Embedding effect systems in Haskell

Dominic Orchard Tomas Petricek
Computer Laboratory, University of Cambridge
firstname.lastname@cl.cam.ac.uk

Abstract

Monads are now an everyday tool in functional programming for abstracting and delimiting effects. The link between monads and effect systems is well-known, but in their typical use, monads provide a much more coarse-grained view of effects. Effect systems capture fine-grained information about the effects, but monads provide only a binary view: *effectful* or *pure*.

Recent theoretical work has unified fine-grained effect systems with monads using a monad-like structure indexed by a monoid of effect annotations (called parametric effect monads). This aligns the power of monads with the power of effect systems.

This paper leverages recent advances in Haskell's type system (as provided by GHC) to embed this approach in Haskell, providing user-programmable effect systems. We explore a number of practical examples that make Haskell even better and safer for effectful programming. Along the way, we relate the examples to other concepts, such as Haskell's implicit parameters and *coeffects*.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Type Structure

Keywords effect systems; parametric effect monads; type systems

1. Introduction

Side effects are an essential part of programming. There are many reasoning and programming techniques for working with them. Two well-known approaches in functional programming are effect systems, for analysis, and monads, for encapsulating and delimiting effects. Monads have a number of benefits. They provide a simple programming abstraction, or a *design pattern*, for encapsulating functionality. They also delimit the scope of effects, showing which parts of a program are pure and which parts are impure. However, compared to effect systems, monads have two limitations.

1). Granularity. The information provided by monadic typing is limited. We can look at the type of an expression and see, for example, that it has state effects if it uses the ST monad, but we no nothing more about the effects from the type; the analysis provided by standard monadic typing provides only binary information.

In contrast, effect systems annotate computations with finer-grained information. For example, stateful computations can be annotated with sets of triples of memory locations, types, and effect markers $\sigma \in \{\mathsf{update}, \mathsf{read}, \mathsf{write}\}$. This provides information on how state is affected, without requiring the code to be examined.

One solution for improving granularity is to define a type class for every effectful operation, with a type class constraint over a polymorphic monadic type [15]. However, this restricts an effect analysis to sets with union and ordering by subsets.

2). Compositionality. Monads do not compose well. In Haskell, we often have to refactor monadic code or add additional bookkeeping (for example, insert lifting when using monad transformers) to compose different notions of effect. In contrast, effect systems which track information about different notions of effect can be more easily composed.

The recent notion of *parametric effect monads* [12] (also called *indexed monads* [19]) provides a solution to *granularity*, and a partial solution to *compositionality*. Parametric effect monads amplify the monadic approach with *effect indices* (annotations) which describe in more detail the effects of a computation. This effect information has the structure of a monoid $(F, \bullet, I)$, where I is the annotation of pure computations and $\bullet$ composes effect information. The monoidal structure adorns the standard monad structure, leading to the operations of a monad having the types:

$$return :: a \to M_I\, a$$
$$(\gg\!=) :: M_F\, a \to (a \to M_G\, b) \to M_{F \bullet G}\, b$$

The indexed data type $M_F A$ may be defined in terms of F, giving a semantic, value-level counterpart to the effect information. This approach thus unifies monads with effect systems. This paper makes the following contributions:

- We encode parametric effect monads in Haskell, using them to embed effect systems (Section 2). This provides a general system for high-granularity effect information and better compositionality for some examples (Sections 5-6). This embedding is *shallow*; we do not require any macros or custom syntax.
- We leverage recent additions to the Haskell type system to make it possible (and practical) to track fine-grained information about effects in the type system, for example using *type-level sets* (Section 3). In particular, we use *type families* [5], *constraint kinds* [3, 20], GADTs [22], *data kinds and kind polymorphism* [25], and *closed type families* [7].
- A number of practical examples are provided, including effect systems arising from reader, writer, and state monads (Sections 5-6), and for analysing and verifying program properties including computational complexity bounds and completeness of data access patterns (Section 9). We provide a Haskell-friendly explanation of recent theoretical work and show how to use it to improve programming practice.
- We discuss the dual situation of *coeffects* and comonads (Section 8) and the connection of effect and coeffect systems to

Haskell '14, September 4–5, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3041-1/14/09…$15.00.
http://dx.doi.org/10.1145/2633357.2633368

Haskell's *implicit parameters*. Implicit parameters can be seen as an existing coeffect system in Haskell.

The code of this paper is available via Hackage (`cabal install ixmonad`) or at `http://github.com/dorchard/effect-monad`.

In the rest of this section we look at two examples that demonstrate the problems with the current state-of-the-art Haskell programming. The rest of the paper shows that we can do better.

Problem 1 Consider programming stream processors. We define two stateful operations, $writeS$ for writing to an output stream (modelled as a list) and $incC$ for counting these writes:

$$writeS :: (Monad\ m) \Rightarrow [a] \to StateT\ [a]\ m\ ()$$
$$incC :: (Monad\ m, Num\ s) \Rightarrow StateT\ s\ m\ ()$$

We have planned ahead by using the state monad transformer to allow composing states. Thus, an operation that both writes to the output stream and increments the counter can be defined using $lift :: (MonadTrans\ t, Monad\ m) \Rightarrow m\ a \to t\ m\ a$:

$$write :: (Monad\ m) \Rightarrow [a] \to StateT\ [a]\ (StateT\ Int\ m)\ ()$$
$$write\ x = \mathbf{do}\ \{ writeS\ x; lift\ \$\ incC \}$$

In combining the two states, an arbitrary choice is made of which one to lift (the counter state). The following example program

$$hellow = \mathbf{do}\ \{ write\ \texttt{"hello"}; write\ \texttt{" "}; write\ \texttt{"world"} \}$$

can be "run" by providing two initial states (in the correct order):

$$runStateT\ (runStateT\ hellow\ \texttt{""})\ 0$$

evaluating to $(((), \texttt{"hello world"}), 3)$. The type of $hellow$ indicates in which order to supply the initial state arguments and which operations to lift in any future reuses of the state.

