Algebraic Effects and Effect Handlers for Idioms and Arrows

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.

1. Introduction

In previous work [7], we advocated Plotkin and Power's algebraic effects [19, 20] and effect handlers [21] 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 [16]. 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) [15] and Hughes' arrows [6]. The resulting calculus, λ_{flow}, extends λ_{eff} with *flow effects*, which explicitly track dependencies between the results of an effectful operation and subsequent effectful computation, allowing us to encode idiom and arrow computations. Crucially, λ_{flow} adds support for effect handlers for idiom and arrow computations.

One reason arrow computations and idiom computations are of interest is that they admit more instances than monadic computations. 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 [11, 12] (providing a uniform foundation for programming with idioms, arrows, and monads with a single effect).

Just as λ_{eff} gave us a firm basis for implementing practical implementations of standard effect handlers, we hope that λ_{flow} can provide a firm basis for implementing practical implementations of idiom and arrow handlers, and in particular support modular 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.
- A direct 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. [12] 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 a generalisation of standard monadic effect handlers.
- We give an embedding of arrow calculus (and its extensions to support monads and idioms) into λ_{flow}.

The remainder of the paper is structured as follows. Section 2 introduces the key ideas of our approach by first presenting abstract effectful computations as computation trees, and then describing 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 describes handlers in λ_{flow} for idiom, arrow, and monad computations along with soundness results for λ_{flow}. Section 6 gives an embedding of the arrow calculus into 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 [19, 20] 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.

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:

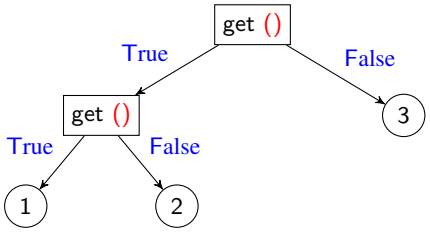
$$\mathsf{get}_{()}(\mathsf{get}_{()}(p, q), r) = \mathsf{get}_{()}(p, r) = \mathsf{get}_{()}(p, \mathsf{get}_{()}(q, r))$$
$$\mathsf{get}_{()}(p, p) = p$$

Note that when specifying equations in the algebraic approach, operations are typically written in a continuation passing style

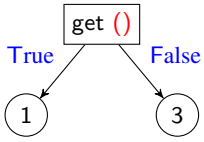
(CPS), exposing their algebraic structure. Thus $\mathsf{get}_{()}(p, q)$ corresponds to a term that a functional programmer would typically write in direct style as **let** $x \leftarrow \mathsf{get}\ ()$ **in if** x **then** p **else** r. This continuation passing style corresponds directly to a view of algebraic computations as 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 [7] 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 [16] 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 [22].

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

In previous work [12], 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.

We say that the data flow is dynamic if the value passed to an operation can depend on the results of prior operations. We say that the 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.

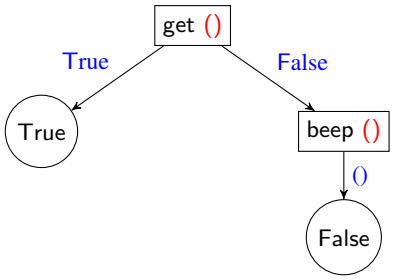
	control flow	**data flow**
idiom	static	static
arrow	static	dynamic
monad	dynamic	dynamic

Table 1. Flow for Idioms, Arrows, and Monads

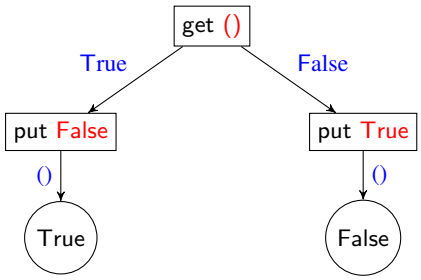
Now we consider how this characterisation relates to abstract computation trees. First let us define some effect signatures:

$$\mathsf{GB} = \{\mathsf{get} : 1 \to \mathsf{Bool},\ \mathsf{beep} : 1 \to 1\} \qquad \mathsf{GP} = \{\mathsf{get} : 1 \to \mathsf{Bool},\ \mathsf{put} : \mathsf{Bool} \to 1\}$$

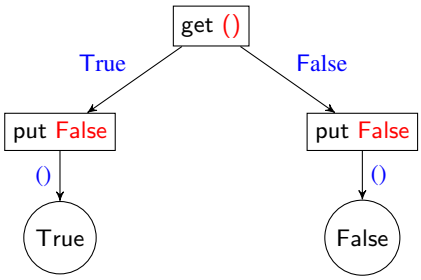
Monadic trees and abstract computation trees are the same thing, as monads impose no restrictions on dependencies. For example, the CPS term, $\mathsf{noisey} = \mathsf{get}(\mathsf{True}, \mathsf{beep}(\mathsf{False}))$ represents monad tree:



Arrow trees are those monadic trees for which the 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, $\mathsf{flip} = \mathsf{get}(\mathsf{put}_{\mathsf{False}}(\mathsf{True}), \mathsf{put}_{\mathsf{True}}(\mathsf{False}))$ 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, $\mathsf{reset} = \mathsf{get}(\mathsf{put}_{\mathsf{False}}(\mathsf{True}), \mathsf{put}_{\mathsf{False}}(\mathsf{False}))$ represents the idiom tree:



Remark 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.

