the homomorphism θ induced by g, that is the unique homomorphism $\theta\colon X + E \to \overline{M}$ extending g along η_X. Note that *all* the homomorphisms from the free model are obtained in this way, and so (this version of) Benton and Kennedy's handling construct is the most general one possible from the algebraic point of view.

We can now see how to give handlers of other algebraic effects. To give a model of a finitary algebraic theory on a set X is to give a map $f_{\mathsf{op}}\colon X^n \to X$ for each operation $\mathsf{op} : n$, on condition that those maps satisfy the equations of the theory. As before, computations are interpreted in the free model and handling constructs are interpreted by the induced homomorphisms. Intuitively, while exceptions are replaced by handling computations, operations are recursively replaced by handling functions on computations.

3 Effects

We start with a *base signature* Σ_{base}, consisting of: a set of *base types* β; a subset of the base types, called the *arity types* α; a collection of *function symbols* $f : (\boldsymbol{\beta}) \to \beta$; and a collection of *relation symbols* $R : (\boldsymbol{\beta})$. We use vector notation $\boldsymbol{a}$ to abbreviate lists $a_1, \ldots, a_n$.

Base terms v are built from variables x and function symbols, while *base formulas* ψ are built from equations between base terms, relation symbols applied to base terms, logical connectives, and quantifiers over base types; we may omit empty parentheses in terms and formulas, and in similar constructs introduced below. In a context Γ of variables bound to base types, we type base terms as $\Gamma \vdash v : \beta$ and base formulas as $\Gamma \vdash \psi : \mathbf{form}$.

An *interpretation of the base signature* is given by: a set $[\![\beta]\!]$ for each base type β, countable if β is an arity type; a map $[\![f]\!] : [\![\boldsymbol{\beta}]\!] \to [\![\beta]\!]$ for each function symbol $f : (\boldsymbol{\beta}) \to \beta$; and a subset $[\![R]\!] \subseteq [\![\boldsymbol{\beta}]\!]$ for each relation symbol $R : (\boldsymbol{\beta})$, where $[\![\boldsymbol{\beta}]\!] = [\![\beta_1]\!] \times \ldots [\![\beta_n]\!]$. Terms $\Gamma \vdash v : \beta$ and formulas $\Gamma \vdash \psi : \mathbf{form}$ are interpreted by maps $[\![v]\!] : [\![\Gamma]\!] \to [\![\beta]\!]$ and subsets $[\![\psi]\!] \subseteq [\![\Gamma]\!]$ as usual [4].

3.1 Effect Theories

Standard equational logic does not give a finitary notation for describing effects given by an infinite family of operations, having an infinite number of outcomes, or described by an infinite number of equations [20]. We present a more general notation to do this, at least in some cases.

To avoid infinite families of operation symbols, we allow operations to have parameters of base types. For example, instead of having a family of operation symbols $\mathsf{update}_{l,d} : 1$ for each location l and datum d, we employ a single operation symbol $\mathsf{update} : \mathbf{loc}, \mathbf{dat}; 1$ that takes parameters $l : \mathbf{loc}$ and $d : \mathbf{dat}$, giving a memory location to be updated and a datum to be stored there.

To avoid operation symbols of infinite arity, their arguments are allowed to depend on an element of an arity type. For example, $\mathsf{lookup} : \mathbf{loc}; \mathbf{dat}$ has parameter $l : \mathbf{loc}$ and a single argument, dependent on a $d : \mathbf{dat}$. The parameter gives a memory location to be looked-up and the argument gives the computation to be then carried out, dependent on the datum stored in that location.

Thus, given a base signature Σ_{base}, an *effect signature* Σ_{eff} consists of *operation symbols* $\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$, where $\boldsymbol{\beta}$ is a list of *parameter* base types, and $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ are lists of *argument* arity types. We omit the semicolon when $\boldsymbol{\beta}$ is empty, and write n instead of $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ when every $\boldsymbol{\alpha}_i$ is empty. *Effect terms* T are given by the following grammar:

$$T ::= z(\boldsymbol{v}) \mid \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i : \boldsymbol{\alpha}_i . T_i)_i \ ,$$

where z ranges over *effect variables*, and $\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i : \boldsymbol{\alpha}_i . T_i)_i$ is an abbreviation for $\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_1 : \boldsymbol{\alpha}_1 . T_1, \ldots, \boldsymbol{x}_n : \boldsymbol{\alpha}_n . T_n)$. We may omit empty binders here and in similar constructs below.

We type effect terms as $Z;\Gamma \vdash T$, where Z consists of effect variables $z\colon(\boldsymbol{\alpha})$, according to the following rules:

$$\frac{\Gamma \vdash \boldsymbol{v}\colon\boldsymbol{\alpha}}{Z;\Gamma \vdash z(\boldsymbol{v})} \quad (z\colon(\boldsymbol{\alpha}) \in Z)$$

$$\frac{\Gamma \vdash \boldsymbol{v}\colon\boldsymbol{\beta} \qquad Z;\Gamma,\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i \vdash T_i \quad (i = 1,\dots,n)}{Z;\Gamma \vdash \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i.T_i)_i} \quad (\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}})\ .$$

Next, *conditional equations* have the form $Z;\Gamma \vdash T_1 = T_2$ (ψ), assuming that $Z;\Gamma \vdash T_1$, $Z;\Gamma \vdash T_2$, and $\Gamma \vdash \psi\colon\mathbf{form}$. Finally, a *conditional effect theory* $\mathfrak{T}_{\mathrm{eff}}$ *(over base and effect signatures* Σ_{base} *and* Σ_{eff}*)* is a collection of such equations. It would be interesting to develop an equational logic for such theories [17].

Example 1. To describe a set E of exceptions, the base signature consists of a base type **exc** and a constant function symbol $e\colon() \to \mathbf{exc}$ for each $e \in E$. We interpret **exc** by E and function symbols by their corresponding elements. The effect signature consists of an operation symbol $\mathsf{raise}: \mathbf{exc}; 0$, while the effect theory is empty. Then, omitting empty parentheses, raise_e is the computation that raises the exception e.

