Algebraic Effects and Effect Handlers for Idioms and Arrows

Sam Lindley
The University of Edinburgh

Abstract

Plotkin and Power's algebraic effects combined with Plotkin and Pretnar's effect handlers provide a foundation for modular programming with effects. We present a generalisation of algebraic effects and effect handlers to support other kinds of effectful computations corresponding to McBride and Paterson's idioms and Hughes' arrows.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language Constructs and Features*]; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

Keywords algebraic effects; effect handlers; idioms; arrows; monads; applicative functors; call-by-push-value

1. Introduction

In previous work [8] we advocated Plotkin and Power's algebraic effects [21, 22] and effect handlers [23] as a foundation for modular programming with effects. We introduced a statically typed effect handler calculus λ_{eff} along with a sound, terminating, small-step operational semantics, and used it as the basis for practical implementations of handlers in Haskell, ML, and Racket.

Our calculus λ_{eff} (and standard algebraic effects and handlers) provide a means for programming with monadic effects [18], supporting generic effectful computations that can be handled in multiple ways. In this work, we adapt λ_{eff} to accommodate other kinds of effectful computations corresponding to McBride and Paterson's idioms (also known as applicative functors) [17] and Hughes' arrows [7]. The resulting calculus, λ_{flow}, extends λ_{eff} with *flow effects*, which explicitly track dependencies between the result of an effectful operation and subsequent effectful computation, allowing us to encode idiom and arrow computations. Crucially, λ_{flow} supports effect handlers for idiom and arrow computations.

A key reason why arrow programs and idiom programs are of interest to functional programmers is that they admit a wider range of implementations than corresponding monadic programs. Every monad is an arrow and every monad is also an idiom. But there exist arrows that are not monads and idioms that are not monads. For instance, non-monadic arrows are often used in functional reactive programming, and non-monadic idioms in parser combinators, in both cases providing more space and time efficient implementations than monadic alternatives.

The λ_{flow} calculus combines the benefits of λ_{eff} (modular support for handling multiple effects) and our earlier work on the arrow calculus [13, 14] (providing a uniform foundation for programming with idioms, arrows, and monads with a single effect). It allows us to use the same syntax for idiom, arrow, and monad computations. Thus, we can write generic effectful combinators in λ_{flow} that can be used in idiom, arrow, and monad computations.

Just as λ_{eff} gave us a firm basis for building practical implementations of standard effect handlers, we hope that λ_{flow} can provide a firm basis for building practical implementations of idiom and arrow handlers, and in particular support generic effectful programs, combining idiom, arrow, and monad computations.

Our main contributions are as follows:

- We introduce flow effects as a means for distinguishing abstract idiom, arrow, and monad computations.
- We provide a uniform foundation for programming with idioms, arrows, and monads with multiple effects, generalising both λ_{eff} and the arrow calculus.
- An immediate consequence of our formulation is that the inclusions between abstract idiom and abstract arrow computations, and abstract arrow and abstraction monad computations of Lindley et al. [14] are strict. Abstract monad programs are strictly more expressive than abstract arrow programs, which are in turn strictly more expressive than abstract idiom programs.
- We introduce idiom and arrow handlers, as variations on standard monadic effect handlers.
- We illustrate the use of an extension of λ_{flow} for writing generic parser combinators.

The remainder of the paper is structured as follows. Section 2 introduces the key ideas of our approach by first characterising abstract effectful computations as computation trees, and then defining flow effects in terms of computation trees. Section 3 describes λ_{eff}, first focusing on abstract computations, and then on effect handlers. Section 4 presents flow effects and Core λ_{flow}, the fragment of λ_{flow} for expressing abstract idiom, arrow, and monad computations. Section 5 presents handlers in λ_{flow} for idiom, arrow, and monad computations. Section 6 presents a parser combinator implementation using an extension of λ_{flow}. Section 7 discusses related work. Section 8 discusses future work.

2. Effects as Computation Trees

2.1 What is an Effectful Computation?

Plotkin and Power [21, 22] introduced algebraic effects for modelling the semantics of effectful computations. They gave an abstract categorical treatment. We will be much more concrete, and after our initial example consider only free algebras.

WGP '14, August 31 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3042-8/14/08…$15.00.
http://dx.doi.org/10.1145/2633628.2633636

An algebraic effect is given by a signature of operations along with a set of equations on those operations. For example, we might define an algebraic effect for read only boolean state with the following signature:

$$\{\mathsf{get} : 1 \to \mathsf{Bool}\}$$

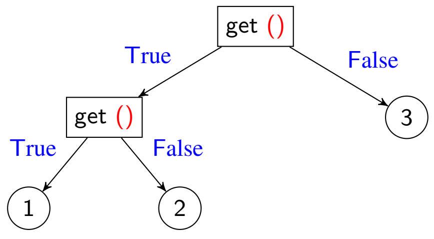
and equations:

$$\mathsf{get}_{()}(\mathsf{get}_{()}(p,q),r) = \mathsf{get}_{()}(p,r) = \mathsf{get}_{()}(p,\mathsf{get}_{()}(q,r))$$
$$\mathsf{get}_{()}(p,p) = p$$

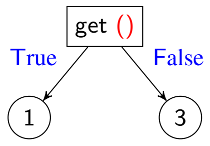
When specifying equations in the algebraic approach, operations are typically written in a continuation passing style (CPS), exposing their algebraic structure. Thus $\mathsf{get}_{()}(p,q)$ corresponds to a term that a functional programmer would typically write in direct style as **let** $x \leftarrow \mathsf{get}\ ()$ **in if** x **then** p **else** q. This continuation passing style corresponds directly to a view of algebraic computations as (possibly infinitely branching) trees, such that:

- nodes are labelled with operations;
- there is an edge labelled with each possible return value in the domain of the operation associated with a parent node;
- each such edge is connected to the corresponding continuation of the computation;
- leaves are labelled with final return values; and
- trees are quotiented modulo the equations.

For example, the CPS term $\mathsf{get}_{()}(\mathsf{get}_{()}(1,2),3)$ is given by the computation tree:



which by the first equation is equivalent to the CPS term $\mathsf{get}_{()}(1,3)$, whose computation tree is:



Following our previous work [8] on handlers for algebraic effects, we consider only algebraic effects for which there are no equations. We call these *abstract effects* and algebraic computations over them *abstract computations*. Thus an abstract computation is a plain unquotiented tree.

Abstract effects are closely related to *monads*, which Moggi successfully advocated [18] as a tool for modelling the semantics of effectful computation. Indeed, an abstract effect over an effect signature Σ is exactly the free monad construction over the functor generated by Σ [25].

2.2 Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous

In previous work [14], we analysed the relative expressiveness of abstract idiom, arrow, and monad computations, proving that abstract idiom computations are less expressive than abstract arrow computations, which in turn are less expressive than abstract monad computations. We also gave an informal characterisation of the differences in terms of *control flow* and *data flow*, which we will now develop into a crisp characterisation in terms of constraints on abstract computation trees.

	control flow	**data flow**
idiom	static	static
arrow	static	dynamic
monad	dynamic	dynamic

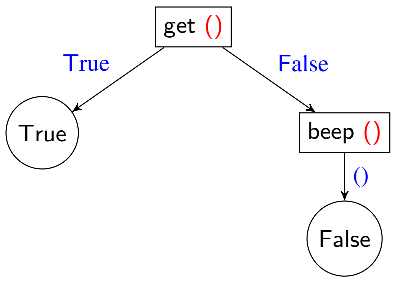
Table 1. Flow for Idioms, Arrows, and Monads

We say that data flow is dynamic if the value passed to an operation can depend on the results of prior operations. We say that control flow is dynamic if which operation to invoke (or whether to invoke an operation at all) can depend on the results of prior operations. The nature of data flow and control flow for idioms, arrows, and monads is given in Table 1. Idioms are entirely static. Arrows have static control flow but dynamic data flow. Monads are entirely dynamic.

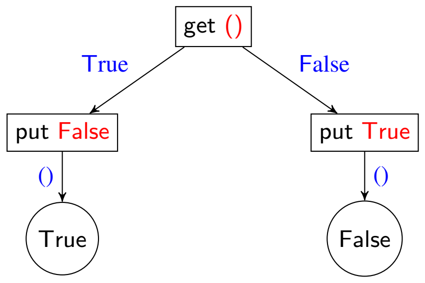
Now we consider how this characterisation relates to abstract computation trees. First let us define some effect signatures:

$$\mathsf{GB} = \{\mathsf{get}\ : 1 \to \mathsf{Bool}\quad \mathsf{beep} : 1 \to 1\ \} \qquad \mathsf{State}\ S = \{\mathsf{get} : 1 \to S\quad \mathsf{put} : S \to 1\}$$

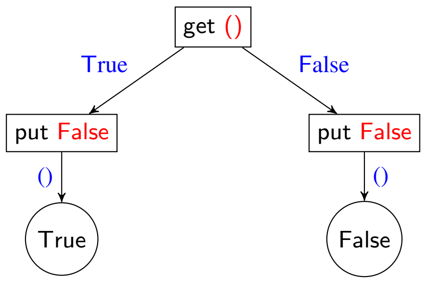
Monad trees are just abstract computation trees, as monads impose no restrictions on dependencies. For example, the CPS term, $\mathsf{noisey} = \mathsf{get}(\mathsf{True}, \mathsf{beep}(\mathsf{False}))$, over effect signature GB, represents monad tree:



Arrow trees are those monad trees for which control flow is static. Concretely, this means that the tree must be completely balanced, and at each level of the tree each node must be labelled with the same operation (though the parameters to the operations may differ). For example, the CPS term, $\mathsf{flip} = \mathsf{get}(\mathsf{put}_{\mathsf{False}}(\mathsf{True}), \mathsf{put}_{\mathsf{True}}(\mathsf{False}))$, over effect signature State Bool, represents the arrow tree:



Idiom trees are those arrow trees for which not only is the control flow static, but so is the data flow. Concretely, this means at each level of the tree the parameters of the operations on each node are identical. For example, the CPS term, $\mathsf{reset} = \mathsf{get}(\mathsf{put}_{\mathsf{False}}(\mathsf{True}), \mathsf{put}_{\mathsf{False}}(\mathsf{False}))$, over effect signature State Bool, represents the idiom tree:



Freedom Just as abstract monadic effects correspond exactly with the free monad construction over an effect signature, so abstract arrow effects correspond exactly with the free arrow construction over an effect signature, and abstract idiom effects correspond exactly with the free idiom construction over an effect signature.

