

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 [1–3]. The computations that return values from a set X are represented 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 [4–6]; the operations give rise to the effects at hand. All of the effects mentioned above are algebraic, with the notable exception of continuations [7], which have to be treated differently (see [8] for initial ideas).

In the algebraic approach the arguments of an operation represent possible computations after an occurrence of an effect. For example, using a binary choice operation `or:2`, a nondeterministically chosen boolean is represented by the term `or(return true, return false):Fbool`, where $F\sigma$ stands for the type of computations that return values of type σ . The equations of the theory, for example the ones stating that `or` is a semi-lattice operation, generate the free-model functor, which is exactly the monad proposed by Moggi to model the corresponding effect [9] (modulo the forgetful functor) and which is used to interpret the type $F\sigma$. The operations are then interpreted by the model structure. When viewed as a family of functions parametric in X , e.g., $\text{or}_X: TX^2 \rightarrow TX$, one obtains a so-called

^{*} Supported by EPSRC grant GR/586371/01 and a Royal Society-Wolfson Award Fellowship.

^{**} Supported by EPSRC grant GR/586371/01.

algebraic operation; such families are characterised by a certain naturality condition [5].

Although this gives a way of constructing, combining [10], and reasoning [11] about algebraic effects, it does not account for their handling, as exception handlers, a well-known programming concept, fail to be algebraic operations [5]. 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 destructors* as they depend on the effects already created. Filinski’s reflection and reification [12] are closely related general concepts.

This paper introduces a handling construction for arbitrary algebraic effects. The central new idea is that, semantically, handling a computation amounts to composing it with 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. The principal example is exception handling, particularly the exception-handling construct of Benton and Kennedy [13], which our new construct generalises. It also includes many other, previously unrelated, concepts. For example, stream redirection of shell processes, renaming and hiding in CCS [14], timeout, and rollback can all be seen as instances of such handlers.

In Section 2 we explain the idea of using homomorphisms for the semantics of handlers via an informal discussion of exception handlers. In the following Sections 3, 4 and 5 we develop a formal calculus in the call-by-push-value framework [15, 11]. Section 3, describes (base) values and the algebraic theory of effects. A natural need for two languages arises: one to describe handlers, given in Section 4, and one where they are used to handle computations, given in Section 5. The second parts of these sections give the relevant denotational semantics; readers may wish to omit these and continue with Section 6, where we give examples.

We outline a version of a logic for algebraic effects [11] 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). Finally, we discuss some open questions and possible future work in Section 9.

2 Exception handlers

We start our study with exception handlers both because they are an established concept [13, 16] 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 represented by elements, γ , of $TX =_{\text{def}} X + E$; the unit of the monad is $\eta = x \mapsto \text{inl}(x)$. Algebraically, one may take a nullary operation, i.e., a constant,

$\text{raise}_e : 0$ for each $e \in E$ and no equations, and then FX has a carrier TX with raise_e interpreted as $\text{inr}(e)$.

An exception handler

$$\gamma \text{ handle } \{e \mapsto x_e\}_{e \in E}$$

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

$$\begin{aligned} \eta(x) \text{ handle } \{e \mapsto x_e\}_{e \in E} &= \text{inl}(x) , \\ \text{raise}_e \text{ handle } \{e \mapsto x_e\}_{e \in E} &= x_e . \end{aligned}$$

From an algebraic point of view, the x_e provide a model for the theory of exceptions on $X + E$, interpreting each operation raise_e by x_e . If we write $X + E$ for the free model and $\overline{X + E}$ for the new model on the same carrier set, we see from the above two equations that

$$h(\gamma) =_{\text{def}} \gamma \text{ handle } \{e \mapsto x_e\}_{e \in E}$$

is the unique homomorphism $h: X + E \rightarrow \overline{X + E}$ extending $\text{inl}: X \rightarrow \overline{X + E}$ along the unit η .

Benton and Kennedy [13] generalised the handling construct to one

$$\text{try } x \leftarrow \gamma \text{ in } g(x) \text{ unless } \{e \mapsto y_e\}_{e \in E} ,$$

where exceptions e may be handled by computations y_e of any given type M (here a model of the theory); returned values are “handled” with a map $g: X \rightarrow M$. (This construct is actually a bit more general than in [13] 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:

$$\begin{aligned} \text{try } x \leftarrow \eta(x) \text{ in } g(x) \text{ unless } \{e \mapsto y_e\}_{e \in E} &= g(x) , \\ \text{try } x \leftarrow \text{raise}_e \text{ in } g(x) \text{ unless } \{e \mapsto y_e\}_{e \in E} &= y_e . \end{aligned}$$

As they remarked, 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 y_e , and the handling construct corresponds to the homomorphism h induced by g , that is the unique homomorphism $h: X + E \rightarrow \overline{M}$ extending g along η . 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 an algebraic theory on a set X is to give a map $f_{\text{op}}: X^n \rightarrow X$ for each operation $\text{op}: n$, 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 were simply replaced by handling computations, operations are recursively replaced by handling functions on computations.