Remark 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.

Remark 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 the memory is dynamic if the values at the leaves depend on the results of prior computations and static if it does not. Idiom, arrow, and monad computations all have dynamic memory. One might conceive of distinguishing other kinds of effectful computation that have static memory. The leaves of such computations 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 by considering computations with unit return type paired up with the constant return value.

An idiom tree that has static memory flow is a monoid tree. It amounts to a free monoid. For example, the CPS term $\mathsf{reset}' = \mathsf{get}(\mathsf{put}_{\mathsf{False}}(\mathsf{True}), \mathsf{put}_{\mathsf{False}}(\mathsf{True}))$ has the monoid tree:

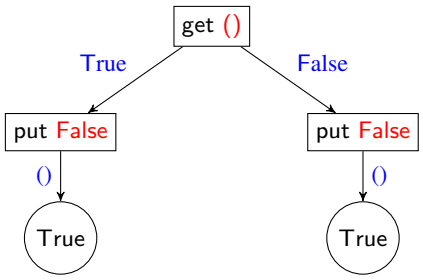


Figure 2.2 illustrates the decomposition of each of our example trees into control flow, data flow, and memory flow. Dynamic flow is represented as a tree. Static flow is represented as a list.

2.3 Flow Effects

In order to distinguish static an 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 foundation for constructing a calculus of effects and handlers for idioms, arrows, and monads, we start with the λ_{eff}-calculus [7], which provides an effect type system and operational semantics for standard monadic algebraic effects and effect handlers. 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 sub-language allows us to write abstract computations over an effect signature. We deviate slightly from our previous presentation [7]. 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:

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

Following Levy's call-by-push-value [9], 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.

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 effects 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 is a collection of operation signatures $\{\mathrm{op}_i : A_i \to B_i\}_i$. Type environments ($\Gamma$) are standard.

The syntax of terms is as follows:

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

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 2. 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 3.

Syntactic Sugar

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

Examples Here are the four example abstract computations from the introduction, written as λ_{eff} terms.