Example 2. For nondeterminism, we take the empty base signature, the empty interpretation, the effect signature with a single nondeterministic choice operation symbol $\mathsf{or}\colon 2$, and the effect theory for a semi-lattice, which states that or is idempotent, commutative, and associative.

Example 3. For state, the base signature contains a base type **loc** of memory locations, an arity type **dat** of data, and appropriate function and relation symbols for the locations and data. We interpret **loc** by a finite set L and **dat** by a countable set D. The effect signature consists of operation symbols $\mathsf{lookup}\colon \mathbf{loc}; \mathbf{dat}$ and $\mathsf{update} : \mathbf{loc}, \mathbf{dat}; 1$, while the effect theory consists of seven conditional equations [19,17]. As an example, $\mathsf{lookup}_l(d\colon\mathbf{dat}.\mathsf{update}_{l',d}(z))$ is the computation that copies d from l to l' and then proceeds as z.

Each effect theory $\mathfrak{T}_{\mathrm{eff}}$ and interpretation of the base signature induces a standard, possibly infinitary, equational theory [7]. For each $\mathsf{op}: \boldsymbol{\beta}; \boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n$ and $\boldsymbol{b} \in [\![\boldsymbol{\beta}]\!]$, we take an operation symbol $\mathsf{op}_{\boldsymbol{b}}$ of countable arity $\sum_i |[\![\boldsymbol{\alpha}_i]\!]|$. Then each effect term $Z;\Gamma \vdash T$ and $\boldsymbol{c} \in [\![\Gamma]\!]$ gives rise to a, possibly infinitary, term $T_{\boldsymbol{c}}$, with variables of the form $z_{\boldsymbol{a}}$ $(z : (\boldsymbol{\alpha}) \in Z$, $\boldsymbol{a} \in [\![\boldsymbol{\alpha}]\!])$. The equations of the theory are $T_{\boldsymbol{c}} = T'_{\boldsymbol{c}}$ for each $Z;\Gamma \vdash T = T'$ (ψ) in $\mathfrak{T}_{\mathrm{eff}}$ and $\boldsymbol{c} \in [\![\psi]\!]$ $(\subseteq [\![\Gamma]\!])$.

An *interpretation* of Σ_{eff} has a set M, its *carrier*, together with a map

$$\mathsf{op}_M\colon [\![\boldsymbol{\beta}]\!] \times \prod_i M^{[\![\boldsymbol{\alpha}_i]\!]} \to M$$

for each $\mathsf{op}: \boldsymbol{\beta}; \boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$; it is a *model of the effect theory* $\mathfrak{T}_{\mathrm{eff}}$ if the corresponding maps $\mathsf{op}_M(\boldsymbol{b}, -)$, where $\boldsymbol{b} \in [\![\boldsymbol{\beta}]\!]$, satisfy the equations of the induced equational theory. A *homomorphism* between models M and N is a map $\theta\colon M \to N$ such that $\mathsf{op}_N \circ (\mathbf{id}_{[\![\boldsymbol{\beta}]\!]} \times \prod_i \theta^{[\![\boldsymbol{\alpha}_i]\!]}) = \theta \circ \mathsf{op}_M$ holds for all $\mathsf{op}\colon\boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}$.

Models and homomorphisms form a category $\mathrm{Mod}_{\mathfrak{T}_{\text{eff}}}$, equipped with the forgetful functor $U\colon \mathrm{Mod}_{\mathfrak{T}_{\text{eff}}} \to \mathbf{Set}$, which maps a model to its carrier and a homomorphism to its underlying map. This functor has a left adjoint F, which constructs the free model FX on a set of generators X. The set UFX represents the set of computations that return values in X, and the monad UF agrees [19] with the monad proposed by Moggi to model the corresponding effect [15] (assuming the effect theory appropriately chosen).

The monad induced by the theory for exceptions in Example 1 maps a set X to $X + E$, the one for non-determinism in Example 2 maps it to the set $\mathcal{F}^+(X)$ of finite non-empty subsets of X, while the one for state in Example 3 maps it to $(S \times X)^S$, where $S = D^L$. One can give an equivalent treatment using countable Lawvere theories [23].

4 Handlers

Exception handlers are usually described and used within the same language: for each exception, we give a replacement computation term, which can contain further exception handlers. Repeating the same procedure for other algebraic effects is less straightforward: in order to interpret the handling construct, the handlers have to be correct in the sense that the redefinition of the operations they provide yields a model of the effect theory.

Equipping a single calculus with a mechanism to verify that handlers are correct would result in a complex interdependence between well-formedness and correctness. We avoid this by providing two calculi: one, given in this section, enables the language designer to specify handlers; another, given in the next section, enables the programmer to use them. In this way the selection of correct handlers is delegated to the meta-level.

Handler types χ, *handler terms* w, and *handlers* h are given by the following grammar:

$$
\begin{aligned}
\chi &::= X \mid F\sigma \mid \mathbf{1} \mid \chi_1 \times \chi_2 \mid \sigma \to \chi \\
w &::= \varphi(\boldsymbol{v}) \mid \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i\!:\!\boldsymbol{\alpha}_i.w_i)_i \mid \mathsf{if}\,\psi\,\mathsf{then}\,w_1\,\mathsf{else}\,w_2 \mid \mathsf{return}\,v \mid \\
&\qquad \mathsf{let}\,x\!:\!\sigma\,\mathsf{be}\,w\,\mathsf{in}\,w' \mid \star \mid \langle w_1, w_2\rangle \mid \mathsf{fst}\,w \mid \mathsf{snd}\,w \mid \lambda x\!:\!\sigma.w \mid wv \\
h &::= (\boldsymbol{\varphi}_p\!:\!\boldsymbol{\chi}; \boldsymbol{x}_p\!:\!\boldsymbol{\sigma}).\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \mapsto w_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\text{eff}}}\ ,
\end{aligned}
$$

where X ranges over *type variables*, σ ranges over *value types* (here the same as the base types), φ ranges over *handler variables*, and ψ ranges over quantifier-free formulas. A handler is given by a handling term for each operation, dependent on parameters $\boldsymbol{x}_p$ and $\boldsymbol{\varphi}_p$. We may omit the semicolon in handlers when either $\boldsymbol{\sigma}$ or $\boldsymbol{\chi}$ is empty, and also in similar constructs introduced below. When $\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \mapsto w_{\mathsf{op}}$ is omitted, we assume that $w_{\mathsf{op}} = \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i\!:\!\boldsymbol{\alpha}_i.\varphi_i(\boldsymbol{x}_i))_i$, so that op is not handled.