Consider writing another function which counts the number of times $hellow$ is run. We reuse $incC$, lifting it to increment an additional state:

$$hellowC = \mathbf{do}\ \{ hellow;\ \ lift\ \$\ lift\ \$\ incC \}$$

Now there are two Int states, so the types provide less guidance on the order to apply arguments. We also see that, for every new state, we have to add more and more *lift* operations, chained together.

The parametric effect monad for state (Section 6) allows definitions of $incC$ and $writeS$ that have precise effect descriptions in their types, written:

$$incC\ \ :: State\ `[\texttt{"count"} :\to Int :! RW]\ ()$$
$$writeS :: [a] \to State\ `[\texttt{"out"} :\to [a] :! RW]\ ()$$

meaning that $incC$ has a read-write effect on a variable `"count"` and $writeS$ has a read-write effect on a variable `"out"`. The two can then be composed, using the usual **do**-notation, as:

$$write :: [a] \to State\ `[\texttt{"count"} :\to Int :! RW,\ \texttt{"out"}\ \ :\to [a] :! RW]\ ()$$
$$write\ x = \mathbf{do}\ \{ writeS\ x; incC \}$$

whose effect information is the union of the effects for $writeS$ and $incC$. Note that we didn't need to use a *lift* operation, and we now also have precise effect information at the type level.

An alternate solution to granularity is to define a type class for each effectful operation parameterised by a result monad type, *e.g.*,

$$\mathbf{class}\ Monad\ m \Rightarrow Output\ a\ m\ \mathbf{where}\ writeS :: [a] \to m\ ()$$
$$\mathbf{class}\ Monad\ m \Rightarrow Counting\ m\ \mathbf{where}\ incC :: m\ ()$$

Suitable instances can be given using monad transformers. This approach provides an effect system via type class constraints, but restricts the effect annotations to sets with union (conjunction of constraints) and ordering of effects by subsets. In this paper, we embed a more general effect system, parameterised by a monoid of effects with a preorder, and show examples leveraging this generality.

$$\mathbf{class}\ Effect\ (m :: k \to * \to *)\ \mathbf{where}$$
$$\quad \mathbf{type}\ Unit\ m :: k$$
$$\quad \mathbf{type}\ Plus\ m\ (f :: k)\ (g :: k) :: k$$
$$\quad \mathbf{type}\ Inv\ m\ (f :: k)\ (g :: k) :: Constraint$$
$$\quad \mathbf{type}\ Inv\ m\ f\ g = ()$$
$$\quad return :: a \to m\ (Unit\ m)\ a$$
$$\quad (\gg\!=) :: Inv\ m\ f\ g \Rightarrow$$
$$\qquad m\ f\ a \to (a \to m\ g\ b) \to m\ (Plus\ m\ f\ g)\ b$$
$$\mathbf{class}\ Subeffect\ (m :: k \to * \to *)\ f\ g\ \mathbf{where}$$
$$\quad sub :: m\ f\ a \to m\ g\ a$$

Figure 1. Parametric effect monad and subeffecting classes

Problem 2 Consider writing a DSL for parallel programming. We want to include the ability to use state, so the underlying implementation uses the state monad everywhere to capture stateful operations. However, we want to statically ensure that a parallel mapping function on lists $parMap$ is only applied to functions with, at most, read-only state effects. The standard monadic approach does not provide any guidance, so we have to resort to other encodings.

With the embedded effect system approach of this paper we can write the following definition for $parMap$:

$$parMap :: (Writes\ f {\sim}`[]) \Rightarrow$$
$$\qquad (a \to State\ f\ b) \to [a] \to State\ f\ [b]$$
$$parMap\ k\ [] = sub\ (return\ [])$$
$$parMap\ k\ (x : xs) = \mathbf{do}\ (y, ys) \leftarrow (k\ x)\ `par`\ (parMap\ k\ xs)$$
$$\qquad\qquad return\ (y : ys)$$

The predicate $Writes\ f {\sim}`[]$ on effect information constrains the computation to be free from write effects. The par combinator provides the parallel behaviour.

2. Parametric effect monads

While monads are defined over parametric types of kind $m :: * \to *$, parametric effect monads are defined over types of kind $m :: k \to * \to *$ with an additional parameter of some kind k of effect types. We define parametric effect monads by replacing the usual $Monad$ type class with the *Effect* class, which has the same operations, but with the effect-parameterisation described in the introduction.

Figure 1 gives the Haskell definition which uses type families, polymorphic kinds, and constraint kinds. $Plus\ m$ declares a binary type family for composing effect annotations (of kind k) when sequentially composing computations with bind ($\gg\!=$). $Unit\ m$ is a nullary type family computing the 'unit' annotation for the trivial (or pure) effect, arising from $return$. The idea is that $Plus\ m$ and $Unit\ m$ form a monoid which is shown by the parametric effect monad axioms (see below).