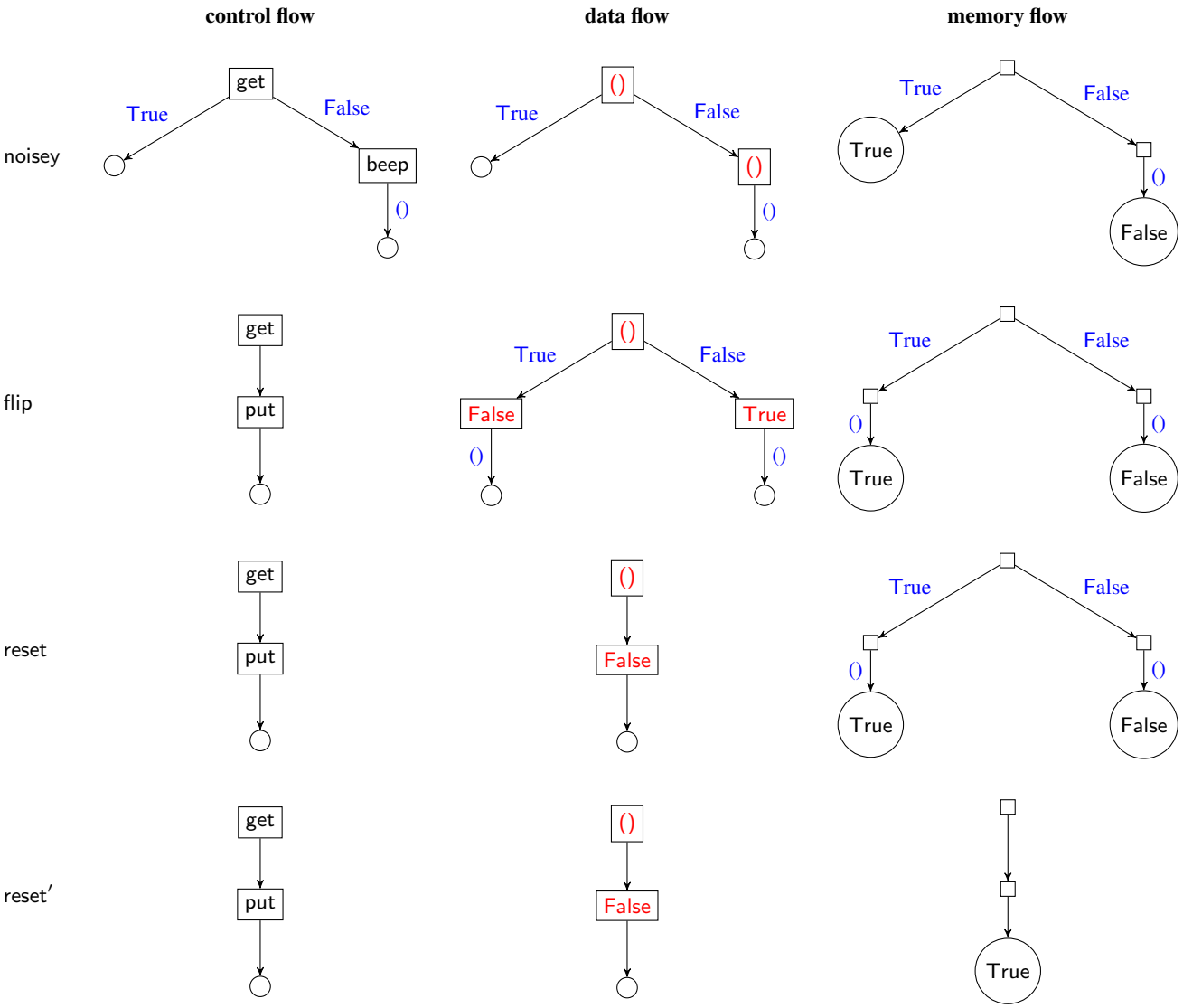


Figure 1. Decomposing Computation Trees

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

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\textsc{Inj}_i\ \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \textsc{Thunk}\ \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : \{C\}_E}$$

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

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

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

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

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

Figure 2. Typing Rules for Core λ_{eff}

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

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

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

Figure 3. Operational Semantics for Core λ_{eff}

$$
\begin{array}{l}
\mathsf{noisey} \ : \ \{\,[\mathsf{Bool}]\,\}_{\mathsf{GB}} \\
\mathsf{noisey} = \{\mathsf{let}\ x \leftarrow \mathsf{get}()\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{return}\ x \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{beep}();\mathsf{return}\ x\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{flip} \ : \ \{\,[\mathsf{Bool}]\,\}_{\mathsf{GP}} \\
\mathsf{flip} = \{\mathsf{let}\ x \leftarrow \mathsf{get}()\ \mathsf{in}\ \mathsf{let}\ y \leftarrow \neg x\ \mathsf{in}\ \mathsf{put}(y);\mathsf{return}\ x\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{reset} \ : \ \{\,[\mathsf{Bool}]\,\}_{\mathsf{GP}} \\
\mathsf{reset} = \{\mathsf{let}\ x \leftarrow \mathsf{get}()\ \mathsf{in}\ \mathsf{put}(\mathsf{False});\mathsf{return}\ x\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{reset}' \ : \ \{\,[\mathsf{Bool}]\,\}_{\mathsf{GP}} \\
\mathsf{reset}' = \{\mathsf{let}\ x \leftarrow \mathsf{get}()\ \mathsf{in}\ \mathsf{put}(\mathsf{False});\mathsf{return}\ \mathsf{True}\}
\end{array}
$$

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

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

Handling

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

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 $\mathsf{return}\ x \mapsto M$ 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:

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

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

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

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.

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

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

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

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

Figure 4. Typing Rules for Handlers

Example As a simple example, consider a handler for boolean state:

$$
\begin{array}{rl}
H_{\mathsf{GP}} = \mathsf{return}\ x & \mapsto \lambda s.\mathsf{return}\ x \\
\mathsf{get}\ ()\ k & \mapsto \lambda s.k\ s\ s \\
\mathsf{put}\ s\ k & \mapsto \lambda s'.k\ ()\ s
\end{array}
$$

We can apply H_{GP} to flip, for instance:

$$\mathsf{handle}\ \mathsf{flip}!\ \mathsf{with}\ H_{\mathsf{GP}}$$

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

Remark 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:

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

3.3 Effect Forwarding

In our prior work [7], 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 allows other operations, which are forwarded to be handled by an outer handler. The big 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 handlers becomes:

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

The only change to the original rule is that the input effects are now $E' \oplus E$ 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 just need to extend the action of a handler to apply to operations without an operation clause as

$$(\mathit{handle}.[\,])\ \textsf{handle}\ (\textsf{return}\ V)\ \textsf{with}\ H \longrightarrow H(\textsf{return}, V)$$
$$(\mathit{handle}.\mathrm{op})\quad \textsf{handle}\ D[\mathrm{op}\ V]\ \textsf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\lambda z.\textsf{handle}\ D[\textsf{return}\ z]\ \textsf{with}\ H\})$$

$$\text{(delimited computation contexts)}\quad D ::= [\,] \mid D\ V \mid \textsf{let}\ x \leftarrow D\ \textsf{in}\ N$$
$$\text{(evaluation contexts)}\quad E ::= [\,] \mid E\ V \mid \textsf{let}\ x \leftarrow E\ \textsf{in}\ N \mid \textsf{handle}\ E\ \textsf{with}\ H$$

Figure 5. Operational Semantics for Handlers

follows:

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

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. The grammar of types is as follows:

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

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.

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

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

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

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

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 rule for returning a value activates all variables in the value.

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

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.

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

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

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

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). Similarly, we introduce a special op M construct.

$$\textsc{OpC}^\star\ \frac{\mathrm{d} \in E \qquad (\mathrm{op} : A \to B) \in E \qquad E' \subseteq \{\mathrm{c}, \mathrm{d}\} \qquad \Gamma^\star \vdash_{E'} M : [A]}{\Gamma \vdash_E \mathrm{op}\ M : [B]}$$

The rule for this construct allows operation parameter values to be computed from any variables in the type environment using an arbitrary pure computation. Finally, we include a special rule for forcing, 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. It provides dynamic data flow in addition to dynamic control flow.

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}.

Remark The reason why the data flow effect appears in the ($\textsc{Force}^\star$) rule is because it allows unrestricted flow by activating the type environment.