3 Values and effects

3.1 Base signature and interpretation

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

Base terms v are built from variables and function symbols, while *base formulas* φ are built from equations between base terms, relation symbols applied to base terms, logical connectives, and quantifiers over base types. In a context Γ of variables bound to base types, we type base terms as $\Gamma \vdash v: \beta$ and base formulas as $\Gamma \vdash \varphi: \mathbf{form}$.

An *interpretation of the base signature* is given by: a set $\llbracket \beta \rrbracket$ for each base type β , countable when β is an arity type; a map $\llbracket f \rrbracket: \llbracket \beta \rrbracket \rightarrow \llbracket \beta \rrbracket$ for each function symbol $f: (\beta) \rightarrow \beta$; and a subset $\llbracket R \rrbracket \subseteq \llbracket \beta \rrbracket$ for each relation symbol $R: (\beta)$, where $\llbracket \beta \rrbracket = \llbracket \beta_1 \rrbracket \times \dots \times \llbracket \beta_n \rrbracket$. Terms $\Gamma \vdash v: \beta$ and formulas $\Gamma \vdash \varphi: \mathbf{form}$ are interpreted by maps $\llbracket v \rrbracket: \llbracket \Gamma \rrbracket \rightarrow \llbracket \beta \rrbracket$ and subsets $\llbracket \varphi \rrbracket \subseteq \llbracket \Gamma \rrbracket$ as usual [17].

3.2 Effect theory

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 [5]. We present a more general notation to do this, at least in some cases. We assume a given base signature and interpretation.

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 $\text{update}_{l,d}: 1$ for each location l and datum d , we take a single operation symbol $\text{update}: \mathbf{loc}, \mathbf{dat}; 1$ that takes parameters $l: \mathbf{loc}$ and $d: \mathbf{dat}$, which give the memory location to be updated and the datum to be stored there.

To avoid operation symbols with infinite arity, we allow each argument of an operation to be dependent on the outcome of an effect, which is a value of an arity type. For example, the argument of an operation $\text{lookup}: \mathbf{loc}; \mathbf{dat}$ is dependent on the datum $d: \mathbf{dat}$, stored in the memory location $l: \mathbf{loc}$, supplied to it as a parameter.

Thus, an *effect signature* Σ_{eff} consists of *operation symbols* $\text{op}: \beta; \alpha_1, \dots, \alpha_n$, where β is a list of the *parameter* base types, and $\alpha_1, \dots, \alpha_n$ are lists of *argument* arity types. We omit the semicolon when β is empty, and we write n instead of $\alpha_1, \dots, \alpha_n$ when all the α_i are empty. *Effect terms* T , which serve as templates for computations, are given by the following grammar:

$$T ::= w(\mathbf{v}) \mid \text{op}_{\mathbf{v}}(\mathbf{x}_i: \alpha_i.T_i)_i,$$

where w ranges over *effect variables*, and $\text{op}_{\mathbf{v}}(\mathbf{x}_i: \alpha_i.T_i)_i$ is an abbreviation for $\text{op}_{\mathbf{v}}(\mathbf{x}_1: \alpha_1.T_1, \dots, \mathbf{x}_n: \alpha_n.T_n)$.

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

$$\frac{\Gamma \vdash \mathbf{v}: \boldsymbol{\alpha}}{\Gamma; \Delta \vdash w(\mathbf{v})} \quad (w: (\boldsymbol{\alpha}) \in \Delta)$$

$$\frac{\Gamma \vdash \mathbf{v}: \boldsymbol{\beta} \quad \Gamma, \mathbf{x}_i: \boldsymbol{\alpha}_i; \Delta \vdash T_i \quad (i = 1, \dots, n)}{\Gamma; \Delta \vdash \text{op}_{\mathbf{v}}(\mathbf{x}_i: \boldsymbol{\alpha}_i. T_i)_i} \quad (\text{op}: \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}).$$

Next, *conditional equations* have the form $\Gamma; \Delta \vdash T_1 = T_2$ (φ), assuming that $\Gamma; \Delta \vdash T_1$, $\Gamma; \Delta \vdash T_2$, and $\Gamma \vdash \varphi: \mathbf{form}$. Finally, a *conditional effect theory* \mathfrak{E} is a collection of such equations; it would be interesting to develop an equational logic for such theories [18].

Example 1. To describe a set E of exceptions, the base signature consists of a base type \mathbf{exc} and a constant function symbol $e: () \rightarrow \mathbf{exc}$ for each $e \in E$. We interpret \mathbf{exc} by E and functional symbols by their corresponding elements. The effect signature consists of an operation symbol $\text{raise}: \mathbf{exc}; 0$, while the effect theory is empty. Then, omitting empty parentheses, raise_e represents 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 $\text{or}: 2$, 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 \mathbf{loc} of memory locations, an arity type \mathbf{dat} of data, and appropriate function and relation symbols to represent the locations and data. We interpret \mathbf{loc} by a finite set L and \mathbf{dat} by a countable set D . The effect signature consists of operation symbols $\text{lookup}: \mathbf{loc}; \mathbf{dat}$ and $\text{update}: \mathbf{loc}, \mathbf{dat}; 1$, while the effect theory consists of seven conditional equations [9, 18]. As an example term, $\text{lookup}_l(d: \mathbf{dat}. \text{update}_{l', d}(w))$ represents the computation that copies d from l to l' and then proceeds as w .