We type handler terms as $\Phi; \Gamma \vdash w : \chi$ where Φ is a context of handler variables $\varphi : (\boldsymbol{\alpha}) \to \chi$, according to the following rules:

$$\frac{\Gamma \vdash \boldsymbol{v} : \boldsymbol{\alpha}}{\Phi; \Gamma \vdash \varphi(\boldsymbol{v}) : \chi} \quad (\varphi : (\boldsymbol{\alpha}) \to \chi \in$$

$$\Phi)\frac{\Gamma \vdash \boldsymbol{v} : \boldsymbol{\beta} \qquad \Phi; \Gamma, \boldsymbol{x}_i : \boldsymbol{\alpha}_i \vdash w_i : \chi \quad (i = 1, \ldots, n)}{\Phi; \Gamma \vdash \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i : \boldsymbol{\alpha}_i . w_i)_i : \chi} \quad (\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}})$$

$$\frac{\Gamma \vdash v : \sigma}{\Phi; \Gamma \vdash \mathsf{return}\, v : F\sigma} \qquad \frac{\Phi; \Gamma \vdash w : F\sigma \qquad \Phi; \Gamma, x : \sigma \vdash w' : \chi}{\Phi; \Gamma \vdash \mathsf{let}\, x : \sigma\, \mathsf{be}\, w\, \mathsf{in}\, w' : \chi} ,$$

and the standard rules for conditionals, products, and functions.

Handlers are typed as $\vdash h : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi$ **handler** by the following rule:

$$\frac{\boldsymbol{\varphi}_p : \boldsymbol{\chi}, (\varphi_i : (\boldsymbol{\alpha}_i) \to \chi)_{i=1}^n; \boldsymbol{x}_p : \boldsymbol{\sigma}, \boldsymbol{x} : \boldsymbol{\beta} \vdash w_{\mathsf{op}} : \chi \quad (\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}})}{\vdash (\boldsymbol{\varphi}_p : \boldsymbol{\chi}; \boldsymbol{x}_p : \boldsymbol{\sigma}).\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \mapsto w_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\text{eff}}} : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi \textbf{ handler}} .$$

Note that a handler may be polymorphic, as type variables may occur in $\boldsymbol{\chi}$ or χ. We say that a handler $\vdash h : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi$ **handler** is *uniform* if $\chi = X$, and *parametrically uniform* if $\chi = \sigma \to X$.

4.1 Semantics

For each assignment ρ of models to type variables, handler types χ are interpreted by models $[\![\chi]\!]_\rho$, given by

$$[\![X]\!]_\rho = \rho(X) \qquad [\![F\sigma]\!]_\rho = F[\![\sigma]\!] \qquad [\![\mathbf{1}]\!]_\rho = \mathbf{1}$$
$$[\![\chi_1 \times \chi_2]\!]_\rho = [\![\chi_1]\!]_\rho \times [\![\chi_2]\!]_\rho \qquad [\![\sigma \to \chi]\!]_\rho = [\![\chi]\!]_\rho^{[\![\sigma]\!]} ,$$

where the model is given component-wise on $M_1 \times M_2$ and point-wise on M^A.

Then, we interpret contexts $\Phi = \varphi_1 : (\boldsymbol{\alpha}_1) \to \chi_1, \ldots, \varphi_n : (\boldsymbol{\alpha}_n) \to \chi_n$ by $[\![\Phi]\!]_\rho = U[\![\chi_1]\!]_\rho^{[\![\boldsymbol{\alpha}_1]\!]} \times \cdots \times U[\![\chi_n]\!]_\rho^{[\![\boldsymbol{\alpha}_n]\!]}$ and handler terms $\Phi; \Gamma \vdash w : \chi$ by maps $[\![w]\!]_\rho \colon [\![\Phi]\!]_\rho \times [\![\Gamma]\!] \to U[\![\chi]\!]_\rho$, defined inductively on valid typing judgements by

$$[\![\Phi; \Gamma \vdash \varphi_i(\boldsymbol{v}) : \chi_i]\!]_\rho = \mathbf{ev}_{[\![\boldsymbol{\alpha}_i]\!], U[\![\chi_i]\!]_\rho} \circ \langle \mathbf{pr}_i \circ \mathbf{pr}_1, \langle [\![\boldsymbol{v}]\!] \rangle \circ \mathbf{pr}_2 \rangle$$
$$[\![\Phi; \Gamma \vdash \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i : \boldsymbol{\alpha}_i . w_i)_i : \chi]\!]_\rho = \mathsf{op}_{[\![\chi]\!]_\rho} \circ \langle \langle [\![\boldsymbol{v}]\!] \rangle \circ \mathbf{pr}_2, \widehat{[\![w_1]\!]}_\rho, \ldots, \widehat{[\![w_n]\!]}_\rho \rangle$$
$$[\![\Phi; \Gamma \vdash \mathsf{return}\, v : F\sigma]\!]_\rho = \eta_{[\![\sigma]\!]} \circ [\![v]\!] \circ \mathbf{pr}_2$$
$$[\![\Phi; \Gamma \vdash \mathsf{let}\, x : \sigma\, \mathsf{be}\, w\, \mathsf{in}\, w' : \chi]\!]_\rho = [\![w']\!]_\rho^\dagger \circ \langle \mathbf{id}_{[\![\Phi]\!]_\rho \times [\![\Gamma]\!]}, [\![w]\!]_\rho \rangle ,$$

where judgements are abbreviated to terms on the right. The maps **ev** and **pr** are evaluation and projection functions; $\widehat{-}$ is the transpose operation (associativity isomorphisms are omitted here, and below); and η is the unit of UF. The map $f^\dagger =_{\text{def}} U(\varepsilon_M \circ Ff) \circ \mathbf{st}_{A,B} \colon A \times UFB \to UM$ is the *parameterised lifting* of $f \colon A \times B \to UM$, where ε is the counit of FU, and $\mathbf{st}$ is the strength of UF. Conditionals, products, and functions are interpreted as usual [11].