The Inv family is a *constraint family* [3, 20] (*i.e.*, constraint-kinded type family) which can be used to restrict effect parameters in instances of *Effect*. The default is the empty constraint.

do-*notation* Haskell's **do** notation provides convenient syntactic sugar over the operations of a monad, resembling the imperative programming approach of sequencing statements. By using the *rebindable syntax* extension of GHC, we can reuse the standard monadic **do**-notation for programming with parametric effect monads in Haskell. This is why we have chosen to use the standard names for the $return$ and ($\gg\!=$) operations here.

Axioms The axioms, or laws, of a parametric effect monad have exactly the same syntactic shape as those of monads, but with the additional effect parameters on the monadic type constructor. These

are as follows, along with their types (where for brevity here we elide the parameter m for *Plus* and *Unit* families and elide *Inv*):

$$\begin{array}{ll} & (return\ x) \gg\!= f \quad :: m\ (Plus\ Unit\ f)\ a \\ \equiv & f\ x \quad :: m\ f\ a \\ & m \gg\!= return \quad :: m\ (Plus\ f\ Unit)\ a \\ \equiv & m \quad :: m\ f\ a \\ & m \gg\!= (\lambda x \to (f\ x) \gg\!= g) \quad :: m\ (Plus\ f\ (Plus\ g\ h))\ a \\ \equiv & (m \gg\!= f) \gg\!= g \quad :: m\ (Plus\ (Plus\ f\ g)\ h)\ a \end{array}$$

For these equalities to hold, the type-level operations *Plus* and *Unit* must form a monoid, where *Unit* is the identity of *Plus* (for the first two laws), and *Plus* is associative (for the last law).

Relation to monads All monads are also parametric effect monads with a trivial singleton effect, *i.e.*, if we take $Unit\ \ m = ()$ and $Plus\ \ m\ ()\ ()\ =\ ()$. We show the full construction to embed monads into parametric effect monads in Section 7.

Relation to effect systems Figure 2(a) recalls the rules of a simple type-and-effect system using sets of effect annotations. The correspondence between type-and-effect systems (hereafter just *effect systems*) and monads was made clear by Wadler and Thiemann, who established a syntactic correspondence by annotating monadic type constructors with the effect sets of an effect system [24]. This is shown for comparison in Figure 2(b), showing a correspondence between (var)-(*unit*), (let)-(*bind*), and (sub)-(*does*).

Wadler and Thiemann established soundness results between an effect system and an operational semantics, and conjectured a "coherent semantics" of effects and monads in a denotational style. They suggested associating to each effect F a different monad M_F. The effect-parameterised monad approach here differs: a type M_F of the indexed family may not be a monad itself. The monadic behaviour is "distributed" over the indexed family of types as specified by the monoidal structure on effects. Figure 2(c) shows the effect system provided by our parametric effect monad encoding.

A key feature of effect systems is that the (abs) rule captures all effects of the body as *latent effects* that happen when the function is run (this is shown by an effect annotated arrow, *e.g.*, $\xrightarrow{F}$). This is also the case in our Haskell embedding: $\lambda x \to \mathbf{do}\ \{...\}$ is a pure function, returning a monadic computation.

The (sub) rule above provides *subeffecting*, where effects can be overapproximated. Instances of the *Subeffect* class in Figure 1 provide the corresponding operation for parametric effect monads.

3. Defining type-level sets

Early examples of effect systems often generated sets of effect information, combined via union [10], or in terms of lattices but then specialised to sets with union [9]. Sets are appropriate for effect annotations when the order of effects is irrelevant (or at least difficult to predict, for example, in a lazy language) and when effects can be treated idempotently, for example, when it is enough to know that a memory cell is read, not how many times it is read.

Later effect system descriptions separated lattices of effects into distinct algebraic structures for sequential composition, alternation, and fixed-points [17]. Our encoding of parametric effect monads is parameterised by a monoid with a preorder, but sets are an important example used throughout. In this section, we develop a type-level notion of sets (that is, sets of types, as a type) with a corresponding value-level representation. We define set union (for the sequential composition of effect information) and the calculation of subsets– providing the monoid and preorder structure on effects.

Defining type-level sets would be easier in a dependently-typed language, but perhaps the most interesting (and practically useful) thing about this paper is that we can embed effect systems in a language *without* resorting to a fully dependently-typed system.

$$\text{(var)}\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau \,!\, \emptyset} \quad \text{(let)}\frac{\Gamma \vdash e_1 : \tau_1 \,!\, F \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \,!\, G}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2 \,!\, F \cup G}$$

$$\text{(abs)}\frac{\Gamma, v : \sigma \vdash e : \tau \,!\, F}{\Gamma \vdash \lambda v.e : \sigma \xrightarrow{F} \tau \,!\, \emptyset} \quad \text{(sub)}\frac{\Gamma \vdash e : \tau \,!\, F \quad F \subseteq G}{\Gamma \vdash e : \tau \,!\, G}$$

(a) Gifford-Lucassen-style effect system [9]

$$(\textit{unit})\frac{\mathcal{E} \vdash e : \tau}{\mathcal{E} \vdash \texttt{<}e\texttt{>} : \mathsf{T}^{\emptyset}\tau} \quad (\textit{does})\frac{\mathcal{E} \vdash e : \mathsf{T}^{\sigma}\tau \quad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash e : \mathsf{T}^{\sigma'}\tau}$$

$$(\textit{bind})\frac{\mathcal{E} \vdash e : \mathsf{T}^{\sigma}\tau \quad \mathcal{E}, x : \tau \vdash e' : \mathsf{T}^{\sigma'}\tau'}{\mathcal{E} \vdash \mathtt{let}\, x \Leftarrow e\, \mathtt{in}\, e' : \mathsf{T}^{\sigma \cup \sigma'}\tau'}$$