Strange Computations An obvious omission from Table 1 is the case where control flow is dynamic, but data flow is static. We are not aware of a corresponding structure in the literature, and it seems rather strange to have dynamic control flow without dynamic data flow as well, so we will refer to such computations as *strange*. We can give a characterisation of strange computations in terms of abstract computation trees. *Strange trees* are those abstract computation trees for which control flow is dynamic, but data flow is static. Concretely, this means that the tree must be completely balanced, and at each level of the tree the parameters of the operations on each node must be identical.

Memory Flow From the structure of abstract computation trees, it is apparent that there is one other place that results may flow to: the leaves of the tree. We refer to this kind of flow as *memory*. We say that memory flow is dynamic if the values at the leaves depend on the results of prior computations and static if they do not. Idiom, arrow, and monad computations all have dynamic memory flow. One might conceive of explicitly distinguishing effectful computations that have static memory flow. The leaves of the corresponding computation trees are all the same; hence they must always have the same return value. This notion does not seem particularly useful, as we can always achieve the same behaviour via computations with unit return type paired up with the constant return value.

2.3 Flow Effects

In order to distinguish static and dynamic control and data flow, we will introduce special effect annotations which we call *flow effects*. The two flow effects are c, for dynamic control flow, and d, for dynamic data flow. We explain how they fit into λ_{flow} in Section 4.

3. An Effect Calculus

As a starting point for a calculus of effects and handlers for idioms, arrows, and monads, we take the λ_{eff}-calculus [8], which provides an effect type system and operational semantics for standard monadic algebraic effects and effect handlers, providing a foundation for generic effectful programming. In this section, we recapitulate the details of λ_{eff}.

3.1 Abstract Effects

In this subsection, we introduce Core λ_{eff}, the fragment of λ_{eff} that describes abstract effects in isolation from their handlers. This sublanguage allows us to write abstract computations over an effect signature. We deviate slightly from our previous presentation [8]. Apart from superficial differences in lexical syntax, we omit computation products and unit, as they are orthogonal to the current work, and we adopt direct-style (as opposed to CPS) operations as primitive.

The syntax of types is as follows:

$$\begin{array}{lrl}
\text{(values)} & A, B ::= & 1 \mid A_1 \times A_2 \\
 & \mid & 0 \mid A_1 + A_2 \\
 & \mid & \{C\}_E \\
\text{(computations)} & C ::= & [A] \mid A \to C \\
\text{(effect signatures)} & E ::= & \{\mathrm{op} : A \to B\} \uplus E \mid \emptyset \\
\text{(environments)} & \Gamma ::= & x_1 : A_1, \ldots, x_n : A_n
\end{array}$$

Following Levy's call-by-push-value [10], types are partitioned into value types (A, B) and computation types (C). The primary benefit we gain from a call-by-push-value approach is that it makes an explicit distinction between supplying an argument to a function and forcing a suspended computation. Effects are associated only with the latter action.

Value types (V, W) comprise unit (1), product ($A_1 \times A_2$), empty (0), sum ($A_1 + A_2$), and thunk types ($\{C\}_E$). In $\{C\}_E$, the computation type C is allowed to perform operations in the *effect signature* E. Computation types (C) comprise returners ($[A]$), which yields values of type A, and function types $A \to C$, from an argument of type A to a computation of type C. An effect signature comprises a collection of operation signatures $\{\mathrm{op}_i : A_i \to B_i\}_i$. Operations in a signature must have distinct names, but order is not important. Type environments (Γ) are standard.

Intuitively, one can think of a returner type $[A]$ as being inhabited by computation trees. The structure of such trees can be quite rich. For a start, given a node for operation $\mathrm{op} : A \to B$, it has n children, where n is the number of inhabitants of B. If we simply add a base type of natural numbers, then this means that our trees can be infinitely wide. More interestingly, an operation may take a suspended returner computation as a parameter, which yields a kind of "hyper tree" structure in which internal nodes may themselves contain further trees. Amongst other things, operations with computation parameters are useful for implementing a binary choice between two computations (see Section 6) and certain forms of aspect-oriented programming patterns [6, 12].

The syntax of terms is as follows:

$$\begin{array}{lrl}
\text{(values)} & V, W ::= & x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\} \\
\text{(computations)} & M, N ::= & \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case}_0(V) \\
 & \mid & \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V! \\
 & \mid & \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
 & \mid & \lambda x.M \mid M\ V \\
 & \mid & \mathrm{op}\ V
\end{array}$$

As with types, the terms are partitioned into value terms and computation terms. Value terms comprise variables (x), unit ($()$), pairs ((V_1, V_2)), injections ($\mathbf{inj}_i\, V$), and thunks ($\{M\}$). Value terms are inert, in that all computation takes place in computation terms. Thus, all of the value constructs apart from variables are introduction forms. Computation terms comprise elimination forms for pairs ($\mathbf{split}(V, x_1.x_2.M)$), the empty type ($\mathbf{case}_0(V)$), sum types ($\mathbf{case}(V, x_1.M_1, x_2.M_2)$), and thunks ($V!$) , introduction ($\mathbf{return}\ V$) and elimination ($\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N$) forms for returners, introduction ($\lambda x.M$) and elimination ($M\ V$) forms for functions, and operation applications ($\mathrm{op}\ V$).

The typing rules for Core λ_{eff} are given in Figure 1. The value judgement $\Gamma \vdash V : A$ asserts that value term V has type A in type environment Γ. The computation judgement $\Gamma \vdash_E M : C$ asserts that computation term M has type C with effects E in type environment Γ. The small-step operational semantics for Core λ_{eff} is given in Figure 2.

$\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V_1 : A_1 \quad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \textsc{Inj}_i\ \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \textsc{Thunk}\ \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : \{C\}_E}$$

$\boxed{\Gamma \vdash_E M : C}$

$$\textsc{Split}\ \frac{\Gamma \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C} \qquad \textsc{CaseZero}\ \frac{\Gamma \vdash V : 0}{\Gamma \vdash_E \mathbf{case}_0(V) : C}$$

$$\textsc{Case}\ \frac{\Gamma \vdash V : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash_E M_1 : C \quad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C} \qquad \textsc{Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash_E \mathbf{return}\, V : [A]}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_E M : [A] \quad \Gamma, x : A \vdash_E N : C}{\Gamma \vdash_E \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : C} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash_E M : C}{\Gamma \vdash_E \lambda x.M : A \to C} \qquad \textsc{App}\ \frac{\Gamma \vdash_E M : A \to C \quad \Gamma \vdash V : A}{\Gamma \vdash_E M\, V : C}$$

$$\textsc{Force}\ \frac{\Gamma \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C} \qquad \textsc{Op}\ \frac{(\mathrm{op} : A \to B) \in E \quad \Gamma \vdash V : A}{\Gamma \vdash_E \mathrm{op}\, V : [B]}$$

Figure 1. Typing Rules for Core λ_{eff}

$$\begin{array}{lrcl}
(\beta.\times) & \mathbf{split}((V_1, V_2), x_1.x_2.M_1) & \longrightarrow & M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \mathbf{case}(\mathbf{inj}_i\, V, x_1.M_1, x_2.M_2) & \longrightarrow & M_i[V/x_i] \\
(\beta.\{\}) & \{M\}! & \longrightarrow & M \\
\\
(\beta.[]) & \mathbf{let}\, x \leftarrow \mathbf{return}\, V\, \mathbf{in}\, M & \longrightarrow & M[V/x] \\
(\beta.\to) & (\lambda x.M)\, V & \longrightarrow & M[V/x]
\end{array}$$

$$\frac{M \longrightarrow N}{E[M] \longrightarrow E[N]}$$

$$E ::= [\,] \mid E\, V \mid \mathbf{let}\, x \leftarrow E\, \mathbf{in}\, N$$

Figure 2. Operational Semantics for Core λ_{eff}

$$\begin{array}{l}
\mathsf{noisey} : \{[\mathsf{Bool}]\}_{\mathsf{GB}} \\
\mathsf{noisey} = \{\mathbf{let}\, x \leftarrow \mathsf{get}()\, \mathbf{in} \\
\qquad \mathbf{if}\, x\, \mathbf{then\, return}\, x\, \mathbf{else}\, (\mathsf{beep}(); \mathbf{return}\, x)\} \\
\\
\mathsf{flip} : \{[\mathsf{Bool}]\}_{\mathsf{State\ Bool}} \\
\mathsf{flip} = \{\mathbf{let}\, x \leftarrow \mathsf{get}()\, \mathbf{in} \\
\qquad \mathbf{let}\, y \leftarrow \neg x\, \mathbf{in} \\
\qquad\quad \mathsf{put}(y); \mathbf{return}\, x\} \\
\\
\mathsf{reset} : \{[\mathsf{Bool}]\}_{\mathsf{State\ Bool}} \\
\mathsf{reset} = \{\mathbf{let}\, x \leftarrow \mathsf{get}()\, \mathbf{in} \\
\qquad \mathsf{put}(\mathsf{False}); \mathbf{return}\, x\}
\end{array}$$

Figure 3. Core λ_{eff} examples

Syntactic Sugar We will use the following syntactic sugar:

$$\begin{array}{rcll}
M; N & \equiv & \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N, & x \text{ fresh} \\
\mathsf{Bool} & \equiv & 1 + 1 & \\
\mathsf{True} & \equiv & \mathbf{inj}_1\, () & \\
\mathsf{False} & \equiv & \mathbf{inj}_2\, () & \\
\mathbf{if}\, V\, \mathbf{then}\, M\, \mathbf{else}\, N & \equiv & \mathbf{case}(V, x.M, y.N), & x, y \text{ fresh} \\
\neg V & \equiv & \mathbf{if}\, V\, \mathbf{then\, return}\, \mathsf{True}\, \mathbf{else\, return}\, \mathsf{False} &
\end{array}$$

Examples Figure 3 presents the four example abstract computations from Section 2 as λ_{eff} terms.

3.2 Effect Handlers

Core λ_{eff} allows us to write abstract computations over arbitrary effect signatures. An effect handler provides an *interpretation* of an abstract computation.

We extend the grammar for Core λ_{eff} as follows.
Handler types

$$R ::= A\ {}^{E}\!\!\Rightarrow^{E'} C$$

Handlers

$$H ::= \{\mathbf{return}\, x \mapsto M\} \mid H \uplus \{\mathrm{op}\, p\, k \mapsto N\}$$

Handling

$$M ::= \cdots \mid \mathbf{handle}\, M\, \mathbf{with}\, H$$

A handler type $A\ {}^{E}\!\!\Rightarrow^{E'} C$ interprets a returner computation of type $[A]$ with effects E as a computation of type C with effects E'. A handler is defined by a return clause $\mathbf{return}\, x \mapsto M$ and a collection of operation clauses $\{\mathrm{op}_i\, p\, k \mapsto N_i\}_i$. The return clause defines how to handle final return values. The returned value is bound to the variable x in M. The operation clauses define how to handle each operation. The operation parameter is bound to p, and the continuation is bound to k in N. Providing direct access to the whole continuation allows it to be used non-linearly, which is important for implementing effects such as exceptions, and non-deterministic choice, for instance.