A handler $h : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi$ **handler** is interpreted by a parameterised family $[\![h]\!]_\rho(\boldsymbol{m}_p, \boldsymbol{a}_p)$ of Σ_{eff}-interpretations, where $\boldsymbol{m}_p \in U[\![\boldsymbol{\chi}]\!]_\rho$ and $\boldsymbol{a}_p \in [\![\boldsymbol{\sigma}]\!]$; each such interpretation has carrier $U[\![\chi]\!]_\rho$, and, for each $\mathsf{op}\!:\!\boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$, the map

$$\mathsf{op}_{U[\![\chi]\!]_\rho} =_{\text{def}} (\boldsymbol{m}, \boldsymbol{a}) \mapsto [\![w_{\mathsf{op}}]\!]((\boldsymbol{m}_p, \boldsymbol{m}), (\boldsymbol{a}_p, \boldsymbol{a}))$$

(identifying models M with their trivial powers $M^{\mathbf{1}}$).

We say that h is *correct (with respect to $\mathfrak{T}_{\text{eff}}$)* if for all assignments ρ, and for all $\boldsymbol{m}_p \in U[\![\boldsymbol{\chi}]\!]_\rho$ and $\boldsymbol{a}_p \in [\![\boldsymbol{\sigma}]\!]$, the Σ_{eff}-interpretation $[\![h]\!]_\rho(\boldsymbol{m}_p, \boldsymbol{a}_p)$ defines a model of the effect theory $\mathfrak{T}_{\text{eff}}$ on $U[\![\chi]\!]_\rho$. If the effect theory is empty, then any handler is correct, but, in general, correctness is undecidable. In particular, the following problem is Π_1-complete: given a multi-sorted finitary equational theory with finite signature and finitely many axioms, decide whether, in the initial model, a given interpretation of the theory in itself satisfies the axioms.

5 Computations

We assume a *handler signature* Σ_{hand} of *handler symbols*

$$H\!:\!(\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi \ \textbf{handler}\,.$$

Then, *computation types* $\underline{\tau}$ and *terms* t are given by the following grammar:

$$\begin{aligned}
\underline{\tau} &::= F\sigma \mid \mathbf{1} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \sigma \to \underline{\tau}\,,\\
t &::= \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i\!:\!\boldsymbol{\alpha}_i.t_i)_i \mid \mathsf{if}\,\psi\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \mid \mathsf{return}\,v \mid \mathsf{let}\,x\!:\!\sigma\,\mathsf{be}\,t\,\mathsf{in}\,t' \mid\\
&\qquad \mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t}; \boldsymbol{v})\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t' \mid \star \mid \langle t_1, t_2\rangle \mid \mathsf{fst}\,t \mid \mathsf{snd}\,t \mid \lambda x\!:\!\sigma.t \mid tv\,,
\end{aligned}$$

where ψ ranges over quantifier-free formulas, as before.

One can see that computation types and terms mirror their handler counterparts, with the omission of type and handler variables, and with the addition of the handling construct. When the full handling construct is not necessary, we write $\mathsf{handle}\,t\,\mathsf{with}\,H(\boldsymbol{t}; \boldsymbol{v})$ instead of $\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t}; \boldsymbol{v})\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,\mathsf{return}\,x$, and when the handler signature consists of a single handler symbol H, we omit it, writing $\mathsf{try}\,t\,\mathsf{with}\,\boldsymbol{v}; \boldsymbol{t}\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t'$ or $\mathsf{handle}\,t\,\mathsf{with}\,\boldsymbol{t}; \boldsymbol{v}$ instead.

We can extend both handlers and computations with other call-by-push-value types and terms [11,22]. A problem arises if we introduce thunks: handler terms then contain value terms, which contain thunked computation terms, which contain the handling construct. To resolve the issue, one further splits the handler types and terms into value and computation ones.

Computation terms are typed as $\Gamma \vdash t : \underline{\tau}$ according to rules similar to the ones for handling terms, except that the handling construct for a handler symbol $H\!:\!(\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi$ **handler** $\in \Sigma_{\text{hand}}$ is typed by

$$\frac{\Gamma \vdash t\!:\!F\sigma \qquad \Gamma \vdash \boldsymbol{t}\!:\!\boldsymbol{\chi}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}] \qquad \Gamma \vdash \boldsymbol{v}\!:\!\boldsymbol{\sigma} \qquad \Gamma, x\!:\!\sigma \vdash t'\!:\!\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]}{\Gamma \vdash \mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t}; \boldsymbol{v})\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t'\!:\!\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]}\,,$$

where $\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ is the computation type obtained by replacing all the type variables $\boldsymbol{X}$ in χ by the computation types $\underline{\boldsymbol{\tau}}$.

5.1 Semantics

To interpret computation terms, we assume given a *handler definition* Δ, mapping each handler symbol $H\colon(\boldsymbol{\chi};\boldsymbol{\sigma})\to\chi$ **handler** $\in\Sigma_{\mathrm{hand}}$ to a correct handler $\vdash\Delta(H):(\boldsymbol{\chi};\boldsymbol{\sigma})\to\chi$ **handler**. Then, computation types and terms are interpreted in the same way as their handler counterparts, except that the handling construct $\Gamma\vdash\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t};\boldsymbol{v})\,\mathsf{as}\,x\colon\sigma\,\mathsf{in}\,t'\colon\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ is interpreted along the lines discussed in Section 2, as we now see.