Each effect theory \mathfrak{E} gives rise to a standard (possibly infinitary) equational theory [19]. For each $\text{op}: \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}$ and $\mathbf{b} \in \llbracket \boldsymbol{\beta} \rrbracket$, we take an operation symbol $\text{op}_{\mathbf{b}}$ of countable arity $\sum_i \llbracket \boldsymbol{\alpha}_i \rrbracket$. Then each term $\Gamma; \Delta \vdash T$ and each $\mathbf{c} \in \llbracket \Gamma \rrbracket$ give rise to a term $\Delta' \vdash T_{\mathbf{c}}$, where Δ' consists of variables $w_{\mathbf{a}}$ for each $w: (\boldsymbol{\alpha}) \in \Delta$ and $\mathbf{a} \in \llbracket \boldsymbol{\alpha} \rrbracket$. The equations of the theory are $\Delta' \vdash T_{\mathbf{c}} = T_{\mathbf{c}'}$ for any equation $\Gamma; \Delta \vdash T = T'$ (φ) in \mathfrak{E} and any $\mathbf{c} \in \llbracket \varphi \rrbracket$.

A *model of the effect theory* is a set M together with a family of maps $\{\text{op}_M: \llbracket \boldsymbol{\beta} \rrbracket \times \prod_i M^{\llbracket \boldsymbol{\alpha}_i \rrbracket} \rightarrow M\}_{\text{op}: \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}}$, such that the corresponding maps $\text{op}_M(\mathbf{b}, -)$, where $\mathbf{b} \in \llbracket \boldsymbol{\beta} \rrbracket$, satisfy the equations of the induced infinitary effect theory. A *homomorphism* between models M and N is a map $f: M \rightarrow N$ such that $\text{op}_N \circ (\mathbf{id}_{\llbracket \boldsymbol{\beta} \rrbracket} \times \prod_i f^{\llbracket \boldsymbol{\alpha}_i \rrbracket}) = f \circ \text{op}_M$ holds.

Models and homomorphisms form a category $\text{Mod}_{\mathfrak{E}}$, equipped with the forgetful functor $U: \text{Mod}_{\mathfrak{E}} \rightarrow \text{Set}$, which maps a model to its underlying set and a

homomorphism to its underlying map. This functor has a left adjoint F , which constructs the free model FA on a set of generators A . The set UFA represents the set of computations that return values in A , and the monad UF corresponds [9] to the monad proposed by Moggi to model the corresponding effect [2]. 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 [20].

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 difficult: 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.

Instead of equipping a single calculus with a mechanism to verify that handlers are correct, we avoid this complex interdependence between well-formedness and correctness by providing two calculi. One, given in this section, enables the language designer to specify handlers; one, 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.

Handlers are described by *handler types* χ and *handler terms* h , given by the following grammar:

$$\begin{aligned} \chi ::= & X \mid F\sigma \mid \mathbf{1} \mid \chi_1 \times \chi_2 \mid \sigma \rightarrow \chi, \\ h ::= & z(\mathbf{v}) \mid \text{op}_{\mathbf{v}}(\mathbf{x}_i : \alpha_i . h_i)_i \mid \text{if } \varphi \text{ then } h_1 \text{ else } h_2 \mid \text{return } u \mid \text{let } x \text{ be } h \text{ in } h' \mid \\ & \star \mid \langle h_1, h_2 \rangle \mid \text{fst } h \mid \text{snd } h \mid \lambda x : \sigma . h \mid hu, \end{aligned}$$

where X ranges over *type variables*, σ ranges over *value types*, z ranges over *handler variables*, u ranges over *value terms*, and φ in the conditional statement ranges over quantifier-free formulas. For the sake of simplicity, only the most basic programming constructs are present, and the only value types and terms are the one stemming from the base signature.