(b) The core effectful rules for Wadler and Thiemann's *Monad* language for unifying effect systems with a monadic metalanguage [24].

$$(\textit{unit})\frac{\Gamma \vdash e : \tau}{\Gamma \vdash return\ e : m\ (Unit\ m)\ \tau} \quad \text{(sub)}\frac{\Gamma \vdash e : m\ f\ \tau \quad Sub\ f\ g}{\Gamma \vdash sub\ e : m\ g\ \tau}$$

$$(\textit{let})\frac{\Gamma \vdash e_1 : m\ f\ \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : m\ g\ \tau_2}{\Gamma \vdash \mathbf{do}\,\{x \leftarrow e_1;\ e_2\} : m\ (Plus\ m\ f\ g)\ \tau_2}$$

(c) The type-embedded effect system provided in this paper by the parametric effect monad definition.

Figure 2. Comparison of different encodings of effect systems

Representing sets with lists We encode type-level sets using various advanced type system features of GHC. The main effort is in preventing duplicate elements and enforcing the irrelevance of the storage order for elements. These properties distinguish sets from lists, which are much easier to define at the type level and will form the basis of our encoding. Type-level functions will be used to remove duplicates and normalise the list (by sorting).

We start by inductively defining *Set* as a parameterised GADT:

$$\begin{array}{l} \mathbf{data}\ Set\ (n :: [*])\ \mathbf{where} \\ \quad Empty :: Set\ `[] \\ \quad Ext \quad :: e \to Set\ s \to Set\ (e\ `:\ s) \end{array}$$

where the parameter has the *list kind* $[*]$ (the kind of lists of types) [25]. This definition encodes heterogeneously-typed lists, with a type-level list representation via type operators of kind:

$$`[] :: [*] \qquad \text{and} \qquad (`:) :: * \to [*] \to [*]$$

These provide a compact notation for types. The data constructor names *Empty* and *Ext* (for extension) remind us that we will treat values of this type as sets, rather than lists.

The first step in using lists to represent sets is to make the ordering irrelevant by (perhaps ironically) fixing an arbitrary ordering on elements of the set and normalising by sorting. We use bubble sort here as it is straightforward to implement at the type level.

A single pass of the bubble sort algorithm recurses over a list and orders successive pairs of elements as follows:

$$\begin{array}{l} \mathbf{type\ family}\ Pass\ (l :: [*]) :: [*]\ \mathbf{where} \\ \quad Pass\ `[] \quad\quad = `[] \\ \quad Pass\ `[e] \quad\quad = `[e] \\ \quad Pass\ (e\ `:\ f\ `:\ s) = Min\ e\ f\ `:\ (Pass\ ((Max\ e\ f)\ `:\ s)) \\ \mathbf{type\ family}\ Min\ (a :: k)\ (b :: k) :: k \\ \mathbf{type\ family}\ Max\ (a :: k)\ (b :: k) :: k \end{array}$$

Here, *Min* and *Max* are open type families which are given instances later for specific applications. The definition of *Pass* here uses a *closed type family* [7]. Closed type families define all of

there instances together, *i.e.*, further instances cannot be defined. This allows instances to be matched against in order, contrasting with open type families where there is no ordering on the instances (which may be scattered throughout different modules and compiled separately). $Pass$ is defined as a closed family only because we do not need it to be open, not because we require the extra power of closed families; a standard open family would suffice here.

To complete the sorting, $Pass$ is applied n-times for a list of length n. The standard optimisation is to stop once the list is sorted, but for brevity we take the simple approach, deconstructing the input list to build a chain of calls to $Pass$:

$$\begin{aligned}&\textbf{type family } \mathit{Bubble}\ l\ l' \textbf{ where}\\&\quad \mathit{Bubble}\ l\ `[] = l\\&\quad \mathit{Bubble}\ l\ (x\ `{:}\ xs) = \mathit{Pass}\ (\mathit{Bubble}\ l\ xs)\\&\textbf{type } \mathit{Sort}\ l = \mathit{Bubble}\ l\ l\end{aligned}$$

Again, we use a closed type family here, not out of necessity but since we do not need an open definition.

This completes type-level sort. Definitions of the value-level counterparts follow exactly the same shape as their types, thus we relegate their full definition to Appendix A. The approach is to implement each type-level case as an instances of the classes:

$$\begin{aligned}&\textbf{type } \mathit{Sortable}\ s = \mathit{Bubbler}\ s\ s\\&\textbf{class } \mathit{Bubbler}\ s\ s' \textbf{ where}\\&\quad \mathit{bubble} :: \mathit{Set}\ s \to \mathit{Set}\ s' \to \mathit{Set}\ (\mathit{Bubble}\ s\ s')\\&\textbf{class } \mathit{Passer}\ s \textbf{ where}\\&\quad \mathit{pass} :: \mathit{Set}\ s \to \mathit{Set}\ (\mathit{Pass}\ s)\\&\textbf{class } \mathit{OrdH}\ e\ f \textbf{ where}\\&\quad \mathit{minH} :: e \to f \to \mathit{Min}\ e\ f\\&\quad \mathit{maxH} :: e \to f \to \mathit{Max}\ e\ f\end{aligned}$$

This provides the type-specific behaviour of each case of the type-level definitions, with room to raise the appropriate type-class constraints for *OrdH* (heterogeneously typed ordering).