Take $\boldsymbol{c}\in[\![\Gamma]\!]$ and let ρ be an assignment that maps X_i to $[\![\underline{\tau}_i]\!]$. Since each handler $\Delta(H)$ is correct, the Σ_{eff} interpretation $[\![\Delta(H)]\!]_\rho(\langle[\![\boldsymbol{t}]\!]\rangle(\boldsymbol{c}),\langle[\![\boldsymbol{v}]\!]\rangle(\boldsymbol{c}))$ is a model $\overline{U[\![\chi]\!]}_\rho$ of the effect theory $\mathfrak{T}_{\mathrm{eff}}$ with carrier $U[\![\chi]\!]_\rho$. By the universality of the free model $F[\![\sigma]\!]$, there is a unique homomorphism $\theta_{\boldsymbol{c}}:F[\![\sigma]\!]\to\overline{U[\![\chi]\!]}_\rho$ extending $[\![t']\!](\boldsymbol{c},-)$ in the sense that the following diagram commutes:

$$\begin{tikzcd}
{[\![\sigma]\!]} \arrow[d, "\eta_{[\![\sigma]\!]}"'] \arrow[dr, "{[\![t']\!](\boldsymbol{c},-)}"] & \\
UF[\![\sigma]\!] \arrow[r, "\theta_{\boldsymbol{c}}"] & U[\![\chi]\!]_\rho
\end{tikzcd}$$

The handling construct $\Gamma\vdash\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t};\boldsymbol{v})\,\mathsf{as}\,x:\sigma\,\mathsf{in}\,t':\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ is then interpreted by the map

$$\boldsymbol{c}\mapsto\theta_{\boldsymbol{c}}([\![t]\!](\boldsymbol{c}))\colon[\![\Gamma]\!]\to U[\![\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]]\!]$$

(note that $[\![\chi]\!]_\rho$ is equal to $[\![\chi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]]\!]$ by the definition of ρ).

6 Examples

We give a number of examples to demonstrate the versatility of our handling construct.

6.1 Exceptions

The standard uniform exception handler is given by

$$(\varphi\colon\mathbf{exc}\to X).\{\mathsf{raise}_e\mapsto\varphi e\}\colon(\mathbf{exc}\to X)\to X\ \mathbf{handler}\ .$$

Benton and Kennedy's construct $\mathsf{try}\,x\Leftarrow t\,\mathsf{in}\,t'\,\mathsf{unless}\,\{e_1\Rightarrow t_1\mid\cdots\mid e_n\Rightarrow t_n\}$ can then be written as $\mathsf{try}\,t\,\mathsf{with}\,t_{\mathrm{exc}}\,\mathsf{as}\,x\colon\sigma\,\mathsf{in}\,t'$ for a suitable σ and $t_{\mathrm{exc}}\colon\mathbf{exc}\to\underline{\tau}$.

Benton and Kennedy noted a few issues about the syntax of their construct when used for programming [2]. It is not obvious that t is handled whereas t' is not, especially when t' is large and the handler is obscured. An alternative they propose is $\mathsf{try}\,x\Leftarrow t\,\mathsf{unless}\,\{e_1\Rightarrow t_1\mid\ldots\mid e_n\Rightarrow t_n\}_i\,\mathsf{in}\,t'$, but then it is not obvious that x is bound in t', but not in the handler. The syntax of our construct $\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{t};\boldsymbol{v})\,\mathsf{as}\,x:\sigma\,\mathsf{in}\,t'$ addresses those issues and clarifies the order of evaluation: after t is handled with H, its results are bound to x and used in t'.

6.2 Stream Redirection

Shell processes in UNIX-like operating systems communicate with the user using input and output streams, usually connected to a keyboard and a terminal window. However, such streams can be rerouted to other processes so simple commands can be combined into more powerful ones.

One case is the redirection `proc > outfile` of the output stream of a process `proc` to a file `outfile`, usually used to store the output for a future analysis. An alternative is the redirection `proc > /dev/null` to the null device, which effectively discards the standard output stream.

Another case is the pipe `proc1 | proc2`, where the output of `proc1` is fed to the input of `proc2`. For example, to get a way (not necessarily the best one) of routinely confirming a series of actions, for example deleting a large number of files, we write `yes | proc`, where the command `yes` outputs an infinite stream made of a predetermined character (the default one being `y`).

We represent interactive input/output by: a base signature, consisting of a base type **char** of characters and constants $\mathsf{a}, \mathsf{b}, \ldots$ of type **char**, together with the obvious interpretation; an effect signature, consisting of operation symbols $\mathsf{out} : \mathbf{char}; 1$ and $\mathsf{in} : \mathbf{char}$, with the empty effect theory. Then, if t is a computation, we can express `yes` | t `> /dev/null` by $\mathsf{handle}\, t\, \mathsf{with}\, H_{\mathrm{red}}$, where $H_{\mathrm{red}} : () \to X$ **handler** is defined to be $\{\mathsf{out}_c(\varphi) \mapsto \varphi, \mathsf{in}(\varphi) \mapsto \varphi(\mathsf{y})\}$.

6.3 CCS Renaming and Hiding

In functional programming, processes are regarded as programs of empty type **0**. The subset of CCS processes [13], given by action prefix and sum, can be represented by: a base signature, consisting of a base type **act** of actions and appropriate constants for actions, interpreted in the evident way; an effect signature, consisting of operation symbols $\mathsf{0} : 0$, $\mathsf{do} : \mathbf{act}; 1$, and $+ : 2$, with the obvious effect theory [22]. Then, process renaming $t[b/a]$ can be written as $\mathsf{handle}\, t\, \mathsf{with}\, H_{\mathrm{ren}}(a, b)$, where, writing $a.\varphi$ for $\mathsf{do}_a(\varphi)$, $H_{\mathrm{ren}} : (\mathbf{act}, \mathbf{act}) \to F\mathbf{0}$ **handler** is defined by:

$$H_{\mathrm{ren}} = (a : \mathbf{act}, b : \mathbf{act}).\{a'.\varphi \mapsto \mathsf{if}\, a' = a\, \mathsf{then}\, b.\varphi\, \mathsf{else}\, a'.\varphi\} \ .$$

Note that 0 and + are handled by themselves, following the convention given above regarding operation symbols omitted from handlers.