Handler terms h are typed with handler types χ in a context Γ of variables, bound to value types, and a context Z of handler variables $z : (\alpha) \rightarrow \chi$, where we write $z : \chi$ if α is empty, according to the following rules:

$$\frac{\Gamma \vdash \mathbf{v} : \alpha}{\Gamma; Z \vdash z(\mathbf{v}) : \chi} \quad (z : (\alpha) \rightarrow \chi \in Z)$$

$$\frac{\Gamma \vdash \mathbf{v} : \beta \quad \Gamma, \mathbf{x}_i : \alpha_i; Z \vdash h_i : \chi \quad (i = 1, \dots, n)}{\Gamma; Z \vdash \text{op}_{\mathbf{v}}(\mathbf{x}_i : \alpha_i . h_i)_i : \chi} \quad (\text{op} : \beta; \alpha_1, \dots, \alpha_n \in \Sigma_{\text{eff}})$$

$$\frac{\Gamma \vdash u : \sigma}{\Gamma; Z \vdash \text{return } u : F\sigma} \quad \frac{\Gamma; Z \vdash h : F\sigma \quad \Gamma, x : \sigma; Z \vdash h' : \chi}{\Gamma; Z \vdash \text{let } x \text{ be } h \text{ in } h' : \chi},$$

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

For greater generality, handlers are parametric in two ways: their type contains type variables, and they are dependent on parameters \mathbf{x}_p and \mathbf{z}_p , supplied through the handling construct. A handler is given by a handling term for each operation, dependent on its parameters \mathbf{x} and arguments \mathbf{z} , and is typed by

$$\frac{\mathbf{x}_p : \sigma, \mathbf{x} : \beta; \mathbf{z}_p : \chi, (z_i : (\alpha_i) \rightarrow \chi)_{i=1}^n \vdash h_{\text{op}} : \chi \quad (\text{op} : \beta; \alpha_1, \dots, \alpha_n \in \Sigma_{\text{eff}})}{\vdash (\mathbf{x}_p : \sigma; \mathbf{z}_p : \chi). \{ \text{op}_{\mathbf{x}}(\mathbf{z}) \mapsto h_{\text{op}} \}_{\text{op} \in \Sigma_{\text{eff}}} : (\sigma; \chi) \rightarrow \chi \text{ handler}}.$$

When $\text{op}_{\mathbf{x}}(\mathbf{z}) \mapsto h_{\text{op}}$ is omitted, we assume that $h_{\text{op}} = \text{op}_{\mathbf{x}}(\mathbf{x}_i : \alpha_i.z_i(\mathbf{x}_i))_i$, so that op is not handled.

4.1 Semantics

For each assignment of models $\llbracket X \rrbracket$ to type variables X , handler types χ are interpreted by models $\llbracket \chi \rrbracket$, given by

$$\begin{aligned} \llbracket F\sigma \rrbracket &= F\llbracket \sigma \rrbracket & \llbracket \mathbf{1} \rrbracket &= \mathbf{1} \\ \llbracket \chi_1 \times \chi_2 \rrbracket &= \llbracket \chi_1 \rrbracket \times \llbracket \chi_2 \rrbracket & \llbracket \sigma \rightarrow \chi \rrbracket &= \llbracket \chi \rrbracket^{\llbracket \sigma \rrbracket}, \end{aligned}$$

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

Then, we interpret contexts $Z = z_1 : (\alpha_1) \rightarrow \chi_1, \dots, z_n : (\alpha_n) \rightarrow \chi_n$ by $\llbracket Z \rrbracket = U\llbracket \chi_1 \rrbracket^{\llbracket \alpha_1 \rrbracket} \times \dots \times U\llbracket \chi_n \rrbracket^{\llbracket \alpha_n \rrbracket}$ and handler terms $\Gamma; Z \vdash h : \chi$ by maps $\llbracket h \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket Z \rrbracket \rightarrow U\llbracket \chi \rrbracket$, defined inductively by

$$\begin{aligned} \llbracket \Gamma; Z \vdash z_i(\mathbf{v}) : \chi_i \rrbracket &= \mathbf{ev} \circ \langle \mathbf{pr}_{U\llbracket \chi_i \rrbracket^{\llbracket \alpha_i \rrbracket}}, \llbracket \mathbf{v} \rrbracket \rangle, \\ \llbracket \Gamma; Z \vdash \text{op}_{\mathbf{v}}(\mathbf{x}_i.h_i) : \chi \rrbracket &= \text{op}_{\llbracket \chi \rrbracket} \circ \langle \llbracket \mathbf{v} \rrbracket, \widehat{\llbracket h_1 \rrbracket} \rangle, \dots, \widehat{\llbracket h_n \rrbracket} \rangle, \\ \llbracket \Gamma; Z \vdash \text{return } u : F\sigma \rrbracket &= \eta_{\llbracket \sigma \rrbracket} \circ \llbracket u \rrbracket, \\ \llbracket \Gamma; Z \vdash \text{let } x \text{ be } h \text{ in } h' : \chi \rrbracket &= \llbracket h' \rrbracket^\dagger \circ \langle \mathbf{id}_\Gamma, \mathbf{id}_Z, \llbracket h \rrbracket \rangle, \end{aligned}$$

where $\widehat{f} : B \rightarrow C^A$ is the *transpose* of $f : A \times B \rightarrow C$ and $f^\dagger : A \times UFB \rightarrow UM$ is the *lifting* of $f : A \times B \rightarrow UM$, which is defined by $U\epsilon \circ UFF \circ \mathbf{st}_{A,B}$, where $\mathbf{st}_{A,B} : A \times UFB \rightarrow UF(A \times B)$ is the *strength* of the functor UF . The interpretations of conditionals, products, and functions are defined as usual [15].