For any handler:

$$H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\mathrm{op}_i\, p\, k \mapsto N_i\}_i$$

we define the action of H on return values as follows:

$$H(\mathbf{return}, V) = M[V/x]$$

and the action of H on operations handled by H as follows:

$$H(\mathrm{op}_i, V, W) = N_i[V/p, W/k]$$

The typing rules for handlers are given in Figure 4. The operational semantics for handlers is given in Figure 5.

Reifying Handlers In essence, the type $A\ {}^{E}\!\!\Rightarrow^{E'} C$ behaves like a suspended function of type $\{\{[A]\}_E \to C\}_{E'}$. Indeed we can reify a handler H as a value as follows:

$$\frac{\Gamma \vdash H : A\ {}^{E}\!\!\Rightarrow^{E'} C}{\Gamma \vdash \{\lambda x.\mathbf{handle}\, x!\, \mathbf{with}\, H\} : \{\{[A]\}_E \to C\}_{E'}}$$

$\boxed{\Gamma \vdash_E M : C}$

$$\ldots \qquad \frac{\Gamma \vdash_E M : [A] \qquad \Gamma \vdash H : A \;{}^{E}\!\!\Rightarrow^{E'}\; C}{\Gamma \vdash_{E'} \textbf{handle}\ M\ \textbf{with}\ H : C}\ \textsc{Handle}$$

$\boxed{\Gamma \vdash H : A \;{}^{E}\!\!\Rightarrow^{E'}\; C}$

$$\frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \quad H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : \{B_i \to C\}_{E'} \vdash_{E'} N_i : C]_i \qquad \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \;{}^{E}\!\!\Rightarrow^{E'}\; C}\ \textsc{Handler}$$

Figure 4. Typing Rules for Handlers

$$\begin{array}{ll} (\textit{handle}.[\,]) & \textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H \longrightarrow H(\textbf{return}, V) \\ (\textit{handle}.\mathrm{op}) & \textbf{handle}\ D[\mathrm{op}\ V]\ \textbf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\lambda z.\textbf{handle}\ D[\textbf{return}\ z]\ \textbf{with}\ H\}) \end{array}$$

$$\begin{array}{ll} \text{(delimited computation contexts)} & D ::= [\,] \mid D\ V \mid \textbf{let}\ x \leftarrow D\ \textbf{in}\ N \\ \text{(evaluation contexts)} & E ::= [\,] \mid E\ V \mid \textbf{let}\ x \leftarrow E\ \textbf{in}\ N \mid \textbf{handle}\ E\ \textbf{with}\ H \end{array}$$

Figure 5. Operational Semantics for Handlers

Examples As a basic example, consider a handler for stateful computations:

$$\begin{array}{l} \mathsf{evalState} : A \;{}^{\mathsf{State}\ S}\!\!\Rightarrow^{\emptyset}\; (S \to [A]) \\ \mathsf{evalState} = \textbf{return}\ x \mapsto \lambda s.\textbf{return}\ x \\ \qquad\qquad\quad \mathsf{get}\ p\ k \quad \mapsto \lambda s.k\ s\ s \\ \qquad\qquad\quad \mathsf{put}\ p\ k \quad \mapsto \lambda s.k\ ()\ p \end{array}$$

This handler interprets a stateful computation of type $[A]$ over state type S as a pure function of type $S \to [A]$. The state is threaded through the computation such that it can be read and written using get and put, but is discarded when the computation returns. Now if we instantiate S to Bool, then we can apply $\mathsf{evalState}$ to flip:

$$\textbf{handle}\ \mathsf{flip}!\ \textbf{with}\ \mathsf{evalState}$$

yielding a function of type $\mathsf{Bool} \to [\mathsf{Bool}]$ that flips its argument.

An important property of abstract computations is that they are generic in the sense that they do not commit to a particular interpretation of the operations. For instance, we can define alternative handlers for state. A mild variation on the $\mathsf{evalState}$ handler is a handler that interprets get and put in the same way, but returns the final state along with the final return value of the computation.

$$\begin{array}{l} \mathsf{runState} : A \;{}^{\mathsf{State}\ S}\!\!\Rightarrow^{\emptyset}\; (S \to [A \times S]) \\ \mathsf{runState} = \textbf{return}\ x \mapsto \lambda s.\textbf{return}\ (x, s) \\ \qquad\qquad\quad \mathsf{get}\ p\ k \quad \mapsto \lambda s.k\ s\ s \\ \qquad\qquad\quad \mathsf{put}\ p\ k \quad \mapsto \lambda s.k\ ()\ p \end{array}$$

A slightly more interesting variation is the following handler, which logs each put operation, returning the list of all values written to the state cell in a list alongside the final return value.

$$\begin{array}{l} \mathsf{logState} : A \;{}^{\mathsf{State}\ S}\!\!\Rightarrow^{\emptyset}\; (S \to [A \times \mathsf{List}\ S]) \\ \mathsf{logState} = \textbf{return}\ x \mapsto \lambda s.\textbf{return}\ (x, \mathsf{Nil}) \\ \qquad\qquad\quad \mathsf{get}\ p\ k \quad \mapsto \lambda s.k\ s\ s \\ \qquad\qquad\quad \mathsf{put}\ p\ k \quad \mapsto \lambda s.\textbf{let}\ z \leftarrow k\ ()\ p\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{split}(z, x.ss.(x, \mathsf{Cons}\ s\ ss)) \end{array}$$

For this example, we have assumed a standard extension of λ_{eff} with list types $\mathsf{List}\ X$ and corresponding list constructors $\mathsf{Nil} : \mathsf{List}\ X$ and $\mathsf{Cons} : X \to \mathsf{List}\ X \to \mathsf{List}\ X$.

3.3 Effect Forwarding

In our prior work [8], we considered a number of practical extensions and variations on handlers, including shallow handlers, parameterised handlers, open handlers, effect forwarding, and effect polymorphism. Perhaps the most important extension is effect forwarding in conjunction with open handlers.

The idea is that an open handler handles all of the operations explicitly specified by its type, but it also supports other operations, which are forwarded to be handled by an outer handler. A key advantage of open handlers is that they compose. We might, for instance, define an open handler for state and an open handler for exceptions. We can then handle a computation that uses state, exceptions, and possibly other effects as well by first handling it with the state handler, and then handling the resulting computation with the exception handler, or vice-versa.

It is straightforward to adapt λ_{eff} to support open handlers. The typing rule for open handlers is:

$$\frac{\begin{array}{c} E = E' \oplus \{\mathrm{op}_i : A_i \to B_i\}_i \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : \{B_i \to C\}_{E'} \vdash_{E'} N_i : C]_i \\ \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \;{}^{E}\!\!\Rightarrow^{E'}\; C}\ \textsc{OpenHandler}$$

The only difference from the vanilla HANDLER rule is that the input effects are $E' \oplus E$ instead of just E, where $E' \oplus E$ is the extension of E' by E (where any clashes are resolved in favour of E).

As far as the semantics goes, we simply extend the action of a handler to forward operations with no operation clause:

$$H(\mathrm{op}, V, W) = \textbf{let}\ x \leftarrow \mathrm{op}\ V\ \textbf{in}\ W!\ x, \quad \mathrm{op} \neq \mathrm{op}_i \text{ for any } i$$

4. Flow Effects

In this section we introduce Core λ_{flow}, a variation of Core λ_{eff} that supports abstract idiom, arrow, and monad computations using a uniform syntax, and thus supporting generic effectful programming over all three kinds of effectful computation. The grammar of types is as follows:

$$\begin{array}{ll} \text{(values)} & A, B ::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid \{C\}_E \\ \text{(computations)} & C ::= [A] \mid A \to C \\ \text{(effect signatures)} & E ::= \{\mathrm{op} : A \to B\} \uplus E \mid \{\mathsf{f}\} \uplus E \mid \emptyset \\ \text{(flow effects)} & \mathsf{f} ::= \mathsf{c} \mid \mathsf{d} \\ \text{(environments)} & \Gamma ::= \Gamma, x : A \mid \Gamma, x^{\star} : A \mid \cdot \end{array}$$

The differences are highlighted in grey. As well as the usual operations, effect signatures can also include flow effects c and d, which denote dynamic control and data flow. In λ_{flow}, type environments distinguish between *active* ($x^{\star} : A$) and *inactive* ($x : A$) variables. Only active variables can be directly used. The flow effects allow inactive variables to be activated appropriately in order to realise dynamic control and data flow.

$\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}^\star\ \frac{(x^\star : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \textsc{Inj}_i\ \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \textsc{Thunk}\ \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : \{C\}_E}$$

$\boxed{\Gamma \vdash_E M : C}$

$$\textsc{Split}^\star\ \frac{\Gamma^\star \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C} \qquad \textsc{CaseZero}^\star\ \frac{\Gamma^\star \vdash V : 0}{\Gamma \vdash_E \mathbf{case}_0(V) : C}$$

$$\textsc{Case}\ \frac{\Gamma \vdash V : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash_E M_1 : C \quad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C} \qquad \textsc{Return}^\star\ \frac{\Gamma^\star \vdash V : A}{\Gamma \vdash_E \mathbf{return}\, V : [A]}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_E M : [A] \quad \Gamma, x : A \vdash_E N : C}{\Gamma \vdash_E \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : C} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash_E M : C}{\Gamma \vdash_E \lambda x.M : A \to C} \qquad \textsc{App}^\star\ \frac{\Gamma \vdash_E M : A \to C \quad \Gamma^\star \vdash V : A}{\Gamma \vdash_E M\, V : C}$$

$$\textsc{Force}\ \frac{\Gamma \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C} \qquad \textsc{Op}\ \frac{(\mathrm{op} : A \to B) \in E \quad \Gamma \vdash V : A}{\Gamma \vdash_E \mathrm{op}\, V : [B]}$$

$$\textsc{Op}^\star\ \frac{\mathrm{d} \in E \qquad (\mathrm{op} : A \to B) \in E \quad \Gamma^\star \vdash V : A}{\Gamma \vdash_E \mathrm{op}\, V : [B]} \qquad \textsc{Force}^\star\ \frac{\mathrm{c}, \mathrm{d} \in E \quad \Gamma^\star \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C} \qquad \textsc{ReturnC}^\star\ \frac{E' \subseteq \{\mathrm{c}, \mathrm{d}\} \quad \Gamma^\star \vdash_{E'} M : [A]}{\Gamma \vdash_E \mathbf{return}\, M : [A]}$$