Hiding can be implemented in a similar way, but whether parallel can be is an open question. The difficulty is that the natural definition of parallel is by a simultaneous recursion on the structure of *both* its arguments, whereas the handler mechanism provides only recursion on one argument. We should mention that some attempts at finding a binary variant of handlers were unsuccessful.

6.4 Explicit Nondeterminism

The evaluation of a nondeterministic computation usually takes only one of all the possible paths. But in logic programming [3], we do an exhaustive search for all solutions that satisfy given constraints in the order given by the solver implementation. Such nondeterminism is represented slightly differently from the one in Example 2. We take: the empty base signature; an effect signature, consisting of operation symbols $\mathsf{fail}:0$ and $\mathsf{pick}:2$, with the effect theory consisting of the following equations stating that the operations form a monoid:

$$z \vdash \mathsf{pick}(z, \mathsf{fail}) = \mathsf{pick}(\mathsf{fail}, z) = z \ ,$$
$$z_1, z_2, z_3 \vdash \mathsf{pick}(z_1, \mathsf{pick}(z_2, z_3)) = \mathsf{pick}(\mathsf{pick}(z_1, z_2), z_3) \ .$$

The free-model monad maps a set to the set of all finite sequences of its elements, which is Haskell's nondeterminism monad [16].

A user is usually presented with a way of browsing through those solutions, for example extracting all the solutions into a list. Since our calculus has no polymorphic lists (although it can easily be extended with them), we take base types α and $\mathbf{list}_\alpha$, function symbols $\mathsf{nil} : () \to \mathbf{list}_\alpha$, $\mathsf{cons} : (\alpha, \mathbf{list}_\alpha) \to \mathbf{list}_\alpha$, $\mathsf{head}:(\mathbf{list}_\alpha) \to \alpha$, $\mathsf{tail}:(\mathbf{list}_\alpha) \to \mathbf{list}_\alpha$, and $\mathsf{append}:(\mathbf{list}_\alpha, \mathbf{list}_\alpha) \to \mathbf{list}_\alpha$. Then, all the results of a computation of type $F\alpha$ can be extracted into a returned value of type $F\mathbf{list}_\alpha$ using the handler

$$\begin{aligned}&\{\mathsf{fail} \mapsto \mathsf{return\,nil}\ ,\\ &\ \mathsf{pick}(\varphi_1, \varphi_2) \mapsto \mathsf{let}\, x_1 : \mathbf{list}_\alpha \,\mathsf{be}\, \varphi_1 \,\mathsf{in\,let}\, x_2 : \mathbf{list}_\alpha \,\mathsf{be}\, \varphi_2 \,\mathsf{in\,return\,append}(x_1, x_2)\}\ .\end{aligned}$$

We can similarly devise a handler that returns the first solution, or one that prints out a solution and asks the user whether to continue the search or not.

6.5 Handlers with Parameter Passing

Sometimes, we wish to handle different instances of the same operation differently, for example suppressing output after a certain number of characters. Although we handle operations in a fixed way, we can use handlers on a function type $\sigma \to \chi$ to simulate handlers on χ that pass around a parameter of type σ.

Instead of

$$(\boldsymbol{\varphi}_p : \boldsymbol{\chi}; \boldsymbol{x}_p : \boldsymbol{\sigma}).\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \mapsto \lambda x : \sigma . w_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathrm{eff}}} : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to (\sigma \to \chi) \ \textbf{handler}\ .$$

where all the occurrences of $\varphi_i(\boldsymbol{v})$ are applied to some $v : \sigma$, the changed parameter, we write

$$(\boldsymbol{\varphi}_p : \boldsymbol{\chi}; \boldsymbol{x}_p : \boldsymbol{\sigma}).\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \,@\, x : \sigma \mapsto w'_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathrm{eff}}} : (\boldsymbol{\chi}; \boldsymbol{\sigma}) \to \chi @ \sigma \ \textbf{handler}\ ,$$

where w'_{op} results from substituting $\varphi_i(\boldsymbol{v})@v$ for $\varphi_i(\boldsymbol{v})v$ in w_{op}. We also write

$$\mathsf{try}\, t \,\mathsf{with}\, H(\boldsymbol{t}; \boldsymbol{v})@v \,\mathsf{as}\, y : \sigma' \,@\, x : \sigma \,\mathsf{in}\, t'$$

instead of

$$(\mathsf{try}\, t \,\mathsf{with}\, H(\boldsymbol{t}; \boldsymbol{v}) \,\mathsf{as}\, y : \sigma' \,\mathsf{in}\, \lambda x : \sigma . t')v\ .$$

We could similarly simulate mutually defined handlers by handlers on product types, but we know no interesting examples of their use.

6.6 Timeout

When the evaluation of a computation takes too long, we may want to abort it and provide a predefined result instead, a behaviour called *timeout.*

We represent time by: a base signature with a base type **int** of integers, appropriate function symbols and a relation symbol $> : (\mathbf{int}, \mathbf{int})$, with the evident interpretation; an effect signature consisting of $\mathsf{delay} : 1$, to represent the passage of some fixed amount of time, with the empty effect theory. Then timeout can be described by a handler which passes around a parameter $T : \mathbf{int}$ representing how long we are willing to wait before we abort the evaluation and return φ_p.

$$(\varphi_p : X).\{\mathsf{delay}(\varphi) \,@\, T : \mathbf{int} \mapsto \mathsf{delay}(\mathsf{if}\, T > 0 \,\mathsf{then}\, \varphi@(T-1) \,\mathsf{else}\, \varphi_p)\}$$

Note that the handling term is wrapped in delay in order to preserve the time spent during the evaluation of the handled computation.