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 contexts for computing inside returned values and operation parameters, and the rules $(ret.ret)$ and $(op.ret)$ for converting the corresponding computations into values, once they have finished computing.

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

$$\textsc{Var}^\star\ \frac{(x^\star : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\textsc{Inj}_i\ \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \textsc{Thunk}\ \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : \{C\}_E}$$

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

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

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

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

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

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

$$\textsc{ReturnC}^\star\ \frac{E' \subseteq \{\mathrm{c}, \mathrm{d}\} \qquad \Gamma^\star \vdash_{E'} M : [A]}{\Gamma \vdash_E \mathbf{return}\, M : [A]} \qquad \textsc{OpC}^\star\ \frac{\mathrm{d} \in E \qquad (\mathrm{op} : A \to B) \in E \qquad E' \subseteq \{\mathrm{c}, \mathrm{d}\} \qquad \Gamma^\star \vdash_{E'} M : [A]}{\Gamma \vdash_E \mathrm{op}\, M : [B]}$$

Figure 6. Typing Rules for Core λ_{flow}

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

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

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

Figure 7. Operational Semantics for Core λ_{flow}

Examples In λ_{flow}, our examples become:

$$\begin{array}{l}
\mathsf{noisey} : \{[\mathsf{Bool}]\}_{\mathsf{GB} \cup \{c,d\}} \\
\mathsf{noisey} = \{\mathbf{let}\ x \leftarrow \mathsf{get}()\ \mathbf{in}\ \{\mathbf{if}\ x\ \mathbf{then}\ \mathbf{return}\ x \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{beep}();\mathbf{return}\ x\}!\}
\end{array}$$

$$\begin{array}{l}
\mathsf{flip} : \{[\mathsf{Bool}]\}_{\mathsf{GP} \cup \{d\}} \\
\mathsf{flip} = \{\mathbf{let}\ x \leftarrow \mathsf{get}()\ \mathbf{in}\ \mathsf{put}(\neg x);\mathbf{return}\ x\}
\end{array}$$

$$\begin{array}{l}
\mathsf{reset} : \{[\mathsf{Bool}]\}_{\mathsf{GP}} \\
\mathsf{reset} = \{\mathbf{let}\ x \leftarrow \mathsf{get}()\ \mathbf{in}\ \mathsf{put}(\mathsf{False});\mathbf{return}\ x\}
\end{array}$$

$$\begin{array}{l}
\mathsf{reset}' : \{[\mathsf{Bool}]\}_{\mathsf{GP}} \\
\mathsf{reset}' = \{\mathbf{let}\ x \leftarrow \mathsf{get}()\ \mathbf{in}\ \mathsf{put}(\mathsf{False});\mathbf{return}\ \mathsf{True}\}
\end{array}$$

The type signatures of the first two examples have been augmented with flow effects. The conditional in noisey has had to be thunked in order to bring x into scope. The negation in flip is now directly inside the parameter to put, which is well-typed because negation is pure. Note that the λ_{eff} version of flip does not type check in λ_{flow} because x is not in scope in the negation.

5. Handling Flow

As there are two flow effects (c and d), there are four possible kinds of computation we might try to handle: monads ($\{\mathrm{c}, \mathrm{d}\}$), arrows ($\{\mathrm{d}\}$), idioms ($\emptyset$), and something strange ($\{\mathrm{c}\}$). 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 $\emptyset \subseteq \{\mathrm{d}\} \subseteq \{\mathrm{c}, \mathrm{d}\}$, 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 8. The operational semantics is given in Figure 9. We will now describe in detail the design of the different kinds of handler. We first note that 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. The operational semantics is unchanged. We annotate monad handlers and monad handler types with a T subscript.

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

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

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

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

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

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

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

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

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

Figure 8. Typing Rules for λ_{flow} Handlers

$$\begin{array}{ll} (\mathit{handle}_{\mathsf{T}}.[\,]) & \mathsf{handle}_{\mathsf{T}}\ (\mathsf{return}\ V)\ \mathsf{with}\ H \longrightarrow H(\mathsf{return}, V) \\ (\mathit{handle}_{\mathsf{T}}.\mathrm{op}) & \mathsf{handle}_{\mathsf{T}}\ D[\mathrm{op}\ V]\ \mathsf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\lambda x.\mathsf{handle}\ D[\mathsf{return}\ x]\ \mathsf{with}\ H\}) \\ (\mathit{handle}_{\rightsquigarrow}.[\,]) & \mathsf{handle}_{\rightsquigarrow}\ (\lambda\Delta.\mathsf{return}\ V)\ \mathsf{with}\ H \longrightarrow H(\mathsf{return}, \{\lambda\Delta.\mathsf{return}\ V\}) \\ (\mathit{handle}_{\rightsquigarrow}.\mathrm{op}) & \mathsf{handle}_{\rightsquigarrow}\ (\lambda\Delta.D[\mathrm{op}\ V])\ \mathsf{with}\ H \longrightarrow H(\mathrm{op}, \{\lambda\Delta.\mathsf{return}\ V\}, \{\mathsf{handle}_{\rightsquigarrow}\ (\lambda(\Delta, x).D[\mathsf{return}\ x])\ \mathsf{with}\ H\}) \\ (\mathit{handle}_{\mathsf{I}}.[\,]) & \mathsf{handle}_{\mathsf{I}}\ (\lambda\Delta.\mathsf{return}\ V)\ \mathsf{with}\ H \longrightarrow H(\mathsf{return}, \{\lambda\Delta.\mathsf{return}\ V\}) \\ (\mathit{handle}_{\mathsf{I}}.\mathrm{op}) & \mathsf{handle}_{\mathsf{I}}\ (\lambda\Delta.D[\mathrm{op}\ V])\ \mathsf{with}\ H \longrightarrow H(\mathrm{op}, V, \{\mathsf{handle}_{\mathsf{I}}\ (\lambda(\Delta, x).D[\mathsf{return}\ x])\ \mathsf{with}\ H\}) \end{array}$$