Figure 6. Typing Rules for Core λ_{flow}

We define two meta operations on type environments. The first, *activation* ($\Gamma^\star$), activates all of the variables in Γ.

$$\begin{aligned} \cdot^\star &= \cdot \\ (\Gamma, x : A)^\star &= \Gamma^\star, x^\star : A \\ (\Gamma, x^\star : A)^\star &= \Gamma^\star, x^\star : A \end{aligned}$$

The second, *flushing* ($\Gamma^\dagger$), removes all of the inactive variables from Γ.

$$\begin{aligned} \cdot^\dagger &= \cdot \\ (\Gamma, x : A)^\dagger &= \Gamma^\dagger \\ (\Gamma, x^\star : A)^\dagger &= \Gamma^\dagger, x^\star : A \end{aligned}$$

The typing rules for Core λ_{flow} are given in Figure 6. The differences from Core λ_{eff} are again highlighted in grey. The variable rule restricts access to active variables.

$$\textsc{Var}^\star\ \frac{(x^\star : A) \in \Gamma}{\Gamma \vdash x : A}$$

The application rule, activates all variables in the argument value.

$$\textsc{App}^\star\ \frac{\Gamma \vdash_E M : A \to C \qquad \Gamma^\star \vdash V : A}{\Gamma \vdash_E M\, V : C}$$

This is always sound because β-reduction will always bind the argument value to an inactive variable. The elimination rules for products and empty sums are amended in order to activate the type environment in the eliminated value.

$$\textsc{Split}^\star\ \frac{\Gamma^\star \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C} \qquad \textsc{CaseZero}^\star\ \frac{\Gamma^\star \vdash V : 0}{\Gamma \vdash_E \mathbf{case}_0(V) : C}$$

This is sound because decomposing a product or an empty sum has no impact on control or data flow. The rule for returning a value activates all variables in the value.

$$\textsc{Return}^\star\ \frac{\Gamma^\star \vdash V : A}{\Gamma \vdash_E \mathbf{return}\, V : [A]}$$

Activating the type environment supports dynamic memory flow. In order to support dynamic data flow, we add a variant of the rule for operations that only applies if the d effect is present.

$$\textsc{Op}^\star\ \frac{\mathrm{d} \in E \qquad (\mathrm{op} : A \to B) \in E \quad \Gamma^\star \vdash V : [A]}{\Gamma \vdash_E \mathrm{op}\, V : [B]}$$

As well as the standard **return** V construct, we introduce a special **return** M construct, which yields the value returned by the pure returner computation M.

$$\textsc{ReturnC}^\star\ \frac{E' \subseteq \{\mathrm{c}, \mathrm{d}\} \qquad \Gamma^\star \vdash_{E'} M : [A]}{\Gamma \vdash_E \mathbf{return}\, M : [A]}$$

This rule allows final return values to be computed from any variables in the type environment using an arbitrary pure computation (E' can only include flow effects, so it must be pure). Finally, we include a special rule for forcing thunks, that only applies to thunked computations with the c and d flow effects.

$$\textsc{Force}^\star\ \frac{\mathrm{c}, \mathrm{d} \in E \qquad \Gamma^\star \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C}$$

This rule activates all of the variables in a type environment when forcing a thunk.

If $\mathrm{c}, \mathrm{d} \in E$ then the following derivation applies:

$$\dfrac{\dfrac{\Gamma^\star \vdash_E M : C}{\Gamma^\star \vdash \{M\} : \{C\}_E}\ \textsc{Thunk}}{\Gamma \vdash_E \{M\}! : C}\ \textsc{Force}^\star$$

Hence in the presence of all flow effects we can systematically activate all variables in the type environment and λ_{flow} degenerates into λ_{eff}, which explains why both the data flow and control flow effects appear in the ($\textsc{Force}^\star$) rule.

The operational semantics for λ_{flow} is given in Figure 7. The only differences from Core λ_{eff} (highlighted in grey) are the additional evaluation context for computing inside returned values and

$$
\begin{array}{lrl}
(\beta.\times) & \textbf{split}((V_1, V_2), x_1.x_2.M_1) & \longrightarrow M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \textbf{case}(\textbf{inj}_i\, V, x_1.M_1, x_2.M_2) & \longrightarrow M_i[V/x_i] \\
(\beta.\{\}) & \{M\}! & \longrightarrow M \\
\\
(\beta.[]) & \textbf{let}\ x \leftarrow \textbf{return}\ V\ \textbf{in}\ M & \longrightarrow M[V/x] \\
(\beta.\to) & (\lambda x.M)\ V & \longrightarrow M[V/x] \\
\\
(ret.ret) & \textbf{return}\ (\textbf{return}\ V) & \longrightarrow \textbf{return}\ V
\end{array}
$$

$$\frac{M \longrightarrow N}{E[M] \longrightarrow E[N]}$$

$$E ::= [\,] \mid E\ V \mid \textbf{let}\ x \leftarrow E\ \textbf{in}\ N \mid \textbf{return}\ E$$

Figure 7. Operational Semantics for Core λ_{flow}

the $(ret.ret)$ rule for converting the result of a pure computation into a plain value.

Examples Figure 8 presents the four example abstract computations of Section 2 and Figure 3 as λ_{flow} terms. The terms noisey and flip differ from the corresponding terms in Figure 3. The term reset is identical because it does not involve dynamic data or control flow.

In λ_{flow} we explicitly list the c and d effects in the type of noisey and we explicitly list the d effect in the type of flip. The λ_{eff} version of noisey does not type check in λ_{flow} because x is not active in the conditional test position. Thunking and forcing the conditional activates it. The λ_{eff} version of flip does not type check in λ_{flow} because x is not active in the negation. We take advantage of λ_{flow}'s ability to treat pure computations like values in order to activate it.

5. Handling Flow

In this section we present effect handlers for λ_{flow}. As there are two flow effects (c and d), there are four possible kinds of computation we might try to handle: monads ($\{\text{c}, \text{d}\}$), arrows ($\{\text{d}\}$), idioms ($\emptyset$), and something strange ($\{\text{c}\}$). As λ_{flow} does not have adequate support for writing computations of the latter kind, we will only consider handlers for the other three kinds.

As we have the inclusions $\emptyset \subseteq \{\text{d}\} \subseteq \{\text{c}, \text{d}\}$, monad handlers can handle arrow and idiom computations, and arrow handlers can handle idiom computations. However, there exist interpretations of arrow computations that cannot be specified using monad handlers and interpretations of idiom computations that cannot be specified using arrow or monad handlers.

The typing rules for monad, arrow, and idiom handlers are given in Figure 9. The operational semantics is given in Figure 10. We use the syntactic sugar:

$$\lambda(x, y).M \equiv \lambda z.\textbf{split}(z, x.y.M), \quad z \text{ fresh}$$

We now describe in detail the design of the different kinds of handler. Each kind of handler has the same syntax as standard handlers. The differences are in the typing rules, operational semantics, and the **handle** constructs.

5.1 Monad Handlers

We already know how to handle arbitrary monadic computations. The typing rules are the same as for standard handlers, except the effects of a handled computation may additionally include arbitrary flow effects. As abstract arrow and idiom computations are just restricted monad computations, it is sound for a monad handler to handle an abstract arrow or idiom computation. The operational semantics is unchanged. We annotate monad handlers and monad handler types with a T subscript.

$$
\begin{array}{l}
\text{noisey} : \{\,[\text{Bool}]\,\}_{\text{GB}\cup\{c,d\}} \\
\text{noisey} = \{\textbf{let}\ x \leftarrow \text{get}()\ \textbf{in} \\
\qquad \{\textbf{if}\ x\ \textbf{then}\ \textbf{return}\ x\ \textbf{else}\ (\text{beep}(); \textbf{return}\ x)\}!\} \\
\\
\text{flip} : \{\,[\text{Bool}]\,\}_{\text{State Bool}\cup\{d\}} \\
\text{flip} = \{\textbf{let}\ x \leftarrow \text{get}()\ \textbf{in} \\
\qquad \textbf{let}\ y \leftarrow \textbf{return}\ \neg x\ \textbf{in} \\
\qquad\quad \text{put}(y); \textbf{return}\ x\} \\
\\
\text{reset} : \{\,[\text{Bool}]\,\}_{\text{State Bool}} \\
\text{reset} = \{\textbf{let}\ x \leftarrow \text{get}()\ \textbf{in} \\
\qquad \text{put}(\text{False}); \textbf{return}\ x\}
\end{array}
$$

Figure 8. Core λ_{flow} examples

5.2 Arrow Handlers

The challenge of adapting conventional effect handlers to interpret arrow computations is that each operation clause must bind a continuation representing the rest of the computation, but in general this continuation need not inhabit the usual function space. Indeed, a key feature of arrows is that they abstract over computations with input and output, that is continuations, in such a way that the input need not be provided up-front.

For a monad T, a continuation is exactly a *Kleisli arrow* of type $A \to T\ B$ (or $A \to [B]$ in λ_{flow}). But arrows generalise Kleisli arrows, and thus continuations. For instance, for any state type S, a state transformer [19] of type:

$$\{S \to [A]\}_{\{\text{c,d}\}} \to [\{S \to [B]\}_{\{\text{c,d}\}}]$$

is an arrow with input type A and output type B.

Recall that an effect handler is a compositional interpreter for an abstract computation. An arrow handler provides an interpretation of an arrow computation with an input and an output. Arrow handler syntax is exactly the same as standard handler syntax. Arrow handler types do however differ from standard handler types.

We let X range over type variables, which we will use in order to ensure that arrow handlers are parametric in the input type. This is crucial, as it allows us to manually thread a context through arrow computations.

Arrow handler types have the following shape:

$$R_{\leadsto} ::= A\ {}^{E}\!\!\Rightarrow^{E'}_{\leadsto} G$$

where G is a unary type operator mapping types to types. The idea is that this interprets an abstract arrow computation of type $X \to [A]$ as a computation of type $G\ X$, where X can be instantiated at any type, and the interpretation is parametric in X. We will sometimes write type operators as type-level lambda functions of the form $\lambda X.C$, where the type variable X is bound in C.

The (ArrowHandle) rule describes how to handle an arrow computation with an arrow handler. The type environment Γ is flushed in M meaning that all dynamic input to the computation must be packaged up in z. As the handled computation has an input z, it is written as a lambda $\lambda z.M$. The return type of the conclusion is $G\,B$.