The remaining idempotence property of sets requires the full power of closed type families, using equality on types. We define the following type-level function Nub to remove duplicates (named after nub for removing duplicates from a list in $Data.List$):

$$\begin{aligned}&\textbf{type family } \mathit{Nub}\ t \textbf{ where}\\&\quad \mathit{Nub}\ `[] &&= `[]\\&\quad \mathit{Nub}\ `[e] &&= `[e]\\&\quad \mathit{Nub}\ (e\ `{:}\ e\ `{:}\ s) &&= \mathit{Nub}\ (e\ `{:}\ s)\\&\quad \mathit{Nub}\ (e\ `{:}\ f\ `{:}\ s) &&= e\ `{:}\ \mathit{Nub}\ (f\ `{:}\ s)\end{aligned}$$

As mentioned, the closed form of type families allows a number of cases to be matched against in lexical order. This allows the type equality comparison in the third case which removes a duplicate when two adjacent elements have the same type. The pattern of the fourth case overlaps the third, but is only tried if the third fails.

A corresponding value-level nub is defined similarly to $bubble$ and $pass$ using a type class with instances for each case of Nub:

$$\begin{aligned}&\textbf{class } \mathit{Nubable}\ t \textbf{ where}\\&\quad \mathit{nub} :: \mathit{Set}\ t \to \mathit{Set}\ (\mathit{Nub}\ t)\\&\textbf{instance } \mathit{Nubable}\ `[] \textbf{ where}\\&\quad \mathit{nub}\ \mathit{Empty} = \mathit{Empty}\\&\textbf{instance } \mathit{Nubable}\ `[e] \textbf{ where}\\&\quad \mathit{nub}\ (\mathit{Ext}\ x\ \mathit{Empty}) = \mathit{Ext}\ x\ \mathit{Empty}\\&\textbf{instance } (\mathit{Nub}\ (e\ `{:}\ f\ `{:}\ s) {\sim} (e\ `{:}\ \mathit{Nub}\ (f\ `{:}\ s)),\\&\qquad \mathit{Nubable}\ (f\ `{:}\ s)) \Rightarrow \mathit{Nubable}\ (e\ `{:}\ f\ `{:}\ s) \textbf{ where}\\&\quad \mathit{nub}\ (\mathit{Ext}\ e\ (\mathit{Ext}\ f\ s)) = \mathit{Ext}\ e\ (\mathit{nub}\ (\mathit{Ext}\ f\ s))\end{aligned}$$

In the last case, the equality constraint is required to explain the behaviour of Nub. The type and value levels are in one-to-one correspondence however we have deliberately omitted the case for actually removing items with the same type. This class instance will be defined later with application-specific behaviour.

Putting this all together, type- and value-level conversion of the list format to the set format is defined:

$$\begin{aligned}&\textbf{type } \mathit{AsSet}\ s = \mathit{Nub}\ (\mathit{Sort}\ s)\\&\mathit{asSet} :: (\mathit{Sortable}\ s, \mathit{Nubable}\ (\mathit{Sort}\ s)) \Rightarrow\\&\qquad \mathit{Set}\ s \to \mathit{Set}\ (\mathit{AsSet}\ s)\\&\mathit{asSet}\ x = \mathit{nub}\ (\mathit{bsort}\ x)\end{aligned}$$

We also define a useful predicate for later definitions which asks whether a type is in the set representation format:

$$\textbf{type } \mathit{IsSet}\ s = (s {\sim} \mathit{Nub}\ (\mathit{Sort}\ s)) :: \mathit{Constraint}$$

This uses the constraint kinds extension [3, 20] where the kind signature explains that this "type" definition is a unary constraint.

Now that we have the representation sorted, we define operations for taking the union and calculating subsets.

Union Set union is defined using our existing infrastructure and the concatenation of the underlying lists:

$$\textbf{type } \mathit{Union}\ s\ t = \mathit{AsSet}\ (\mathit{Append}\ s\ t)$$

$Append$ concatenates the two list representations (acting like a disjoint union of sets) and $AsSet$ normalises the result into the set form. $Append$ is defined in the usual way as a type family:

$$\begin{aligned}&\textbf{type family } \mathit{Append}\ (s :: [*])\ (t :: [*]) :: [*] \textbf{ where}\\&\quad \mathit{Append}\ `[]\ t &&= t\\&\quad \mathit{Append}\ (x\ `{:}\ xs)\ ys &&= x\ `{:}\ (\mathit{Append}\ xs\ ys)\end{aligned}$$

The value-level version is identical (*mutatis mutandis*):

$$\begin{aligned}&\mathit{append} :: \mathit{Set}\ s \to \mathit{Set}\ t \to \mathit{Set}\ (\mathit{Append}\ s\ t)\\&\mathit{append}\ \mathit{Empty}\ x = x\\&\mathit{append}\ (\mathit{Ext}\ e\ xs)\ ys = \mathit{Ext}\ e\ (\mathit{append}\ xs\ ys)\end{aligned}$$

This twin definition, and the previous definition for Nub/nub, exposes a weakness of Haskell: we have to write both the value and type level, even though they are essentially identical. Languages that implement richer dependent-type theories tend to avoid this problem but, for the moment, this is the state of play in Haskell.