A handler $(\mathbf{x}_p : \sigma; \mathbf{z}_p : \chi). \{ \text{op}_{\mathbf{x}}(\mathbf{z}) \mapsto h_{\text{op}} \}_{\text{op} \in \Sigma_{\text{eff}}} : (\sigma; \chi) \rightarrow \chi$ *handler* is *correct (with respect to \mathfrak{E})* if for all assignments of models $\llbracket X \rrbracket$ to type variables X , and for all parameters $\mathbf{a}_p \in \llbracket \sigma \rrbracket$ and $\mathbf{m}_p \in \llbracket \chi \rrbracket$, the family of maps

$$\{ \llbracket h_{\text{op}} \rrbracket \circ \langle \mathbf{a}_p, \mathbf{pr}_{\llbracket \beta \rrbracket}, \mathbf{m}_p, \mathbf{pr}_{\prod_i U\llbracket \chi \rrbracket^{\llbracket \alpha_i \rrbracket}} \rangle : \llbracket \beta \rrbracket \times \prod_i U\llbracket \chi \rrbracket^{\llbracket \alpha_i \rrbracket} \rightarrow U\llbracket \chi \rrbracket \}_{\text{op} \in \Sigma_{\text{eff}}}$$

defines a model of the effect theory \mathfrak{E} on $U\llbracket \chi \rrbracket$.

5 Computations

A *handler signature* Σ_{hand} consists of *handler symbols* H , each with a corresponding correct handler. Then, *computation types* $\underline{\tau}$ and *computation terms* t are given by the following grammar:

$$\begin{aligned} \underline{\tau} &::= F\sigma \mid \mathbf{1} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \sigma \rightarrow \underline{\tau}, \\ t &::= \text{op}_v(x_i : \alpha_i.t_i)_i \mid \text{if } \varphi \text{ then } t_1 \text{ else } t_2 \mid \text{return } u \mid \text{let } x \text{ be } t \text{ in } t' \mid \\ &\quad \text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t' \mid \star \mid \langle t_1, t_2 \rangle \mid \text{fst } t \mid \text{snd } t \mid \lambda x : \sigma.t \mid tu. \end{aligned}$$

One can see that computation types and terms mirror their handler counterparts, with the omission of type and handler variables, and the addition of the handling construct. When the extended handling construct is not necessary, we write `handle t with $H(\mathbf{u}; \mathbf{t})$` instead of `try t with $H(\mathbf{u}; \mathbf{t})$ as x in return x` , and when the handler signature consists of a single handler symbol H , we omit it and simply write `try t with $\mathbf{u}; \mathbf{t}$ as x in t'` .

We can extend both handlers and computations with other call-by-push-value constructs [15]. 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, we would further split the handler types and terms into value and computation ones.

Computation terms t are typed with computation types $\underline{\tau}$, according to rules similar to the ones for handling terms, while the handling construct for a handler $H : (\sigma; \chi) \rightarrow \chi$ **handler** $\in \Sigma_{\text{hand}}$ is typed by

$$\frac{\Gamma \vdash t : F\sigma \quad \Gamma \vdash \mathbf{u} : \sigma \quad \Gamma \vdash \mathbf{t} : \chi[\underline{\tau}/\mathbf{X}] \quad \Gamma, x : \sigma \vdash t' : \chi[\underline{\tau}/\mathbf{X}]}{\Gamma \vdash \text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t' : \chi[\underline{\tau}/\mathbf{X}]},$$

where $\chi[\underline{\tau}/\mathbf{X}]$ is the computation type obtained by replacing all the type variables \mathbf{X} in χ by computation types $\underline{\tau}$. A handler $H : (\sigma; \chi) \rightarrow \chi$ **handler** is *uniform*, if $\chi = X$, and *parametrically uniform*, if $\chi = \sigma \rightarrow X$.

5.1 Semantics

Computation types and terms are interpreted in the same way as their handler counterparts, while the handling construct is interpreted as follows. Each handler $H : (\sigma; \chi) \rightarrow \chi$ **handler** $\in \Sigma_{\text{hand}}$ is correct and so induces a model $\llbracket \chi[\underline{\tau}/\mathbf{X}] \text{ with } H(\mathbf{u}; \mathbf{t}) \rrbracket$ for any assignment of computation types $\underline{\tau}$ to type variables \mathbf{X} , any value terms $\Gamma \vdash \mathbf{u} : \sigma$, and any computation terms $\Gamma \vdash \mathbf{t} : \chi[\underline{\tau}/\mathbf{X}]$. From the universality of the free model $F[\sigma]$, we get the unique map $\llbracket t' \rrbracket$, homomorphic in the second argument, for which the diagram below commutes.

$$\begin{array}{ccc} \llbracket \Gamma \rrbracket \times UF[\sigma] & & \\ \uparrow & \searrow \llbracket t' \rrbracket & \\ \llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket & \xrightarrow{\llbracket t' \rrbracket} & U[\chi[\underline{\tau}/\mathbf{X}] \text{ with } H(\mathbf{u}; \mathbf{t})] \end{array}$$