The (ArrowHandler) rule follows a similar structure to the (MonadHandler) rule. The key differences arise because arrow handlers handle computations with inputs. Thus the type of x in the return clause is a function from the input type X to A and similarly the type of the parameter p in an operation clause is a function from X to A_i. The computation type of the continuation k is $G\,(X \times B_i)$ instead of $B_i \to C$. The idea is that G models the type of an arrow computation: the argument to G is the input type, and the result

$\boxed{\Gamma \vdash_E M : C}$

$$\ldots \qquad \textsc{MonadHandle}\ \frac{\Gamma \vdash_E M : [A] \qquad \Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{T}} C}{\Gamma \vdash_{E'} \textbf{handle}_{\mathsf{T}}\ M\ \textbf{with}\ H : C} \qquad \textsc{ArrowHandle}\ \frac{\Gamma^{\dagger}, z : B \vdash_E M : [A] \qquad \Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\leadsto} G}{\Gamma \vdash_{E'} \textbf{handle}_{\leadsto}\ (\lambda z.M)\ \textbf{with}\ H : G\,B}$$

$$\textsc{IdiomHandle}\ \frac{\Gamma^{\dagger}, z : B \vdash_E M : [A] \qquad \Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{I}} G}{\Gamma \vdash_{E'} \textbf{handle}_{\mathsf{I}}\ (\lambda z.M)\ \textbf{with}\ H : G\,B}$$

$\boxed{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{T}} C}$

$$\textsc{MonadHandler}\ \frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \cup \{f_j\}_j \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : \{B_i \to C\}_{E'} \vdash_{E'} N_i : C]_i \qquad \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{T}} C}$$

$\boxed{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\leadsto} G}$

$$\textsc{ArrowHandler}\ \frac{\begin{array}{c} X \text{ fresh} \qquad \begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \cup \{f_j\}_j \qquad \mathrm{c} \notin E \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \end{array} \\ [\Gamma, p : \{X \to [A_i]\}_{E'}, k : \{G\,(X \times B_i)\}_{E'} \vdash_{E'} N_i : G\,X]_i \qquad \Gamma, x : \{X \to [A]\}_{E'} \vdash_{E'} M : G\,X \end{array}}{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\leadsto} G}$$

$\boxed{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{I}} G}$

$$\textsc{IdiomHandler}\ \frac{\begin{array}{c} X \text{ fresh} \qquad \begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \cup \{f_j\}_j \qquad \mathrm{c}, \mathrm{d} \notin E \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \end{array} \\ [\Gamma, p : A_i, k : \{G\,(X \times B_i)\}_{E'} \vdash_{E'} N_i : G\,X]_i \qquad \Gamma, x : \{X \to [A]\}_{E'} \vdash_{E'} M : G\,X \end{array}}{\Gamma \vdash H : A \,{}^{E}\!\!\Rightarrow^{E'}_{\mathsf{I}} G}$$

Figure 9. Typing Rules for λ_{flow} Handlers

$$\begin{array}{ll}
(\textit{handle}_{\mathsf{T}}.[\,]) & \textbf{handle}_{\mathsf{T}}\ (\textbf{return}\ V)\ \textbf{with}\ H \longrightarrow H(\textbf{return}, V) \\
(\textit{handle}_{\mathsf{T}}.\mathrm{op}) & \textbf{handle}_{\mathsf{T}}\ D[\mathrm{op}\ V]\ \textbf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\lambda x.\textbf{handle}\ D[\textbf{return}\ x]\ \textbf{with}\ H\}) \\
(\textit{handle}_{\leadsto}.[\,]) & \textbf{handle}_{\leadsto}\ (\lambda z.\textbf{return}\ P)\ \textbf{with}\ H \longrightarrow H(\textbf{return}, \{\lambda z.\textbf{return}\ P\}) \\
(\textit{handle}_{\leadsto}.\mathrm{op}) & \textbf{handle}_{\leadsto}\ (\lambda z.D[\mathrm{op}\ V])\ \textbf{with}\ H \longrightarrow H(\mathrm{op}, \{\lambda z.\textbf{return}\ V\}, \{\textbf{handle}_{\leadsto}\ (\lambda(z,x).D[\textbf{return}\ x])\ \textbf{with}\ H\}) \\
(\textit{handle}_{\mathsf{I}}.[\,]) & \textbf{handle}_{\mathsf{I}}\ (\lambda z.\textbf{return}\ P)\ \textbf{with}\ H \longrightarrow H(\textbf{return}, \{\lambda z.\textbf{return}\ P\}) \\
(\textit{handle}_{\mathsf{I}}.\mathrm{op}) & \textbf{handle}_{\mathsf{I}}\ (\lambda z.D[\mathrm{op}\ V])\ \textbf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\textbf{handle}_{\mathsf{I}}\ (\lambda(z,x).D[\textbf{return}\ x])\ \textbf{with}\ H\})
\end{array}$$

$$\begin{array}{ll}
\text{(value or computation term)} & P ::= V \mid M \\
\text{(delimited computation contexts)} & D ::= [\,] \mid D\ V \mid \textbf{let}\ x \leftarrow D\ \textbf{in}\ N \\
\text{(evaluation contexts)} & E ::= [\,] \mid E\ V \mid \textbf{let}\ x \leftarrow E\ \textbf{in}\ N \mid \textbf{return}\ E \\
& \quad \mid\ \textbf{handle}_{\mathsf{T}}\ E\ \textbf{with}\ H \mid \textbf{handle}_{\leadsto}\ (\lambda z.E)\ \textbf{with}\ H \mid \textbf{handle}_{\mathsf{I}}\ (\lambda z.E)\ \textbf{with}\ H
\end{array}$$

Figure 10. Operational Semantics for λ_{flow} Handlers

of applying G to a type is the output type. In the continuation, the current input type is paired up with the return type of the operation.

The operational semantics is similar to that for monadic handlers. The differences are all related to explicitly threading the context through the handler. When handling a return clause (*handle*$_{\leadsto}$.[]), the returned term is a function of the input. (Note that we must account for the possibility that P is a computation term M as z may appear free in M, in which case **return** M may be stuck.) When handling an operation (*handle*$_{\leadsto}$.op), the parameter is a function of the input, and the continuation extends the context with the return value of the operation.

It is not sound to apply an arrow handler to a monadic computation and thus the (ArrowHandler) rule prevents arrow handlers from being applied to computations with dynamic control flow. Concretely, this constraint ensures that closed terms do not get stuck. For instance, it disallows the term:

$$\textbf{handle}_{\leadsto}\ (\lambda z.\{\textbf{case}(z, x_1.M_1, x_2.M_2)\}!)\ \textbf{with}\ H$$

which reduces to the stuck term:

$$\textbf{handle}_{\leadsto}\ (\lambda z.\textbf{case}(z, x_1.M_1, x_2.M_2))\ \textbf{with}\ H$$

As every abstract idiom computation is just a restricted abstract arrow computation, it is perfectly sound to handle an abstract idiom computation with an arrow handler.

Reifying Arrow Handlers Just as standard handlers can be reified as values, so can arrow handlers. In essence, for any type X, the arrow handler type $A\ {}^{E}{\Rightarrow}^{E'}_{\rightsquigarrow}\ G$ behaves like a suspended function of type $\{\{X \to [A]\}_E \to G\,X\}_{E'}$. Indeed, for any type X, we can reify an arrow handler H as a value as follows:

$$\frac{\Gamma \vdash H : A\ {}^{E}{\Rightarrow}^{E'}_{\rightsquigarrow}\ G}{\Gamma \vdash \{\lambda y.\mathbf{handle}_{\rightsquigarrow}\ (\lambda x.y!\ x)\ \mathbf{with}\ H\} : \{\{X \to [A]\}_E \to GX\}_{E'}}$$

Examples We can systematically transform any monad handler into a corresponding arrow handler. For instance, evalState can be rewritten as an arrow handler as follows.

$$\begin{array}{lll}
\mathsf{evalState}_{\rightsquigarrow} : A\ {}^{\mathsf{State}\ S \uplus \{\mathrm{d}\}}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\rightsquigarrow}\ \lambda X.X \to S \to [A] \\
\mathsf{evalState}_{\rightsquigarrow} = \mathbf{return}\,x & \mapsto x! \\
\qquad\qquad\quad \mathsf{get}\,p\,k & \mapsto \lambda z.\lambda s.k!\ (z, s)\ s \\
\qquad\qquad\quad \mathsf{put}\,p\,k & \mapsto \lambda z.\lambda s.\mathbf{let}\ p' \leftarrow p\ z\ \mathbf{in}\ k!\ (z, ())\ p'
\end{array}$$

However, we can also provide other interpretations that are not possible with monad handlers. As a simple, albeit slightly contrived, example, we can write an alternative arrow handler for state that determines statically whether a computation performs a put operation or not.

$$\begin{array}{lll}
\mathsf{puts}_{\rightsquigarrow} : A\ {}^{\mathsf{State}\ S \uplus \{\mathrm{d}\}}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\rightsquigarrow}\ \lambda X.[\mathsf{Bool}] \\
\mathsf{puts}_{\rightsquigarrow} = \mathbf{return}\,x & \mapsto \mathbf{return}\ \mathsf{False} \\
\qquad\quad \mathsf{get}\,p\,k & \mapsto k! \\
\qquad\quad \mathsf{put}\,p\,k & \mapsto \mathbf{return}\ \mathsf{True}
\end{array}$$

As a computation handled by puts must be an abstract arrow computation, in which the operations performed are independent of the values returned by intermediate operations, it is possible to compute the result statically without actually supplying any state. This is not possible for any monad handler for state as the number of times that put is performed is in general a dynamic property in a monadic computation.