$$\begin{array}{ll} \text{(delimited computation contexts)} & D ::= [\,] \mid D\ V \mid \mathsf{let}\ x \leftarrow D\ \mathsf{in}\ N \\ \text{(evaluation contexts)} & E ::= [\,] \mid E\ V \mid \mathsf{let}\ x \leftarrow E\ \mathsf{in}\ N \\ & \quad \mid \mathsf{handle}_{\mathsf{T}}\ E\ \mathsf{with}\ H \mid \mathsf{handle}_{\rightsquigarrow}\ (\lambda\Delta.E)\ \mathsf{with}\ H \mid \mathsf{handle}_{\mathsf{I}}\ (\lambda\Delta.E)\ \mathsf{with}\ H \end{array}$$

Figure 9. Operational Semantics for λ_{flow} Handlers

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 in such a way that the input need not be provided up-front. For instance, a state transformer [17] of type

$$\{\mathsf{Bool} \to [A]\}_{\{\mathrm{c},\mathrm{d}\}} \to [\{\mathsf{Bool} \to [B]\}_{\{\mathrm{c},\mathrm{d}\}}]$$

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.

Arrow handler types have the following shape:

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

where Γ is a type operator. The idea is that this maps a computation M of type $[A]$ parameterised by a context Δ to a computation of type $G\,\Delta$.

We overload $\Delta = x_1 : A_1, \ldots, x_n : A_n$ to mean: a type environment consisting entirely of inactive variables; the left nested product type $1 \times A_1 \times \cdots \times A_n$; and the left-nested tuple type $((), x_1, \ldots, x_n)$. We write $\lambda\Delta.M$ as sugar for $\lambda z.\mathbf{split}(z, z.x_n. \ldots \mathbf{split}(z, z.x_1.M))$, where z is a fresh variable.

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 Δ. As the handled computation has an input Δ, it is written as a lambda $\lambda\Delta.M$. The return type of the conclusion is $G\,\Delta$.

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. We let X range over type variables and use a type variable here in order to ensure that the handler is parametric in the input type. This is crucial, as it allows us to manually thread the context through computations. 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 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 (ARROWHANDLER) rule prevents arrow handlers from being applied to computations with dynamic control flow. This is necessary in order to ensure that closed terms do not get stuck. For instance, this constraint disallows stuck terms of the form:

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

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*$_{\rightsquigarrow}$.[]), the value is a function of the input. When handling an operation (*handle*$_{\rightsquigarrow}$.op), the parameter is a function of the input, and the continuation extends the context with the return value of the operation.

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

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

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.

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

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

5.4 Example: Parser Combinators

We now illustrate λ_{flow} with a small example combining monad handlers and idiom handlers. In order to make our example slightly realistic, we assume λ_{flow} has been extended with pattern-matching, polymorphic type variables X, and a polymorphic list type $\mathsf{List}\ X$. Let us imagine we wish to write code to parse a list of characters. To keep things simple, we assume an effect signature for parsing containing only two operations:

$$\mathsf{Parse} = \{\mathsf{any} : 1 \to \mathsf{Char}, \mathsf{char} : \mathsf{Char} \to \mathsf{Char}\}$$

The any operation parses any character and the char operation parses a specific character.

We also define other effect signatures for reading characters, state, and failure:

$$\begin{aligned}&\mathsf{Read} = \{\mathsf{getc} : 1 \to \mathsf{char}\}\\ &\mathsf{State}\ X = \{\mathsf{get} : 1 \to X, \mathsf{put} : X \to 1\}\\ &\mathsf{Fail} = \{\mathsf{fail} : \forall X.1 \to X\}\end{aligned}$$

First let us define an idiom handler for parsing.

```
parse : {{X}_{Parse∪Fail} → X}_{Read∪Fail∪{c,d}}
parse m = {
  (handle_I (λ().m!) with
    return x ↦ x!
    any () k ↦ λz.let c ← getc () in k! (z, c)
    char c k ↦ λz.let c' ← getc () in
                  let b ← (c = c') in {if b then k! (z, c)
                                      else fail ()}!
    fail p k ↦ λz.fail ()) ()
}
```

The handler maps the parsing operations any and char to the getc operation. In addition, it forwards failure. Now let us define a handler for reading characters from a list.

```
read : {List char → {X}_{Read∪{c,d}} → X}_{State (List char)∪Fail{c,d}}
read cs m = {
  handle_T m! with
    return x ↦ return x
    getc () k ↦ let cs ← get () in
                 {case cs of Nil ↦ fail ()
                             Cons c cs ↦ put cs; return c}!
}
```

This handler interprets getc in terms of state and failure. We handle failure with a standard option type $(1 + X)$.