Given all of the above, union of value sets is then:

$$\begin{aligned}&\textbf{type } \mathit{Unionable}\ s\ t = (\mathit{Sortable}\ (\mathit{Append}\ s\ t),\\&\quad \mathit{Nubable}\ (\mathit{Sort}\ (\mathit{Append}\ s\ t)))\\&\mathit{union} :: (\mathit{Unionable}\ s\ t) \Rightarrow \mathit{Set}\ s \to \mathit{Set}\ t \to \mathit{Set}\ (\mathit{Union}\ s\ t)\\&\mathit{union}\ s\ t = \mathit{nub}\ (\mathit{bsort}\ (\mathit{append}\ s\ t))\end{aligned}$$

with the binary predicate $Unionable$ hiding the underlying type class constraints associated with sorting and removing duplicates.

Subsets A notion of *subeffecting* is useful for combining effect information arising from non-linear control flow (for example, to implement conditionals). We recursively define a binary predicate Sub where $Sub\ \ s\ \ t$ means $s \subseteq t$. This type class has a single method that calculates the value representation of the subset:

$$\begin{aligned}&\textbf{class } \mathit{Subset}\ s\ t \textbf{ where}\\&\quad \mathit{subset} :: \mathit{Set}\ t \to \mathit{Set}\ s\\&\textbf{instance } \mathit{Subset}\ `[]\ t \textbf{ where}\\&\quad \mathit{subset}\ xs = \mathit{Empty}\\&\textbf{instance } \mathit{Subset}\ s\ t \Rightarrow \mathit{Subset}\ (x\ `{:}\ s)\ (x\ `{:}\ t) \textbf{ where}\\&\quad \mathit{subset}\ (\mathit{Ext}\ x\ xs) = \mathit{Ext}\ x\ (\mathit{subset}\ xs)\\&\textbf{instance } \mathit{Subset}\ s\ t \Rightarrow \mathit{Subset}\ s\ (x\ `{:}\ t) \textbf{ where}\\&\quad \mathit{subset}\ (\mathit{Ext}\ \_\ xs) = \mathit{subset}\ xs\end{aligned}$$

Thus, in the first instance: empty sets are subsets of all sets; in the second: $(\{x\} \cup S) \subseteq (\{x\} \cup T)$ if $S \subseteq T$; and in the third,

$S \subseteq (\{x\} \cup T)$ if $S \subseteq T$. Note that we have used a multi-parameter type class here since the value-level behaviour depends on both the source and target types.

Set union and subset operations will be used in the next three sections, where additional set operations will appear as necessary.

4. Writer effects

Our first example effect system will capture write effects, related to the *writer monad*. The classic writer monad provides a write-only cumulative state, useful for producing a log (or trace) along with a computation. The data type is essentially that of a product. In Haskell, this monad is defined:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Writer}\ w\ a = \mathit{Writer}\ \{\mathit{runWriter} :: (a, w)\}\\
&\mathbf{instance}\ \mathit{Monoid}\ w \Rightarrow \mathit{Monad}\ (\mathit{Writer}\ w)\ \mathbf{where}\\
&\quad \mathit{return}\ a = \mathit{Writer}\ (a, \mathit{mempty})\\
&\quad (\mathit{Writer}\ (a, w)) \gg\!= k = \mathbf{let}\ (b, w') = \mathit{runWriter}\ (k\ a)\\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \mathit{Writer}\ (b, w\ `\mathit{mappend}`\ w')
\end{aligned}$$

where $\mathit{mempty} :: \mathit{Monoid}\ w \Rightarrow w$ and $\mathit{mappend} :: \mathit{Monoid}\ w \Rightarrow w \to w \to w$ are respectively the unit element and the binary operation of a monoid on w. Thus, a pure computation writes the unit element of the monoid and $(\gg\!=)$ composes write state using the binary operation of the monoid.

Using a parametric effect monad, we can define a more flexible version of the writer monad that allows multiple writes to be easily combined and extended (without the need for tuples or monad transformers), using an effect system for write effects. This approach allows us to define programs like the following:

$$\begin{aligned}
&\mathit{prog} :: \mathit{Writer}\ `[\texttt{"x"} :\!\to \mathit{Int}, \texttt{"y"} :\!\to \mathit{String}]\ ()\\
&\mathit{prog} = \mathbf{do}\ \mathit{put}\ (\mathit{Var} :: (\mathit{Var}\ \texttt{"x"}))\ 42\\
&\qquad\quad \mathit{put}\ (\mathit{Var} :: (\mathit{Var}\ \texttt{"y"}))\ \texttt{"hello"}\\
&\qquad\quad \mathit{put}\ (\mathit{Var} :: (\mathit{Var}\ \texttt{"x"}))\ 58
\end{aligned}$$

where `"x"` and `"y"` are type level symbols and *Writer* is parameterised by a set of variable-type mappings. Running this computation produces $((), \{(\mathit{Var}, 100), (\mathit{Var}, \texttt{"hello"})\})$.