Then, $\Gamma \vdash \text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t'$ is interpreted by

$$\llbracket t' \rrbracket \circ \langle \text{id}_{\llbracket \Gamma \rrbracket}, \llbracket t \rrbracket \rangle : \llbracket \Gamma \rrbracket \rightarrow U[\llbracket \chi[\underline{\tau}/\mathbf{X}] \text{ with } H(\mathbf{u}; \mathbf{t}) \rrbracket] = U[\llbracket \chi[\underline{\tau}/\mathbf{X}] \rrbracket] .$$

6 Examples

6.1 Exceptions

The standard uniform exception handler $H_{\text{exc}} : (\mathbf{exc} \rightarrow X) \rightarrow X$ **handler** is

$$(z : \mathbf{exc} \rightarrow X) . \{ \text{raise}_e() \mapsto ze \} .$$

Benton and Kennedy's construct $\text{try } x \Leftarrow t \text{ in } t' \text{ unless } \{e_1 \Rightarrow t_1 \mid \dots \mid e_n \Rightarrow t_n\}$ can then be written as $\text{try } t \text{ with } t_{\text{exc}} \text{ as } x \text{ in } t'$ for a suitable term $t_{\text{exc}} : \mathbf{exc} \rightarrow \underline{\tau}$.

Benton and Kennedy noted a few issues about the syntax of their construct when used for programming [13]. 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 $\text{try } x \Leftarrow t \text{ unless } \{e_1 \Rightarrow t_1 \mid \dots \mid e_n \Rightarrow t_n\}_i \text{ in } t'$, but then it is not obvious that x is bound in t' , but not in the handler. The syntax of our construct $\text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t'$ 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 and 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 (default one being `y`).

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

6.3 CCS renaming and hiding

In functional programming, processes are regarded as programs of the empty type $\mathbf{0}$. The subset of CCS processes [21], given by action prefix and sum, can be represented by: a base signature, consisting of a base type \mathbf{act} of actions and appropriate constants for actions, interpreted in the evident way; an effect signature, consisting of operation symbols $\mathbf{0} : \mathbf{0}$, $\mathbf{do} : \mathbf{act}; \mathbf{1}$, and $\mathbf{+} : \mathbf{2}$, with the obvious effect theory [11]. Then, process renaming $t[b/a]$ can be written as *handle t with $H_{\text{ren}}(a, b)$* using the handler $H_{\text{ren}} : (\mathbf{act}, \mathbf{act}) \rightarrow F\mathbf{0}$ **handler**, where, writing $a.z$ for $\mathbf{do}_a(z)$:

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

Hiding can be implemented in a similar way, but, and most unfortunately, it seems that parallel cannot be, as it is recursively defined on the structure of *two* processes and thus apparently requires a binary variant of handlers. Presently, we do not see a possible such generalisation, as primitive recursion is inherently defined on a single structure.

6.4 Explicit nondeterminism

The evaluation of a nondeterministic computation usually takes only one of all the possible paths. But in logic programming [22, 23], 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; the effect signature, consisting of operation symbols $\mathbf{fail} : \mathbf{0}$ and $\mathbf{pick} : \mathbf{2}$, with the effect theory consisting of the following equations stating that the operations form a monoid:

$$\begin{aligned} w \vdash \mathbf{pick}(w, \mathbf{fail}()) &= \mathbf{pick}(\mathbf{fail}(), w) = w , \\ w_1, w_2, w_3 \vdash \mathbf{pick}(w_1, \mathbf{pick}(w_2, w_3)) &= \mathbf{pick}(\mathbf{pick}(w_1, w_2), w_3) . \end{aligned}$$

The free-model monad maps a set to the set of all finite sequences of its elements, which is Haskell's nondeterminism monad [24].

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}_α , function symbols $\mathbf{nil} : () \rightarrow \mathbf{list}_\alpha$, $\mathbf{cons} : (\alpha, \mathbf{list}_\alpha) \rightarrow \mathbf{list}_\alpha$, $\mathbf{head} : (\mathbf{list}_\alpha) \rightarrow \alpha$, $\mathbf{tail} : (\mathbf{list}_\alpha) \rightarrow \mathbf{list}_\alpha$, and $\mathbf{append} : (\mathbf{list}_\alpha, \mathbf{list}_\alpha) \rightarrow \mathbf{list}_\alpha$. 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

$$\begin{aligned} \{ \mathbf{fail}() \mapsto \mathbf{return} \mathbf{nil}() , \\ \mathbf{pick}(z_1, z_2) \mapsto \mathbf{let } x_1 \text{ be } z_1 \text{ in let } x_2 \text{ be } z_2 \text{ in } \mathbf{return} \mathbf{append}(x_1, x_2) \} . \end{aligned}$$

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 $\sigma \rightarrow \chi$ to simulate handlers on χ that pass around a parameter of type σ .