$$\begin{array}{l}\mathsf{option} : \{\{X\}_{\mathsf{Fail}\cup\{\mathrm{c},\mathrm{d}\}} \to 1 + X\}_{\{\mathrm{c},\mathrm{d}\}}\\ \mathsf{option}\ m = \{\\ \quad \mathbf{handle}_{\mathsf{T}}\ m!\ \mathbf{with}\\ \quad\quad \mathbf{return}\ x \mapsto \mathbf{return}\ (\mathbf{inj}_2\ x)\\ \quad\quad \mathsf{fail}\ ()\ k \mapsto \mathbf{return}\ (\mathbf{inj}_1\ ())\\ \}\end{array}$$

Finally, we implement a handler for state that throws away the final state.

$$\begin{array}{l}\mathsf{state} : \{\{X\}_{\mathsf{State}\ S\cup\{\mathrm{c},\mathrm{d}\}} \to S \to X\}_{\{\mathrm{c},\mathrm{d}\}}\\ \mathsf{state}\ m = \{\\ \quad \mathbf{handle}_{\mathsf{T}}\ m!\ \mathbf{with}\\ \quad\quad \mathbf{return}\ x \mapsto \lambda s.\mathbf{return}\ x\\ \quad\quad \mathsf{get}\ ()\ k \mapsto \lambda s.k\ s\ s\\ \quad\quad \mathsf{put}\ s\ k \mapsto \lambda s'.k\ ()\ s\\ \}\end{array}$$

Having defined these four handlers, the idea is that we can now compose them together. In fact, we would need to add a little more to our language for this to work completely smoothly. In particular, we would need our handlers to support forwarding in order to avoid having to write dummy forwarding clauses in the monadic handlers, and we would also need effect polymorphism, or effect subtyping in order to allow the effect handlers to be used in the presence of different ambient effects. Developing a practical language or library that supports such features, along with a more convenient source syntax, is left as future work. (Our Haskell library does support forwarding and effect polymorphism, so it is straightforward to implement all of the monad handlers and compose them using our library. Of course, it does not support idiom handlers, though.)

5.5 Forwarding for Arrow and Idiom Handlers

As the continuation of an operation by 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.

5.6 Correctness of λ_{flow}

THEOREM 1 (Type Soundness). *If $E \subseteq \{\mathrm{c},\mathrm{d}\}$ and $\vdash_E M : [A]$, then either there exists N such that $M \longrightarrow N$ or there exists V such that $M = \mathbf{return}\ V$.*

The proof of type soundness is pretty standard and does not deviate significantly from that for λ_{eff}. However, there are some subtleties arising from flow effects. The usual form of subject reduction (preservation of typing under reduction) does not hold for λ_{flow}. The reason is that applying the $(\beta.\{\})$ reduction rule sometimes yields a term that can only be typed if some of the inactivate variables in the type environment are activated. For instance, we have:

$$x : A \vdash_{\mathrm{c},\mathrm{d}} \{\mathbf{return}\ x\}! : [A]$$

and:

$$\{\mathbf{return}\ x\}! \longrightarrow \mathbf{return}\ x$$

but the following judgement is invalid:

$$x : A \vdash_{\mathrm{c},\mathrm{d}} \mathbf{return}\ x : [A]$$

This is not a serious problem because we are primarily interested in reduction on closed terms.

There are a number of ways of restoring subject reduction, however. One way is to add a non-syntactic rule for activating variables along the lines of:

$$\frac{\mathrm{c},\mathrm{d} \in E \qquad \Gamma^\star \vdash_E M : C}{\Gamma \vdash_E M : C}$$

which would effectively make the (Force*) rule redundant. Another way of restoring subject reduction is to modify all of the binding rules such that the bound variable is immediately activated in the case that c and d are in the current effect. What we choose to do instead is to use a weaker form of subject reduction.

LEMMA 2 (Weak Subject Reduction). *If $\Gamma \vdash_E M : C$, and $M \longrightarrow N$, then $\Gamma^\star \vdash_E N : C$.*

The proof is a straightforward inductive argument using the following lemma:

LEMMA 3. *If $\Gamma \vdash_E M : C$ then $\Gamma^\star \vdash_E M : C$.*

as well as a suitable substitution lemma.

The proof of termination for λ_{eff} readily adapts to λ_{flow}.

THEOREM 4 (Termination). *If $\Gamma \vdash_E M : C$, then reduction on M terminates.*

The key property is that (*handle*.op)-reduction is a form of structural recursion, so is guaranteed to terminate.

6. Embedding Arrow Calculus into λ_{flow}

Thusfar, we have claimed that λ_{flow} is a meta programming language for idioms, arrows, and monads. The correspondence with λ_{eff} is clear, but we have not yet shown that it corresponds in any way with existing calculi for idioms and arrows. The arrow calculus [11] is a meta language for programming with arrows. In previous work [12] we used variations on the arrow calculus to elucidate the relationship between idiom, arrow, and monad computations. In this section we show that there is a straightforward embedding of the arrow calculus into λ_{flow}, and furthermore this embedding extends to cover variants of the arrow calculus that support idioms and monads.

6.1 The Arrow Calculus

The arrow calculus is an extension of the simply-typed lambda calculus. Following Lindley et al [11] we present it as a simply-typed equational theory. The typing rules and equational laws are given in Figure 10. Value terms are ranged over by L, M, N. Computation terms are ranged over by P, Q.