Let us now consider an effect signature for failure:

$$\mathsf{Fail} = \{\mathsf{fail} : \forall X.1 \to X\}$$

The universal quantifier indicates that fail is a polymorphic operation. Effectively it defines an infinite family of operations parameterised by X. Correspondingly, an operation clause for fail defines an infinite family of operation clauses parameterised by X. Polymorphic operations and corresponding polymorphic operation clauses are a straightforward extension that applies just as well to λ_{flow} as it does λ_{eff} [8]. We could give fail the signature $1 \to 0$, but we choose to make it polymorphic because doing so is more convenient for writing example code. We can handle failure as an option type in the standard way:

$$\begin{array}{lll}
\mathsf{maybe}_{\rightsquigarrow} : A\ {}^{\mathsf{Fail} \uplus \{\mathrm{d}\}}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\rightsquigarrow}\ \lambda X.[\{X \to A\}_{\{\mathrm{c},\mathrm{d}\}} + 1] \\
\mathsf{maybe}_{\rightsquigarrow} = \mathbf{return}\,x & \mapsto \mathbf{return}\ (\mathbf{inj}_1\ x) \\
\qquad\qquad \mathsf{fail}\,p\,k & \mapsto \mathbf{return}\ (\mathbf{inj}_2\ ())
\end{array}$$

Alternatively, we can write a non-standard handler that in the event of failure counts the total number of failures (a quantity which is fixed for an abstract arrow computation, but not for an abstract monad computation):

$$\begin{array}{lll}
\mathsf{fails}_{\rightsquigarrow} : A\ {}^{\mathsf{Fail} \uplus \{\mathrm{d}\}}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\rightsquigarrow}\ \lambda X.[\{X \to A\}_{\{\mathrm{c},\mathrm{d}\}} + \mathsf{Nat}] \\
\mathsf{fails}_{\rightsquigarrow} = \mathbf{return}\,x & \mapsto \mathbf{return}\ (\mathbf{inj}_1\ x) \\
\qquad\quad \mathsf{fail}\,p\,k & \mapsto \mathbf{let}\ r \leftarrow k!\ \mathbf{in} \\
 & \qquad \mathbf{case}(r, \\
 & \qquad\qquad x.\mathbf{return}\ (\mathbf{inj}_2\ (\mathsf{S}\ \mathsf{Z}))) \\
 & \qquad\qquad n.\mathbf{return}\ (\mathbf{inj}_2\ (\mathsf{S}\ n)),
\end{array}$$

where we assume a data type of natural numbers Nat with constructors S and Z. If we define:

$$\begin{array}{l}
\mathsf{twoFail} : \{[A \times B]\}_{\mathsf{Fail} \uplus \{\mathrm{d}\}} \\
\mathsf{twoFail} = \{\mathbf{let}\ x \leftarrow \mathsf{fail}\ ()\ \mathbf{in}\ \mathbf{let}\ y \leftarrow \mathsf{fail}\ ()\ \mathbf{in}\ \mathbf{return}\ (x, y)\}
\end{array}$$

then:

$$(\mathbf{handle}_{\rightsquigarrow}\ (\lambda z.\mathsf{twoFail!})\ \mathbf{with}\ \mathsf{maybe}_{\rightsquigarrow})\ ()\ \longrightarrow^*\ \mathbf{return}\ ()$$

and:

$$(\mathbf{handle}_{\rightsquigarrow}\ (\lambda z.\mathsf{twoFail!})\ \mathbf{with}\ \mathsf{fails}_{\rightsquigarrow})\ ()\ \longrightarrow^*\ \mathbf{return}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))$$

5.3 Idiom Handlers

The IDIOMHANDLER rule is similar to the ARROWHANDLER rule. The difference is that the context is not threaded through parameters in operation clauses — directly capturing the property that idiom computations do not have the data flow effect. The (*handle*$_{\mathsf{I}}$.[]) rule is identical to the (*handle*$_{\rightsquigarrow}$.[]) rule. The (*handle*$_{\mathsf{I}}$.op) rule is similar to the (*handle*$_{\rightsquigarrow}$.op) rule; the only difference is that the context is not threaded through operation parameters.

It is not sound to apply an idiom handler to an abstract monad or arrow computation and thus the (IDIOMHANDLER) rule prevents idiom handlers from being applied to computations with dynamic data flow. Concretely, this constraint ensures that closed terms do not become open. For instance, if we were to allow dynamic data flow then we could write:

$$M = \mathbf{handle}_{\mathsf{I}}\ (\lambda y.\mathrm{op}\ y)\ \mathbf{with}\ H$$

Now suppose:

$$H(\mathrm{op}, y, \{\mathbf{handle}_{\mathsf{I}}\ ((\lambda(z, x).D[\mathbf{return}\ x]))\ \mathbf{with}\ H\}) = \mathbf{return}\ y$$

then M reduces to the open term $\mathbf{return}\ y$.

Reifying Idiom Handlers Just as monadic and arrow handlers can be reified as values, so can idiom handlers. For any type X, we can reify an idiom handler H as a value as follows:

$$\frac{\Gamma \vdash H : A\ {}^{E}{\Rightarrow}^{E'}_{\mathsf{I}}\ G}{\Gamma \vdash \{\lambda y.\mathbf{handle}_{\mathsf{I}}\ (\lambda x.y!\ x)\ \mathbf{with}\ H\} : \{\{X \to [A]\}_E \to GX\}_{E'}}$$

Examples Just as we can transform any monad handler into an corresponding arrow handler, we can systematically transform any arrow handler (or monad handler) into a corresponding idiom handler. For instance, $\mathsf{evalState}_{\rightsquigarrow}$ and $\mathsf{puts}_{\rightsquigarrow}$ can be written as idiom handlers as follows.

$$\begin{array}{lll}
\mathsf{evalState}_{\mathsf{I}} : A\ {}^{\mathsf{State}\ S}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\mathsf{I}}\ \lambda X.X \to S \to [A] \\
\mathsf{evalState}_{\mathsf{I}} = \mathbf{return}\,x & \mapsto \lambda z.\lambda s.x!\ z \\
\qquad\qquad\quad \mathsf{get}\,p\,k & \mapsto \lambda z.\lambda s.k!\ (z, s)\ s \\
\qquad\qquad\quad \mathsf{put}\,p\,k & \mapsto \lambda z.\lambda s.k!\ (z, ())\ p
\end{array}$$

$$\begin{array}{lll}
\mathsf{puts}_{\mathsf{I}} : A\ {}^{\mathsf{State}\ S}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\mathsf{I}}\ \lambda X.[\mathsf{Bool}] \\
\mathsf{puts}_{\mathsf{I}} = \mathbf{return}\,x & \mapsto \mathbf{return}\ \mathsf{False} \\
\qquad\quad \mathsf{get}\,p\,k & \mapsto k! \\
\qquad\quad \mathsf{put}\,p\,k & \mapsto \mathbf{return}\ \mathsf{True}
\end{array}$$

The only difference between the definition of $\mathsf{evalState}_{\rightsquigarrow}$ and $\mathsf{evalState}_{\mathsf{I}}$ is that the latter does not apply p to the context in the clause that handles put, as it cannot depend on the context. The definitions of $\mathsf{puts}_{\rightsquigarrow}$ and $\mathsf{puts}_{\mathsf{I}}$ are identical because the operation parameters are never used in either.

Here is another idiom handler for state that statically sums all of the values that will be written to the state cell by an abstract computation:

$$\begin{array}{lll}
\mathsf{putSum}_{\mathsf{I}} : A\ {}^{\mathsf{State}\ \mathsf{Nat}}{\Rightarrow}^{\{\mathrm{c},\mathrm{d}\}}_{\mathsf{I}}\ \lambda X.[\mathsf{Nat}] \\
\mathsf{putSum}_{\mathsf{I}} = \mathbf{return}\,x & \mapsto \mathbf{return}\ Z \\
\qquad\qquad \mathsf{get}\,p\,k & \mapsto k! \\
\qquad\qquad \mathsf{put}\,p\,k & \mapsto \mathbf{let}\ x \leftarrow k!\ \mathbf{in}\ \mathbf{return}\ p + x
\end{array}$$

This is not possible for any arrow handler for state, as in an abstract arrow computation the parameters to put operations can be dynamic.

For the following example, we assume λ_{flow} has been extended with a List data type with constructors Nil and Cons and String $\equiv$ List char. Let us consider a variation on failure in which an error message is reported alongside failure:

$$\mathsf{Error} = \{\mathsf{Err} : \forall X.\mathsf{String} \to X\}$$

We can straightforwardly write idiom handlers that have analogous behaviour to $\mathsf{maybe}_{\rightsquigarrow}$ and $\mathsf{fails}_{\rightsquigarrow}$. We can also define a handler that, in the case of error, aggregates all possible error messages.

$$\begin{array}{l}
\mathsf{errs_I} : A \overset{\mathsf{Error}}{\Rightarrow}{}_{\mathsf{I}}^{\{\mathrm{c,d}\}}\ \lambda X.\,[\{X \to A\}_{\{\mathrm{c,d}\}} + (\mathsf{List\ String})] \\
\mathsf{errs_I} = \mathbf{return}\ x \mapsto \mathbf{return}\ (\mathbf{inj}_1\ x) \\
\qquad\quad \mathsf{Err}\ s\ k \quad \mapsto \mathbf{let}\ r \leftarrow k!\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{case}(r, \\
\qquad\qquad\qquad\qquad\qquad ss.\mathbf{return}\ (\mathbf{inj}_2\ (\mathsf{Cons}\ s\ ss)), \\
\qquad\qquad\qquad\qquad\qquad x.\mathbf{return}\ (\mathbf{inj}_2\ (\mathsf{Cons}\ s\ \mathsf{Nil})))
\end{array}$$

This is not possible for any arrow handler for state, as in an abstract arrow computation the parameters to Err operations can be dynamic.

5.4 Forwarding for Arrow and Idiom Handlers

As the continuation of an operation handled by an arrow or an idiom handler may not be in the standard function space, we cannot use the universal definition of forwarding that works for monadic handlers. One possibility is to add a special forwarding clause:

$$\mathit{default}\ p\ k \mapsto N'$$

where

$$H(\mathrm{op}, V, W) = N'[\{\lambda x.\mathrm{op}\ x\}/\mathit{default}, V/p, W/k], \quad \mathrm{op} \neq \mathrm{op}_i \text{ for any } i$$

We leave it to future work to investigate how this idea pans out in practice.

6. Example: Parser Combinators

As a larger example, we now build a collection of generic effectful parser combinators. We choose parser combinators [24] as there exist monad, arrow and idiom variations, each with different practical tradeoffs. Monad parser combinators are the most general, admitting fully context-sensitive grammars. Arrow parser combinators admit a small amount of context-sensitivity, for instance supporting parsing of XML, where a close tag must match the corresponding open tag. Idiom parser combinators capture context-free grammars.

In λ_{flow} each of these can be constructed using the same signature of parser operations. An important benefit of restricting the power of parser combinators is that doing so admits more efficient implementations. Rather than focusing on the intricacies of such implementations (which are discussed in detail elsewhere [24]), we present the necessary infrastructure to support generic parsers, without committing to a particular concrete implementation.

We make use of a number of extensions to λ_{flow} in order to make the example more realistic: recursive functions, pattern matching, effect polymorphism, and effect forwarding. We also make use of additional syntactic sugar for writing code in a more concise call-by-value style.