6.7 Input Redirection

With parameter passing, we can implement the redirection `proc < infile`, which feeds the contents of `infile` to the standard input of `proc`. We take the base signature, etc., of Section 6.2, extended by the base type $\mathbf{list_{char}}$, etc., of Section 6.4. Then a handler $H_{\mathrm{in}} : () \to X@\mathbf{list_{char}}$ **handler** to pass a string to a process is defined by

$$\{\mathsf{in}(\varphi) \,@\, \ell : \mathbf{list_{char}} \mapsto \mathsf{if}\, \ell = \mathsf{nil} \;\, \mathsf{then} \;\, \mathsf{in}(a.\varphi(a)@\mathsf{nil}) \;\, \mathsf{else} \;\, \varphi(\mathsf{head}(\ell))@\mathsf{tail}(\ell)\} \;.$$

Unfortunately we do not see how to implement the pipe $t_1 \mid t_2$: the difficulty is very much like that with the CCS parallel combinator.

6.8 Rollback

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

We take the base and the effect signatures for exceptions as in Example 1 and state as in Example 3, and the effect theory for state, together with the equation $\mathsf{update}_{l,d}(\mathsf{raise}_e) = \mathsf{raise}_e$ for each exception e [10]. The standard exception handler, extended to state, is no longer correct. Instead, working in an extended language as described above, we use a parametrically uniform handler $H_{\mathrm{rollback}} : () \to X@\,U(\mathbf{exc} \to X)$ **handler** with parameter a thunked function to revert modified locations. It is defined, omitting some type declarations, by

$$\begin{aligned}
&\{\mathsf{update}_{l,d}(\varphi) \,@\, f : U(\mathbf{exc} \to X) \mapsto \\
&\quad \mathsf{lookup}_l(d'.\mathsf{update}_{l,d}(\varphi@\,\mathsf{thunk}(\lambda e.\, \mathsf{let}\, x \,\mathsf{be}\, (\mathsf{force}\, f)e \,\mathsf{in}\, \mathsf{update}_{l,d'}(\mathsf{return}\, x)))) \;, \\
&\quad \mathsf{lookup}_l(\varphi) \,@\, f : U(\mathbf{exc} \to X) \mapsto \mathsf{lookup}_l(d.\varphi(d)@f) \;, \\
&\quad \mathsf{raise}_e \,@\, f : U(\mathbf{exc} \to X) \mapsto (\mathsf{force}\, f)e\} \;,
\end{aligned}$$

and is used on $t : F\sigma$ by $\mathsf{handle}\, t \,\mathsf{with}\, H_{\mathrm{rollback}}@t_0$ for some $t_0 : \mathbf{exc} \to F\sigma$.

We can also give a variant of rollback that passes around a list of changes to the memory, committed only after the computation has returned a value.

7 Logic

We sketch an adaptation of the logic for algebraic effects of [22] to account for handlers. This is relatively straightforward as the notions needed to interpret the handling construct are embodied in the computation induction (CI) and free algebra principles of [22]: the latter allows the ad-hoc construction of models and guarantees the existence of the required unique homomorphism.

We add handler types and terms to the language of the logic and state that handling is a homomorphic extension by:

$$\Gamma \vdash \mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\,H(\boldsymbol{t}_p;\boldsymbol{v}_p)\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t = t[v/x]\ ,$$
$$\Gamma \vdash \mathsf{try}\,\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.t_i)_i\,\mathsf{with}\,H(\boldsymbol{t}_p;\boldsymbol{v}_p)\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t = w'_{\mathsf{op}}\ ,$$

where, if $\Delta(H) = (\boldsymbol{\varphi}_p\!:\!\boldsymbol{\chi}\,;\,\boldsymbol{x}_p\!:\!\boldsymbol{\sigma}).\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{\varphi}) \mapsto w_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathrm{eff}}}$, then w'_{op} is obtained by substituting $\boldsymbol{t}_p$ for $\boldsymbol{\varphi}_p()$ and $\mathsf{try}\,t_i[\boldsymbol{v}_i/\boldsymbol{x}_i]\,\mathsf{with}\,H(\boldsymbol{t}_p;\boldsymbol{v}_p)\,\mathsf{as}\,x\!:\!\sigma\,\mathsf{in}\,t$ for $\varphi_i(\boldsymbol{v}_i)$ in $w_{\mathsf{op}}[\boldsymbol{v}_p/\boldsymbol{x}_p, \boldsymbol{v}/\boldsymbol{x}]$; CI yields uniqueness. These two equations generalise the first two 'handle-sequencing' equations of [12].

The fourth 'associativity' equation has no valid generalisation of the form

$$\begin{aligned}&\mathsf{try}\,(\mathsf{try}\,t_1\,\mathsf{with}\,m_2\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,t_2)\,\mathsf{with}\,m_3\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t_3\\ &\qquad\qquad = \mathsf{try}\,t_1\,\mathsf{with}\,m'_3\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,(\mathsf{try}\,t_2\,\mathsf{with}\,m_3\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t_3)\ ,\end{aligned}$$

for some m'_3, given the t_i and the m_j (m ranges over *model expressions* $H(\boldsymbol{t};\boldsymbol{v})$): such an m'_3 may not exist, and there may even be no possible model for it to denote. There are generalisations of the third and fourth equations provable by CI, subject to conditions involving the model expressions.

Still, the associativity of exception handlers is expressible, and provable by CI. We then have $m_1 = H_{\mathrm{exc}}(t)$ for some $t\!:\!\mathbf{exc} \to F\sigma_2$, and we set

$$m_3 =_{\mathrm{def}} H_{\mathrm{exc}}(\lambda e\!:\!\mathbf{exc}.\,\mathsf{try}\,te\,\mathsf{with}\,m_2\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t_3)\ .$$

(Note that, although handlers cannot contain handler constructs, it is possible to achieve something of that effect through the use of parameters, as we do here.) It may be, more generally, that unconditional associativity is provable for limited classes of handlers, such as the uniform ones. Further equations for exception-handling (also provable by CI) are given in [2]; it remains to consider their possible general forms.