Instead of

$$(\mathbf{x}_p : \sigma; \mathbf{z}_p : \chi). \{ \text{op}_{\mathbf{x}}(\mathbf{z}) \mapsto \lambda x : \sigma. h_{\text{op}} \}_{\text{op} \in \Sigma_{\text{eff}}} : (\sigma; \chi) \rightarrow (\sigma \rightarrow \chi) \text{ handler .}$$

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

$$(\mathbf{x}_p : \sigma; \mathbf{z}_p : \chi). \{ \text{op}_{\mathbf{x}}(\mathbf{z}) @ x \mapsto h'_{\text{op}} \}_{\text{op} \in \Sigma_{\text{eff}}} : (\sigma; \chi) \rightarrow \chi @ \sigma \text{ handler ,}$$

where h'_{op} results from substituting $z_i(\mathbf{v})v$ for $z_i(\mathbf{v})@v$ in h_{op} . We also write

$$\text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) @ v \text{ as } x @ y \text{ in } t'$$

instead of

$$(\text{try } t \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } \lambda y : \sigma. t')v .$$

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 **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 z_p .

$$(z_p : X). \{ \text{delay}(z) @ T \mapsto \text{delay}(\text{if } T > 0 \text{ then } z @ (T - 1) \text{ else } z_p) \} .$$

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 **list_{char}**, etc., of Section 6.4. Then a handler $H_{\text{in}} : X @ \mathbf{list}_{\text{char}}$ **handler** to pass a string to a process is given by:

$$\{ \text{in}(z) @ \ell \mapsto \text{if } \ell = \text{nil}() \text{ then in}(a.z(a) @ \text{nil}()) \text{ else } z(\text{head}(\ell)) @ \text{tail}(\ell) \} .$$

Unfortunately we do not see how to implement the pipe $t_1 | t_2$: 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 $\text{update}_{l,d}(\text{raise}_e) = \text{raise}_e$ for each exception e we deem unrecoverable [10]. In this case, the standard exception handler, extended to state, is not correct. Instead, we use a handler $H_{\text{rollback}} : X @ (\mathbf{exc} \rightarrow X)$ **handler**, which passes around a function that reverts the modified locations. Such a handler is given by

$$\begin{aligned} \{ & \text{update}_{l,d}(z) @ f \mapsto \\ & \quad \text{lookup}_l(d'.\text{update}_{l,d}(z @ (\lambda e : \mathbf{exc}. \text{let } x \text{ be } fe \text{ in } \text{update}_{l,d'}(x)))) , \\ & \text{lookup}_l(z) @ f \mapsto \text{lookup}_l(d.z(d) @ f) , \\ & \text{raise}_e() @ f \mapsto fe \} , \end{aligned}$$

and is used on $t : F\sigma$ by **handle** t with $H_{\text{rollback}} @ t_0$ for some $t_0 : \mathbf{exc} \rightarrow F\sigma$.

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

Since the notions needed to interpret the handling construct are present in all the interpretations of algebraic effects, it is relatively easy to adapt the logic for algebraic effects of [11] to account for handlers. (In fact, handlers are already present in the logic in a way, as the free algebra principle allows an ad-hoc construction of models and guarantees the existence of the required unique homomorphism.)

To incorporate handlers, we extend the language of the logic with handler types and terms, and state that the handling construct acts as a homomorphism by:

$$\begin{aligned} \Gamma \vdash \text{try return } u \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t &= t[u/x] , \\ \Gamma \vdash \text{try op}_v(\mathbf{x}_i.t_i)_i \text{ with } H(\mathbf{u}; \mathbf{t}) \text{ as } x \text{ in } t' &= h_{\text{op}'}[v/\mathbf{x}] , \end{aligned}$$

where h'_{op} is the computation term, obtained by taking the handling term h_{op} and substituting $\text{try } t_i[v_i/\mathbf{x}_i]$ with $H(\mathbf{u}; \mathbf{t})$ as x in t for $z_i(v_i)$. The principle of computation induction yields uniqueness.

This principle was used in [11] to derive associativity and other properties of let binding and we can obtain analogous equations for the exception-handling construct, instead of taking them as axioms [13, 16]. However, a general associativity of the handling construct would have the form

$$\begin{aligned} \text{try} (\text{try } t_1 \text{ with } H_1 \text{ as } x_1 \text{ in } t_2) \text{ with } H_2 \text{ as } x_2 \text{ in } t \\ = \text{try } t_1 \text{ with } H \text{ as } x_1 \text{ in} (\text{try } t_2 \text{ with } H_2 \text{ as } x_2 \text{ in } t) , \end{aligned}$$

but may not be expressible, as the model needed for H , the image of the model induced by H_1 under the homomorphism induced by t_2 , may not be definable.