We use our type-level sets representation coupled with type-level symbols to provide variables, where the constructor $:\!\to$ describes a pair of a variable and its written type. The *Writer* data type and its accompanying parametric effect monad are defined:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Writer}\ w\ a = \mathit{Writer}\ \{\mathit{runWriter} :: (a, \mathit{Set}\ w)\}\\
&\mathbf{instance}\ \mathit{Effect}\ \mathit{Writer}\ \mathbf{where}\\
&\quad \mathbf{type}\ \mathit{Inv}\ \mathit{Writer}\ s\ t = (\mathit{IsSet}\ s, \mathit{IsSet}\ t, \mathit{Unionable}\ s\ t)\\
&\quad \mathbf{type}\ \mathit{Unit}\ \mathit{Writer} = `[]\\
&\quad \mathbf{type}\ \mathit{Plus}\ \mathit{Writer}\ s\ t = \mathit{Union}\ s\ t\\
&\quad \mathit{return}\ x = \mathit{Writer}\ (x, \mathit{Empty})\\
&\quad (\mathit{Writer}\ (a, w)) \gg\!= k = \mathbf{let}\ \mathit{Writer}\ (b, w') = k\ a\\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ \mathit{Writer}\ (b, w\ `\mathit{union}`\ w')
\end{aligned}$$

Thus, *return* has the empty set effect, and $(\gg\!=)$ composes writer states by taking the union, with the *Union* effect annotation. The *IsSet* predicates ensure the effect indices are in the set format.

The *put* operation is then defined as follows, introducing an effect with a variable-type mapping:

$$\begin{aligned}
&\mathit{put} :: \mathit{Var}\ v \to t \to \mathit{Writer}\ `[v :\!\to t]\ ()\\
&\mathit{put}\ v\ x = \mathit{Writer}\ ((), \mathit{Ext}\ v\ x\ \mathit{Empty})
\end{aligned}$$

The mapping operator $:\!\to$ and *Var* type are defined:

$$\begin{aligned}
&\mathbf{data}\ (v :: \mathit{Symbol}) :\!\to (t :: *) = (\mathit{Var}\ v) :\!\to t\\
&\mathbf{data}\ \mathit{Var}\ (v :: \mathit{Symbol}) \qquad\quad = \mathit{Var}
\end{aligned}$$

Members of the kind of symbols *Symbol* are type-level strings, provided by the *data kinds* extension.

Recall that we did not define the *nub* operation on sets fully; the case for removing duplicates at the value level was not included in the definition of Section 3. We define this here by combining values of the same variable using the *mappend* operation of a monoid:

$$\begin{aligned}
&\mathbf{instance}\ (\mathit{Monoid}\ a, \mathit{Nubable}\ ((v :\!\to a)\ `:\ s)) \Rightarrow\\
&\qquad\qquad \mathit{Nubable}\ ((v :\!\to a)\ `:\ (v :\!\to a)\ `:\ s)\ \mathbf{where}\\
&\quad \mathit{nub}\ (\mathit{Ext}\ (_ :\!\to a)\ (\mathit{Ext}\ (v :\!\to b)\ s)) =\\
&\qquad \mathit{nub}\ (\mathit{Ext}\ (v :\!\to (a\ `\mathit{mappend}`\ b))\ s)
\end{aligned}$$

We finally implement the type-level ordering of mappings $v :\!\to t$ by providing instances for *Min* and *Max*:

$$\begin{aligned}
&\mathbf{type\ instance}\ \mathit{Min}\ (v :\!\to a)\ (w :\!\to b) =\\
&\quad (\mathit{Select}\ v\ w\ v\ w) :\!\to (\mathit{Select}\ v\ w\ a\ b)\\
&\mathbf{type\ instance}\ \mathit{Max}\ (v :\!\to a)\ (w :\!\to b) =\\
&\quad (\mathit{Select}\ v\ w\ w\ v) :\!\to (\mathit{Select}\ v\ w\ b\ a)\\
&\mathbf{type}\ \mathit{Select}\ a\ b\ p\ q = \mathit{Choose}\ (\mathit{CmpSymbol}\ a\ b)\ p\ q\\
&\mathbf{type\ family}\ \mathit{Choose}\ (o :: \mathit{Ordering})\ p\ q\ \mathbf{where}\\
&\quad \mathit{Choose}\ \mathit{LT}\ p\ q = p\\
&\quad \mathit{Choose}\ \mathit{EQ}\ p\ q = p\\
&\quad \mathit{Choose}\ \mathit{GT}\ p\ q = q
\end{aligned}$$

where $\mathit{CmpSymbol} :: \mathit{Symbol} \to \mathit{Symbol} \to \mathit{Ordering}$ from the base library compares symbols, returning a type of kind *Ordering* upon which *Choose* matches. The type function *Select* selects its third or fourth parameter based on the variables passed as its first two parameters; *Select* returns its third parameter if the first parameter is less than the second, otherwise it returns its fourth. The corresponding value level is a straightforward (and annoying!) transcription of the above, shown in Appendix B for reference.

Examples and polymorphism The following gives a simple example (using an additive monoid on *Int*):

$$\begin{aligned}
&\mathit{varX} = \mathit{Var} :: (\mathit{Var}\ \texttt{"x"})\\
&\mathit{varY} = \mathit{Var} :: (\mathit{Var}\ \texttt{"y"})\\
&\mathit{test} = \mathbf{do}\ \mathit{put}\ \mathit{varX}\ (42 :: \mathit{Int})\\
&\qquad\quad \mathit{put}\ \mathit{varY}\ \texttt{"hello"}\\
&\qquad\quad \mathit{put}\ \mathit{varX}\ (58 :: \mathit{Int})\\
&\qquad\quad \mathit{put}\ \mathit{varY}\ \texttt{" world"}
\end{aligned}$$

The effects are easily inferred (shown here by querying GHCi):

```
*Main> :t test
test :: Writer '["x" :-> Int, "y" :-> [Char]] ()
```

and the code executes as expected:

```
*Main> runWriter (test 1)
((),(x, 100), (y, "hello world"))
```