8 Recursion

We sketch how to adapt the above ideas to deal with recursion. Base signatures are as before; for their interpretations we use ω-cpos and continuous functions, still, however, interpreting arity types by countable sets, equipped with the trivial order (with some additional effort this can be generalised to ω-cpos countable in the categorical sense). Effect syntax is as before, except that we allow conditional *inequations* $Z;\Gamma \vdash T_1 \leq T_2$ (ψ) and assume there is always a constant Ω and the inequation $\Omega \leq z$. We again obtain a category of models, now using

ω-cpos (necessarily with a least element) as carriers and continuous functions (necessarily strict) as homomorphisms; free models exist as before.

The handler and computation syntax is also as before except that we add recursion terms $\mu\varphi{:}\chi.w$ and $\mu y{:}\underline{\tau}.t$ (and so also computation variables y) with the usual least fixed-point interpretation. Correct handlers cannot redefine Ω because of the inequation $\Omega \leq z$. The adaptation of the logic of effects to allow recursion in [22] further adapts to handlers, analogously to the above; in this regard one notes that inequations are admissible and therefore one may still use computation induction to prove associativity and so on.

9 Conclusions

Some immediate questions stem from the current work. The most important is how to simultaneously handle two computations to describe parallel operators, e.g., that of CCS or the Unix pipe combinator; that would bring parallelism within the ambit of the algebraic theory of effects. More routinely, perhaps, the logic should be worked out more fully, the work done on combinations of effects in [10] should be extended to combinations of handlers, and there should be a general operational semantics [18] including that of Benton and Kennedy in [2].

The separation between the languages for handlers and computations is essential in the development of this paper. A possible alternative is to give a single language and a mechanism limiting well-typed handlers to correct ones. This might be done by means of a suitable type-theory.

It is interesting to compare our approach to that taken in Haskell [16], where a monad is given by a type with unit and binding maps. The type-checker only checks the signature of the maps, but not the monadic laws they should satisfy. Still, the only way to use effects in Haskell is through the use of the built-in monads, and their laws were checked by their designers. Building on this similarity, one can imagine extending Haskell in two ways: enriching the built-in effects with operations and handlers; and giving programmers a means to write their own handlers which could be used to program in an extension of the monadic style.

A given handler may or may not be computationally feasible for a given effect and so there is a question as to which are. We may expect uniform, or parametrically uniform, handlers to be feasible, as they cannot use the properties of a specific data-type and so, one may imagine, cannot be as contrived. In this connection, note too that a single monad or algebraic theory may model distinct effects. For example, the complexity monad $\mathbb{N} \times -$ may be used to model either space or time.

Lastly, one advantage of Benton and Kennedy's handling construct is the elegant programming style it introduces. We gave various examples of our more general construct above; some used parameter-passing, but none, unfortunately, used mutually defined handlers. We hope our new programming construct proves useful, and we look forward to feedback from the programming community.

Acknowledgments. The authors thank Andrej Bauer, Andrzej Filinski, Paul Levy, John Power, Mojca Pretnar, and Alex Simpson for their insightful comments and support.

References

1. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 42–122. Springer, Heidelberg (2002)
2. Benton, N., Kennedy, A.: Exceptional syntax. Journal of Functional Programming 11(4), 395–410 (2001)
3. Clocksin, W.F., Mellish, C.: Programming in Prolog, 3rd edn. Springer, Heidelberg (1987)
4. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, London (2000)
5. Filinski, A.: Representing layered monads. In: 26th Symposium on Principles of Programming Languages, pp. 175–188 (1999)
6. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: PLDI, pp. 237–247 (1993)
7. Grätzer, G.A.: Universal Algebra, 2nd edn. Springer, Heidelberg (1979)
8. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
9. Hyland, M., Levy, P.B., Plotkin, G.D., Power, A.J.: Combining algebraic effects with continuations. Theoretical Computer Science 375(1-3), 20–40 (2007)
10. Hyland, M., Plotkin, G.D., Power, A.J.: Combining effects: Sum and tensor. Theoretical Computer Science 357(1-3), 70–99 (2006)
11. Levy, P.B.: Call-by-push-value: Decomposing call-by-value and call-by-name. Higher-Order and Symbolic Computation 19(4), 377–414 (2006)
12. Levy, P.B.: Monads and adjunctions for global exceptions. Electronic Notes in Theoretical Computer Science 158, 261–287 (2006)
13. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)
14. Moggi, E.: Computational lambda-calculus and monads. In: 4th Symposium on Logic in Computer Science, pp. 14–23 (1989)
15. Moggi, E.: Notions of computation and monads. Information And Computation 93(1), 55–92 (1991)
16. Peyton Jones, S.L.: Haskell 98. Journal of Functional Programming 13(1), 255 (2003)
17. Plotkin, G.D.: Some varieties of equational logic. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 150–156. Springer, Heidelberg (2006)
18. Plotkin, G.D., Power, A.J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 1–24. Springer, Heidelberg (2001)
19. Plotkin, G.D., Power, A.J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002)
20. Plotkin, G.D., Power, A.J.: Algebraic operations and generic effects. Applied Categorical Structures 11(1), 69–94 (2003)
21. Plotkin, G.D., Power, A.J.: Computational effects and operations: An overview. Electronic Notes in Theoretical Computer Science 73, 149–163 (2004)
22. Plotkin, G.D., Pretnar, M.: A logic for algebraic effects. In: 23rd Symposium on Logic in Computer Science, pp. 118–129 (2008)
23. Power, A.J.: Countable Lawvere theories and computational effects. Electronic Notes in Theoretical Computer Science 161, 59–71 (2006)