Still, the associativity of exception handlers is not only expressible, but also derivable by induction, because, then, $H_1 = H_{\text{exc}}(f_1)$ and $H_2 = H_{\text{exc}}(f_2)$ for some $f_1 : \mathbf{exc} \rightarrow F\sigma_2$ and $f_2 : \mathbf{exc} \rightarrow \underline{\tau}$, and we can set

$$H =_{\text{def}} H_{\text{exc}}(\lambda e : \mathbf{exc}. \text{try } f_1 e \text{ with } H_2 \text{ as } x_2 \text{ in } t) .$$

8 Recursion

We sketch how to adapt the above ideas to deal with recursion. We work with ω -cpos and continuous functions. 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 countable ω -cpos, in the categorical sense). Effect syntax is as before, except that we allow conditional *inequations* $\Gamma; \Delta \vdash T_1 \leq T_2$ (φ) and assume there is always a constant Ω and the inequation $\Omega \leq w$. We again obtain a category of models, now using ω -cpos (necessarily with a least element) and continuous functions; free models exist as before.

Handler and computation syntax are also as before except that we add recursion terms $\mu x : \chi. h$ and $\mu x : \underline{\tau}. t$ (and so also computation variables) with the usual least fixed-point interpretation. Correct handlers cannot redefine Ω because of the inequation $\Omega \leq w$. The adaptation of the logic of effects to allow recursion in [11] further adapts to handlers, analogously to the above; in this regard one notes that equations 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 work done on combinations of effects [10] should be extended to combinations of handlers; one would also like a general operational semantics [4] including Benton and Kennedy's in [13].

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 [24], 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: one could enrich the built-in effects with operations and handlers, and one could give programmers a way to write their own handlers, which have no direct access to effects, but which could be used to program in an extension of the monadic style.

Monads, or algebraic theories, are used to model particular computational effects, and it is even possible for one monad to model distinct effects. For example the complexity monad $\mathbb{N} \times -$ may account for either space or time. A given handler may or may not be computationally feasible for a given effect and there is a question as to which are. We may expect 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.

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. Moggi, E.: Computational lambda-calculus and monads. In: 4th Symposium on Logic in Computer Science. (1989) 14–23
2. Moggi, E.: Notions of computation and monads. *Information And Computation* **93**(1) (1991) 55–92
3. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: International Summer School on Applied Semantics 2000. Volume 2395 of Lecture Notes in Computer Science. (2000) 42–122
4. Plotkin, G.D., Power, A.J.: Adequacy for algebraic effects. In: 4th International Conference on Foundations of Software Science and Computation Structures. Volume 2030 of Lecture Notes in Computer Science. (2001) 1–24
5. Plotkin, G.D., Power, A.J.: Algebraic operations and generic effects. *Applied Categorical Structures* **11**(1) (2003) 69–94
6. Plotkin, G.D., Power, A.J.: Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science* **73** (2004) 149–163
7. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Conference on Programming Language Design and Implementation. (1993) 237–247
8. Hyland, M., Levy, P.B., Plotkin, G.D., Power, A.J.: Combining algebraic effects with continuations. *Theoretical Computer Science* **375**(1-3) (2007) 20–40
9. Plotkin, G.D., Power, A.J.: 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. (2002) 342–356

10. Hyland, M., Plotkin, G.D., Power, A.J.: Combining effects: Sum and tensor. *Theoretical Computer Science* **357**(1-3) (2006) 70–99
11. Plotkin, G.D., Pretnar, M.: A logic for algebraic effects. In: 23rd Symposium on Logic in Computer Science. (2008) 118–129
12. Filinski, A.: Representing layered monads. In: 26th Symposium on Principles of Programming Languages. (1999) 175–188
13. Benton, N., Kennedy, A.: Exceptional syntax. *Journal of Functional Programming* **11**(4) (2001) 395–410
14. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. *Journal of the ACM* **32**(1) (1985) 137–161
15. Levy, P.B.: Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* **19**(4) (2006) 377–414
16. Levy, P.B.: Monads and adjunctions for global exceptions. *Electronic Notes in Theoretical Computer Science* **158** (2006) 261–287
17. Enderton, H.B.: *A Mathematical Introduction to Logic*. 2nd edn. Academic Press (2000)
18. Plotkin, G.D.: Some varieties of equational logic. In: *Essays Dedicated to Joseph A. Goguen*. Volume 4060 of *Lecture Notes in Computer Science*. (2006) 150–156
19. Grätzer, G.A.: *Universal algebra*. 2nd edn. Springer (1979)
20. Power, A.J.: Countable Lawvere theories and computational effects. *Electronic Notes in Theoretical Computer Science* **161** (2006) 59–71
21. Milner, R.: *A calculus of communicating systems*. Springer (1980)
22. Clocksin, W.F., Mellish, C.: *Programming in Prolog*. 3rd edn. Springer (1987)
23. Pfenning, F., Schürmann, C.: System description: Twelf – a meta-logical framework for deductive systems. In: 16th International Conference on Automated Deduction. Volume 1632 of *Lecture Notes in Computer Science*. (1999) 202–206
24. Jones, S.L.P.: Haskell 98. *Journal of Functional Programming* **13**(1) (2003) 0–255