There are two typing judgements. The judgement $\Gamma \vdash M : A$ is the standard one of the simply-typed lambda calculus. The judgement $\Gamma; \Delta \vdash P\ !\ A$ is for arrow computations. The type environment is separated into two parts: the variables in Γ are accessible everywhere, whereas those in Δ are restricted. Variables in Γ correspond to active variables, and those in Δ to inactive variables. The term P is an arrow computation term. The type A is the output type of the computation.

We assume a fixed signature Σ of arrow operations. An arrow abstraction $\lambda^\bullet x.P$ abstracts over an arrow computation P. The variable x is bound in the Δ environment. In an arrow application $L \bullet M$, only the variables of Γ are accessible to L. When returning a value, both type environments are merged. Let bindings are bound in the Δ type environment.

Typing Rules

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

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}$$

$$\textsc{Fst}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{Fst}\ M : A} \qquad \textsc{Snd}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{Fst}\ M : A}$$

$$\textsc{Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\textsc{ArrowAbs}\ \frac{\Gamma; x : A \vdash P\ !\ B}{\Gamma \vdash \lambda^{\bullet} x.P : A \rightsquigarrow B} \qquad \textsc{Op}\ \frac{\mathrm{op} : A \rightsquigarrow B \in \Sigma}{\Gamma \vdash \mathrm{op} : A \rightsquigarrow B}$$

$\boxed{\Gamma; \Delta \vdash M\ !\ A}$

$$\textsc{ArrowApp}\ \frac{\Gamma \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \bullet M\ !\ B} \qquad \textsc{Return}\ \frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathsf{return}\ M\ !\ A}$$

$$\textsc{Let}\ \frac{\Gamma; \Delta \vdash P\ !\ A \qquad \Gamma; \Delta, x : A \vdash Q\ !\ B}{\Gamma; \Delta \vdash \mathsf{let}\ x \leftarrow P\ \mathsf{in}\ Q\ !\ B}$$

Laws

$$\begin{array}{ll}
(\beta_1^{\times}) & \mathsf{Fst}\ (M, N) = M \\
(\beta_2^{\times}) & \mathsf{Snd}\ (M, N) = N \\
(\eta^{\times}) & (\mathsf{Fst}\ L, \mathsf{Snd}\ L) = L \\
(\beta^{\to}) & (\lambda x.N)\ M = N[M/x] \\
(\eta^{\to}) & \lambda x.(L\ x) = L \\
(\eta^{1}) & () = M \\
(\beta^{\rightsquigarrow}) & (\lambda^{\bullet} x.Q) \bullet M = Q[M/x] \\
(\eta^{\rightsquigarrow}) & \lambda^{\bullet} x.(L \bullet x) = L \\
\text{(left)} & \mathsf{let}\ x \leftarrow \mathsf{return}\ M\ \mathsf{in}\ Q = Q[M/x] \\
\text{(right)} & \mathsf{let}\ x \leftarrow P\ \mathsf{in}\ \mathsf{return}\ x = P \\
\text{(assoc)} & \mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow P\ \mathsf{in}\ Q)\ \mathsf{in}\ R = \mathsf{let}\ x \leftarrow P\ \mathsf{in}\ (\mathsf{let}\ y \leftarrow Q\ \mathsf{in}\ R)
\end{array}$$

Figure 10. The Arrow Calculus

The equational laws are standard β and η-laws along with a commuting conversion for computations.

6.2 The Embedding

Given $\Sigma = \{\mathrm{op}_i : A_i \rightsquigarrow B_i\}_i$, we define an embedding $[\![-]\!]$ of the arrow calculus into λ_{flow}. The translation on types is as follows:

$$\begin{aligned}
[\![1]\!] &= \{\,\texttt{[1]}\,\}_{\{\mathrm{c},\mathrm{d}\}} \\
[\![A \times B]\!] &= \{[\![A]\!] \times [\![B]\!]\}_{\{\mathrm{c},\mathrm{d}\}} \\
[\![A \to B]\!] &= \{[\![A]\!] \to \texttt{[}[\![B]\!]\texttt{]}\}_{\{\mathrm{c},\mathrm{d}\}} \\
[\![A \rightsquigarrow B]\!] &= \{[\![A]\!] \to \texttt{[}[\![B]\!]\texttt{]}\}_{\{\mathrm{d}\} \cup E}
\end{aligned}$$

where

$$E = \{\mathrm{op}_i : [\![A_i]\!] \to [\![B_i]\!]\}_i$$

The translation on type environments is defined pointwise on the types:

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] = x_1 : [\![A_1]\!], \ldots, x_n : [\![A_n]\!]$$