- In the presence of control and data flow effects we automatically activate the entire type environment (so we never need to write $\{M\}!$).
- We allow **return** V to be written as simply V when doing so is not ambiguous.
- We allow forcing to be omitted when applying a function to an argument, writing $V\ W$ for $V!\ W$.
- We allow top-level definitions to be written directly without curly braces. For instance: id $x = x$ means id $= \{\lambda x.x\}$.
- We allow computation terms to appear anywhere a value term is normally required within a computation term, in which case we desugar by binding such computations to values in left-to-right order. For instance $M\ N$ means **let** $x \leftarrow N$ **in** $M\ x$ and (M, N) means **let** $x \leftarrow M$ **in let** $y \leftarrow N$ **in return** (x, y), where x, y are fresh variables.

Let us begin with some primitive parsing operations:

$$\mathsf{any} : 1 \to \mathsf{Char} \qquad\qquad \mathsf{char} : \mathsf{Char} \to \mathsf{Char}$$

The any operation parses any character and the char operation parses a specific character. We do not need to define a special operation for the constant parser as it is simply **return**. Similarly, we do not need a special operation for sequential composition of parsers, as let binding gives us this functionality for free. For instance, the following invokes parsers p and q in sequence and then combines their results by function application:

$$\mathbf{let}\ f \leftarrow p\ \mathbf{in}\ \mathbf{let}\ x \leftarrow q\ \mathbf{in}\ \mathbf{return}\ (f\ x)$$

We make use of a failure operation:

$$\mathsf{fail} : \forall X.1 \to X$$

along with an operation to choose between two parsers:

$$\mathsf{choose} : \forall X.(\{[X]\}_E, \{[X]\}_E) \to X$$

where we will need to instantiate E appropriately. The idea is that this operation takes two parsers returning type X as arguments, tries running both, and then returns an answer of type X; exactly how it does so depends on how the operation is handled. A handler might return a list of all matches, or it might just return the first match; it might run the parsers sequentially, or it might run them in parallel. The effect signature E must include all of the operations we intend to support for parsing — including choose itself. (The fail and choose operations together allow us to variously implement the behaviour of the Haskell type classes `Alternative`, `ArrowChoice`, and `MonadPlus`, which add additional monoidal structure respectively to idioms, arrows, and monads.)

Let us now define a suitable effect signature, Parse:

$$\begin{array}{l}
\mathsf{Parse}\ \varepsilon = \{\mathsf{any} : 1 \to \mathsf{Char}, \\
\qquad\qquad \mathsf{char} : \mathsf{Char} \to \mathsf{Char} \\
\qquad\qquad \mathsf{fail} : \forall X.1 \to X \\
\qquad\qquad \mathsf{choose} : \forall X.(\{[X]\}_{\mathsf{Parse}\ \varepsilon}, \{[X]\}_{\mathsf{Parse}\ \varepsilon}) \to X\} \\
\qquad \uplus\ \varepsilon
\end{array}$$

It is parameterised by an effect variable ε that can be instantiated with any of the flow effects in order to choose between idiom, arrow, and monad interpretations.

Using the choose operation and a recursive function call, we can define a composite parser that applies its argument zero or more times:

$$\begin{array}{l}
\mathsf{many} : \{\{[X]\}_{\mathsf{Parse}\ \varepsilon} \to [\mathsf{List}\ X]\}_{\mathsf{Parse}\ \varepsilon} \\
\mathsf{many}\ p = \mathsf{choose}\ (\mathsf{Nil}, \mathsf{Cons}\ p\ (\mathsf{many}\ p))
\end{array}$$

We parse digits as follows:

$$\begin{array}{l}
\mathsf{digit} : \{[\mathsf{char}]\}_{\mathsf{Parse}\ \varepsilon} \\
\mathsf{digit} = \mathsf{choose}(\mathsf{char}\ '0', \\
\qquad\qquad \mathsf{choose}(\mathsf{char}\ '1', \ldots, \mathsf{choose}(\mathsf{char}\ '9') \ldots))
\end{array}$$

Let us suppose that we have a function for converting a sequence of digits into a natural number:

$$\mathsf{digitsToNat} : \{\mathsf{String} \to [\mathsf{Nat}]\}_{\{\mathrm{c,d}\}}$$

The following parser parses natural numbers:

```
$\mathsf{num} : \{[\mathsf{Nat}]\}_{\mathsf{Parse}\ \varepsilon}$
$\mathsf{num} = \mathbf{let}\ x \leftarrow \mathsf{many\ digit}\ \mathbf{in}\ \mathsf{digitsToNat}$
```

Here is a function that parses a comma delimited list of natural numbers terminated by an at sign (@):

```
$\mathsf{nums} : \{\mathsf{Nat} \to [\mathsf{List\ Nat}]\}_{\mathsf{Parse}\ \{c,d\}}$
$\mathsf{nums\ Z}$       $= \mathsf{char}\ \text{‘@’}; \mathsf{Nil}$
$\mathsf{nums}\ (\mathsf{S}\ n) = \mathbf{let}\ x \leftarrow \mathsf{num}\ ()\ \mathbf{in}\ \mathsf{char}\ \text{‘,’}; \mathsf{Cons}\ x\ (\mathsf{nums}\ n)$
```

Because it performs a case split on its argument, nums is context-sensitive and can only be handled by a monad handler. However, by separating out the computation into two stages, we can define a variant that generates a context-free parser computation that can be handled by arrow or even idiom handlers:

```
$\mathsf{nums}' : \{\mathsf{Nat} \to [\{[\mathsf{List\ Nat}]\}_{\mathsf{Parse}\ \varepsilon}]\}_{\{c,d\}}$
$\mathsf{nums}'\ \mathsf{Z}$       $= \{\mathsf{char}\ \text{‘@’}; \mathsf{Nil}\}$
$\mathsf{nums}'\ (\mathsf{S}\ n) = \mathbf{let}\ ys \leftarrow \mathsf{nums}'\ n\ \mathbf{in}$
                $\{\mathbf{let}\ x \leftarrow \mathsf{num}\ ()\ \mathbf{in}\ \mathsf{char}\ \text{‘,’}; \mathsf{Cons}\ x\ (ys!)\}$
```

The outer computation performs the recursion on n. The generated parser computation is independent of n.

Here is a context-sensitive parser that parses a single number delimited by a character read dynamically from the input:

```
$\mathsf{delim} : \{[\mathsf{Nat}]\}_{\mathsf{Parse}\ \{d\}}$
$\mathsf{delim} = \mathbf{let}\ c \leftarrow \mathsf{any}\ ()\ \mathbf{in}$
        $\mathsf{char}\ \text{‘ ’}; \mathbf{let}\ x \leftarrow \mathsf{num}\ ()\ \mathbf{in}\ \mathsf{char}\ \text{‘ ’}; \mathsf{char}\ c; x$
```

As c does not affect control flow, delim can be handled by an arrow handler.

Here is a context-sensitive parser that parses a number n followed by a list of n numbers:

```
$\mathsf{nnums} : \{[\mathsf{Nat}]\}_{\mathsf{Parse}\ \{c,d\}}$
$\mathsf{nnums} = \mathbf{let}\ x \leftarrow \mathsf{num}\ ()\ \mathbf{in}\ \mathsf{char}\ \text{‘:’}; \mathsf{nums}\ n$
```

This is a truly dynamic parser, that can only be handled by a monad handler.

In order to implement a handler for abstract parser computations, we will make use of effect signatures for reading characters, state, and failure:

```
$\mathsf{Read} = \{\mathsf{getc} : 1 \to \mathsf{char}\}$
$\mathsf{State}\ S = \{\mathsf{get} : 1 \to S, \mathsf{put} : S \to 1\}$
$\mathsf{Fail} = \{\mathsf{fail} : \forall X.1 \to X\}$
```

Let us define an idiom handler for parsing (reified as a function):

```
$\mathsf{parse_I} : \{\{[X]\}_{\mathsf{Parse}\ \emptyset} \to [X]\}_{\mathsf{Read} \uplus \mathsf{Fail} \uplus \{c,d\}}$
$\mathsf{parse_I} =$
  $\mathbf{handle_I}\ (\lambda m.m!)\ \mathbf{with}$
    $\mathbf{return}\ x$       $\mapsto x!$
    $\mathsf{any}\ ()\ k$       $\mapsto \mathbf{let}\ c \leftarrow \mathsf{getc}\ ()\ \mathbf{in}\ \lambda z.k\ (z, c)$
    $\mathsf{char}\ c\ k$       $\mapsto \mathbf{if}\ c == \mathsf{getc}\ ()\ \mathbf{then}\ \lambda z.k\ (z, c)$
                    $\mathbf{else}\ \lambda z.\mathsf{fail}\ ()$
    $\mathsf{fail}\ ()\ k$       $\mapsto \lambda z.\mathsf{fail}\ ()$
    $\mathsf{choose}\ (p, q)\ k \mapsto \mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse_I}\ p)\ \mathbf{of}$
                      $\mathsf{Just}\ v$    $\to \lambda z.k\ (z, v)$
                      $\mathsf{Nothing} \to$
                        $\mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse_I}\ q)\ \mathbf{of}$
                          $\mathsf{Just}\ v$    $\to \lambda z.k\ (z, v)$
                          $\mathsf{Nothing} \to \lambda z.\mathsf{fail}\ ()$
```

The most interesting operation clause is the one for choose. Given two parsers p and q, we first try to parse with p, and if that fails then we try with q. In order to test for failure, we recursively invoke parse and handle the result with maybe, a straightforward handler that interprets failure as an option data type Maybe X with constructors Just and Nothing:

```
$\mathsf{maybe} : \{\{X\}_{\mathsf{Fail} \uplus \{c,d\} \uplus \varepsilon} \to [\mathsf{Maybe}\ X]\}_{\{c,d\} \uplus \varepsilon}$
$\mathsf{maybe}\ m = \mathbf{handle_T}\ m!\ \mathbf{with}$
                 $\mathbf{return}\ x \mapsto \mathsf{Just}\ x$
                 $\mathsf{fail}\ ()\ k$  $\mapsto \mathsf{Nothing}$
```

We now define a handler for reading characters:

```
$\mathsf{read} : \{\{[X]\}_{\mathsf{Read} \uplus \{c,d\} \uplus \varepsilon} \to [X]\}_{\mathsf{State\ String} \uplus \mathsf{Fail}\{c,d\} \uplus \varepsilon}$
$\mathsf{read}\ cs\ m = \mathbf{handle_T}\ m!\ \mathbf{with}$
             $\mathbf{return}\ x \mapsto x$
             $\mathsf{getc}\ ()\ k \mapsto \mathbf{case}\ \mathsf{get}\ ()\ \mathbf{of}\ \mathsf{Nil}$          $\mapsto \mathsf{fail}\ ()$
                                     $\mathsf{Cons}\ c\ cs \mapsto \mathsf{put}\ cs; c$
```