The translation on value terms is as follows:

$$\begin{aligned}
[\![x]\!] &= x \\
[\![()]\!] &= \{\mathsf{return}\ ()\} \\
[\![(M, N)]\!] &= \{\mathsf{return}\ ([\![M]\!], [\![N]\!])\} \\
[\![\mathsf{Fst}\ M]\!] &= \{\mathsf{let}\ z \leftarrow M!\ \mathsf{in}\ \{\mathsf{split}(z, x.y.\mathsf{return}\ x)\}!\} \\
[\![\mathsf{Snd}\ M]\!] &= \{\mathsf{let}\ z \leftarrow M!\ \mathsf{in}\ \{\mathsf{split}(z, x.y.\mathsf{return}\ y)\}!\} \\
[\![()]\!] &= \{\mathsf{return}\ ()\} \\
[\![\lambda x.M]\!] &= \{\lambda x.\mathsf{return}\ [\![M]\!]\} \\
[\![M\ N]\!] &= \{[\![M]\!]!\ [\![N]\!]\} \\
[\![\lambda^{\bullet} x.P]\!] &= \{\lambda x.[\![P]\!]\} \\
[\![\mathrm{op}]\!] &= \{\lambda x.\mathrm{op}\ x\}
\end{aligned}$$

The translation on computation terms is as follows:

$$\begin{aligned}
[\![L \bullet M]\!] &= [\![L]\!]!\ [\![M]\!] \\
[\![\mathsf{return}\ M]\!] &= \mathsf{return}\ [\![M]\!] \\
[\![\mathsf{let}\ x \leftarrow P\ \mathsf{in}\ Q]\!] &= \mathsf{let}\ x \leftarrow [\![P]\!]\ \mathsf{in}\ [\![Q]\!]
\end{aligned}$$

THEOREM 5 (Type Soundness).

- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.
- *If* $\Gamma; \Delta \vdash P\ !\ A$ *then* $[\![\Gamma]\!]^{\star}, [\![\Delta]\!] \vdash_E [\![P]\!] : \mathcal{U}[\![A]\!]\mathcal{J}$.

We believe that the operational semantics respects equality in the arrow calculus via the embedding, but we have not yet proved this. It may make more sense in future to relate a reduction-oriented variant of arrow calculus with λ_{flow}.

6.3 Higher Order Arrows

In order to support monadic computations in the arrow calculus, one simply adds a relaxed form of arrow application in which the argument has access to both type environments. Its typing rule is as follows:

$$\textsc{HArrowApp}\ \frac{\Gamma, \Delta \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \star M\ !\ B}$$

and it comes with the additional laws:

$$\begin{array}{ll}
(\beta^{app}) & (\lambda^{\bullet} x.Q) \star M = Q[x := M] \\
(\eta^{app}) & \lambda^{\bullet} x.(L \star x) = L
\end{array}$$

The translation of higher-order arrow application into λ_{flow} is identical to the translation of standard arrow application.

$$[\![L \star M]\!] = [\![L]\!]!\ [\![M]\!]$$

When working with higher-order arrows, the translation on arrow types is amended to add the c effect.

$$[\![A \rightsquigarrow B]\!] = \{[\![A]\!] \to [\![B]\!]\}_{\{\mathrm{c},\mathrm{d}\} \cup E}$$

6.4 Static Arrows

In order to support idiom computations in the arrow calculus, one adds a special **run** operator that coerces any value of type $A \rightsquigarrow B$ into a computation of type $A \to B$. Its typing rule is as follows:

$$\frac{\Gamma \vdash L : A \rightsquigarrow B}{\Gamma; \Delta \vdash \mathsf{run}\ L\ !\ A \to B}$$

and it comes with the additional laws:

$$(ob_1) \qquad L \bullet M = \mathsf{let}\ f \leftarrow \mathsf{run}\ L\ \mathsf{in}\ \mathsf{return}\ f\ M$$
$$(ob_2) \qquad \mathsf{run}\ (\lambda^\bullet x.\mathsf{return}\ M) = \mathsf{return}\ \lambda x.M$$
$$(ob_3) \quad \mathsf{run}(\lambda^\bullet x.\mathsf{let}\ y \leftarrow p\ \mathsf{in}\ Q) = \mathsf{let}\ y \leftarrow P\ \mathsf{in}\ \mathsf{let}\ f \leftarrow \mathsf{run}(\lambda^\bullet(x,y).Q)\ \mathsf{in}\ \mathsf{return}\ \lambda x.f\ (x,y)$$

The translation of **run** into λ_{flow} essentially requires us to simulate an upcast on the effects of the body of a function. We do so using a straightforward identity handler.

$$[\![\mathsf{run}\ L]\!] = \mathsf{handle}_{\mathsf{I}}\ \lambda x.[\![L]\!]!\ x\ \mathsf{with}\ \mathsf{return}\ y \mapsto \mathsf{return}\ y$$

When working with static arrows, the translation on arrow types is amended to remove the d effect.

$$[\![A \rightsquigarrow B]\!] = \{[\![A]\!] \to [\![B]\!]\}_E$$

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 [8] have implemented a similar library for Haskell, and Brady [3] has implemented 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 [13, 14]. None of these systems support algebraic effects or handlers for idioms or arrows.

Capriotti and Kaposi explore free idioms [4], and their Haskell implementations. Free idioms correspond to abstract idiom computations. Yallop's thesis [23, Chapter 2] provides an in-depth analysis of idioms, arrows, and monads, expanding on the work of Lindley et al [12], and characterising the normal forms for idioms and arrows. We have implemented both free monad and free arrow constructions in Haskell [10] directly inspired by the normal forms of Yallop.

Petricek and Syme [18] describe a novel use of F# computation expression syntax to write idiom computations using let notation. Their work 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.

We believe it should be possible to 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.

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

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

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

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

[19] 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.

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

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

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

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