This handler interprets getc in terms of state and failure. We can handle state with evalState:

```
$\mathsf{state} : \{\{X\}_{\mathsf{State}\ S \uplus \{c,d\} \uplus \varepsilon} \to S \to [X]\}_{\{c,d\}}$
$\mathsf{state}\ m = \mathbf{handle_T}\ m\ \mathbf{with}\ \mathsf{evalState}$
```

and use maybe for failure. Finally we compose all of these handlers together:

```
$\mathsf{runIdiomParser} : \{\{[X]\}_{\mathsf{Parse}\ \emptyset} \to \mathsf{String} \to [\mathsf{Maybe}\ X]\}_{\{c,d\}}$
$\mathsf{runIdiomParser}\ p\ cs = \mathsf{state}\ (\mathsf{maybe}\ (\mathsf{read}\ (\mathsf{parse_I}\ p)))\ cs$
```

Now:

```
$\mathbf{let}\ p \leftarrow \mathsf{nums}'\ 3\ \mathbf{in}\ \mathsf{runIdiomParser}\ p$ “42,21,7,@”
```

evaluates to Just (Cons 42 (Cons 21 (Cons 7 Nil))) and:

```
$\mathbf{let}\ p \leftarrow \mathsf{nums}'\ 4\ \mathbf{in}\ \mathsf{runIdiomParser}\ p$ “42,21,7,@”
```

evaluates to Nothing.

We can abstract over the $\mathsf{parse_I}$ handler, in particular allowing it to be replaced by arrow or monad handlers:

```
$\mathsf{runParser} : \{\{\{[X]\}_{\mathsf{Parse}\ \varepsilon} \to [X]\}_{\mathsf{Read} \uplus \mathsf{Fail} \uplus \{c,d\}} \to$
                 $\{[A]\}_{\mathsf{Parse}\ \varepsilon} \to \mathsf{String} \to [\mathsf{Maybe}\ X]\}_{\{c,d\}}$
$\mathsf{runParser}\ h\ p\ cs = \mathsf{state}\ (\mathsf{maybe}\ (\mathsf{read}\ (\mathsf{h\ p})))\ cs$
```

Let us define arrow and monad analogues of $\mathsf{parse_I}$:

```
$\mathsf{parse}_{\leadsto} : \{\{[X]\}_{\mathsf{Parse}\ \{d\}} \to [X]\}_{\mathsf{Read} \uplus \mathsf{Fail} \uplus \{c,d\}}$
$\mathsf{parse}_{\leadsto} =$
  $\mathbf{handle}_{\leadsto}\ (\lambda m.m!)\ \mathbf{with}$
    $\mathbf{return}\ x$       $\mapsto x!$
    $\mathsf{any}\ ()\ k$       $\mapsto \mathbf{let}\ c \leftarrow \mathsf{getc}\ ()\ \mathbf{in}\ \lambda z.k\ (z, c)$
    $\mathsf{char}\ c\ k$       $\mapsto \lambda z.\mathbf{if}\ c\ z == \mathsf{getc}\ ()\ \mathbf{then}\ k\ (z, c)$
                      $\mathbf{else}\ \mathsf{fail}\ ()$
    $\mathsf{fail}\ ()\ k$       $\mapsto \lambda z.\mathsf{fail}\ ()$
    $\mathsf{choose}\ (p, q)\ k \mapsto \lambda z.\mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse}_{\leadsto}\ (p\ z))\ \mathbf{of}$
                        $\mathsf{Just}\ v$    $\to \lambda z.k\ (z, v)$
                        $\mathsf{Nothing} \to$
                          $\mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse}_{\leadsto}\ (q\ z))\ \mathbf{of}$
                            $\mathsf{Just}\ v$    $\to \lambda z.k\ (z, v)$
                            $\mathsf{Nothing} \to \lambda z.\mathsf{fail}\ ()$
```

```
$\mathsf{parse_T} : \{\{[X]\}_{\mathsf{Parse}\ \{c,d\}} \to [X]\}_{\mathsf{Read} \uplus \mathsf{Fail} \uplus \{c,d\}}$
$\mathsf{parse_T} = \lambda m.$
  $\mathbf{handle_T}\ m!\ \mathbf{with}$
    $\mathbf{return}\ x$       $\mapsto x!$
    $\mathsf{any}\ ()\ k$       $\mapsto \mathbf{let}\ c \leftarrow \mathsf{getc}\ ()\ \mathbf{in}\ k\ c$
    $\mathsf{char}\ c\ k$       $\mapsto \mathbf{if}\ c == \mathsf{getc}\ ()\ \mathbf{then}\ k\ c$
                  $\mathbf{else}\ \mathsf{fail}\ ()$
    $\mathsf{fail}\ ()\ k$       $\mapsto \mathsf{fail}\ ()$
    $\mathsf{choose}\ (p, q)\ k \mapsto \mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse_T}\ p)\ \mathbf{of}$
                      $\mathsf{Just}\ v$    $\to k\ v$
                      $\mathsf{Nothing} \to$
                        $\mathbf{case}\ \mathsf{maybe}\ (\mathsf{parse_T}\ q)\ \mathbf{of}$
                          $\mathsf{Just}\ v$    $\to k\ v$
                          $\mathsf{Nothing} \to \mathsf{fail}\ ()$
```

Now, both:

$$\mathsf{runParser}\ \mathsf{parse}_{\leadsto}\ \mathsf{delim}\ \text{“}\texttt{* 42,21,7 *}\text{”}$$

and:

$$\mathsf{runParser}\ \mathsf{parse}_{\mathsf{T}}\ \mathsf{nnums}\ \text{“}\texttt{3:42,21,7,@}\text{”}$$

type check and evaluate to Just (Cons 42 (Cons 21 (Cons 7 Nil))).

We can use $\mathsf{parse}_{\mathsf{T}}$ to handle any arrow or idiom parsers (such as the nums′ and delim examples above). We can use $\mathsf{parse}_{\leadsto}$ to handle any idiom parsers (such as the nums′ example above). In practice, we would write optimised handlers for arrow and idiom parsers [24]. A key advantage of using a language like λ_{flow} is that we can write code that is generic in the choice of concrete implementation while using a uniform direct-style syntax.

7. Related Work

There has been a recent spate of work on practical languages and libraries for effect handlers. Apart from our own libraries, Kiselyov et al [9] has a similar library for Haskell, and Brady [3] has an effect handlers library for his dependently-typed language Idris. Two programming languages that build in algebraic effects and handlers as primitives are Bauer and Pretnar's Eff [1, 2] and McBride's Frank [15, 16]. None of these systems support algebraic effects or handlers for idioms or arrows.

Capriotti and Kaposi explore free idioms [4]. Free idioms correspond to abstract idiom computations. Yallop's thesis [26, Chapter 2] provides an in-depth analysis of idioms, arrows, and monads, expanding on the work of Lindley et al [14], and characterising the normal forms for idioms and arrows. We have implemented both free idiom and free arrow constructions in Haskell [11] directly inspired by the normal forms of Yallop.

Petricek and Syme [20] describe a novel use of F# computation expression syntax to write idiom computations using let notation, which is partly inspired by syntax for formlets [5], an abstraction for building web forms that is an idiom.

8. Future Work

This paper focuses on the theory of algebraic effects and handlers for arrows and idioms. In order to evaluate the practice of algebraic effects and handlers for arrows and idioms we would like to build an implementation.

One can implement handlers on top of our existing free idiom and free arrow constructions in Haskell. However, programming with free idioms and free arrows in Haskell requires the programmer to use a different syntax. Idioms only support a pointless syntax. Arrows support a direct-style syntax, but it is not quite the same as the do notation used for monads. Given that F# computation expressions are already expressive enough to cover a range of computation types including monads and idioms, it might be interesting to try to use computation expressions as a basis for building a source language for λ_{flow}. It may, however, be difficult to adequately encode an effect type system on top of F#. Ultimately, we expect the most fruitful path may be to build a new language, or extend a custom language like Frank or Eff.

On the theoretical side, it would be interesting to explore denotational semantics for λ_{flow} and to consider how the story is affected by reintroducing equations to the picture. Another direction is to explore algebraic effects and handlers for other variations on the basic theme, such as for linear and dependent types.

Acknowledgments

We would like Bob Atkey, Pierre-Evariste Dagand, and the anonymous referees for helpful feedback. This work was supported by EPSRC grants EP/J014591/1 and EP/K034413/1.

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[2] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. In *CALCO*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[3] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 2013.

[4] P. Capriotti and A. Kaposi. Free applicative functors. *CoRR*, abs/1403.0749, 2014.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. The essence of form abstraction. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008.

[6] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular reasoning about interference in incremental programming. *J. Funct. Program.*, 22(6):797–852, 2012.

[7] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[8] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In G. Morrisett and T. Uustalu, editors, *ICFP*, pages 145–158. ACM, 2013.

[9] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

[10] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[11] S. Lindley. Free idioms and free arrows in haskell, 2013. `https://github.com/slindley/dependent-haskell/tree/master/Free`.

[12] S. Lindley. Aspect oriented programming with handlers, 2013. `https://github.com/slindley/effect-handlers/blob/master/Examples/AOP.hs`.

[13] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *J. Funct. Program.*, 20(1):51–69, 2010.

[14] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electr. Notes Theor. Comput. Sci.*, 229(5):97–117, 2011.

[15] C. McBride. How might effectful programs look? In *Workshop on Effects and Type Theory*, 2007. `http://cs.ioc.ee/efftt/mcbride-slides.pdf`.

[16] C. McBride. Frank (0.3), 2012. `http://hackage.haskell.org/package/Frank`.

[17] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[18] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[19] R. Paterson. A new notation for arrows. In B. C. Pierce, editor, *ICFP*, pages 229–240. ACM, 2001.

[20] T. Petricek and D. Syme. The F# computation expression zoo. In M. Flatt and H.-F. Guo, editors, *PADL*, volume 8324 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.

[21] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *FoSSaCS*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[22] G. D. Plotkin and J. Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, 45:332–345, 2001.

[23] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[24] S. D. Swierstra. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, volume 5520 of *Lecture Notes in Computer Science*, pages 252–300. Springer, 2008.

[25] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[26] J. Yallop. *Abstraction for web programming*. PhD thesis, The University of Edinburgh, 2010.