Explicit type signatures were used on assignments to `"x"` otherwise our implementation cannot unify the two writes to `"x"`. If we want `"x"` to be polymorphic we must use a *scoped type variable* with a type signature fixing the type of each *put* to x. For example:

$$\begin{aligned}
&\mathit{test}'\ (n :: a) = \mathbf{do}\ \mathit{put}\ \mathit{varX}\ (42 :: a)\\
&\qquad\qquad\quad \mathit{put}\ \mathit{varY}\ \texttt{"hello"}\\
&\qquad\qquad\quad \mathit{put}\ \mathit{varX}\ (n :: a)
\end{aligned}$$

for which Haskell can infer the expected polymorphic effect type:

```
*Main> :t test'
test' :: (Monoid a, Num a) =>
         a -> Writer '["x" :-> a, "y" :-> [Char]] ()
```

While it is cumbersome to have to add explicit type signatures for the polymorphism here, the overhead is not vast and the type system can still infer the effect type for us. We can also be entirely polymorphic in an effect, and in a higher-order setting. For exam-

ple, the following function takes an effectful function as a parameter and applies it, along with some of its own write effects:

$$\begin{aligned}&test2 :: (IsSet\ f, Unionable\ f\ \text{`}[\texttt{"y"} :\!\rightarrow String]) \\ &\quad\Rightarrow\quad (Int \rightarrow Writer\ f\ t) \\ &\quad\rightarrow\quad Writer\ (Union\ f\ \text{`}[\texttt{"y"} :\!\rightarrow String])\ () \\ &test2\ f = \mathbf{do}\ \{f\ 3;\ \ put\ varY\ \texttt{"world."}\}\end{aligned}$$

Thus, *test2* takes an effectful f, calls it with 3, and then writes to `"y"`. The resulting effect is thus the union of f's effects and $\text{`}[\texttt{"y"} :\!\rightarrow String]$. To test, $runWriter\ (test2\ test')$ returns the expected values $((), \{(x, 45), (y, \texttt{"hello world."})\})$.

While the type of *test2* can be inferred (if we give a signature on 3, *e.g.*, $3 :: Int$), we include an explicit type signature here as GHC has a habit of expanding type synonym definitions, making the inferred type a bit inscrutable.

Subeffecting Since sets appear in a positive position in our *Writer* data type, subeffecting overapproximates what is written, requiring a superset operation for writer effects. At the value level, we fill these additional writer cells with unit of the corresponding monoid ($mempty$), thus completing the use of monoids in this example (rather than just semigroups). We define a binary predicate *Superset* with a *superset* method:

$$\begin{aligned}&\mathbf{class}\ Superset\ s\ t\ \mathbf{where}\ superset :: Set\ s \rightarrow Set\ t \\ &\mathbf{instance}\ Superset\ \text{`}[]\ \text{`}[]\ \mathbf{where}\ superset\ \_ = Empty \\ &\mathbf{instance}\ (Monoid\ a, Superset\ \text{`}[]\ s) \Rightarrow \\ &\qquad Superset\ \text{`}[]\ ((v :\!\rightarrow a)\ \text{`}{:}\ s)\ \mathbf{where} \\ &\quad superset\ \_ = Ext\ (Var :\!\rightarrow mempty)\ (superset\ Empty) \\ &\mathbf{instance}\ Superset\ s\ t \Rightarrow \\ &\qquad Superset\ ((v :\!\rightarrow a)\ \text{`}{:}\ s)\ ((v :\!\rightarrow a)\ \text{`}{:}\ t)\ \mathbf{where} \\ &\quad superset\ (Ext\ x\ xs) = Ext\ x\ (superset\ xs)\end{aligned}$$

The subeffecting operation for *Writer* is then:

$$\begin{aligned}&\mathbf{instance}\ Superset\ s\ t \Rightarrow Subeffect\ Writer\ s\ t\ \mathbf{where} \\ &\quad sub\ (Writer\ (a, w)) = Writer\ (a, (superset\ w) :: (Set\ t))\end{aligned}$$

To illustrate, we apply *sub* to our earlier example:

$$\begin{aligned}&test3 :: Writer\text{`}[\texttt{"x"} :\!\rightarrow Int, \texttt{"y"} :\!\rightarrow String, \texttt{"z"} :\!\rightarrow Int]\ () \\ &test3 = sub\ (test2\ test')\end{aligned}$$

which evaluates to the following showing the 0 value given to `"z"` coming from the additive monoid for *Int*:

```
*Main> runWriter test3
((),(x, 45), (y, "hello world."), (z, 0))
```

Using plain lists A simpler, but less useful version of writer effects uses just type-level lists, rather than sets. This provides a *write-once* writer where values can be written but with no accumulating behaviour. We elide this example here as it is less useful, but it can be found in `Control.Effect.WriteOnceWriter`.

4.1 Update effects

An alternate form of writer effect provides an updateable memory cell, without any accumulating behaviour. This corresponds to the usual writer monad with the monoid over *Maybe*: writing a value wrapped by the *Just* constructor updates the cell, writing *Nothing* leaves the cell unmodified. With a parametric effect monad we can treat the type of the cell as an effect annotation, providing a heterogeneously-typed update monad. The standard monadic definition must have the same type throughout the computation. Thus, this effect system is more about generalising the power of the monad than program analysis per se.

This parametric effect monad is defined by lifting the *Maybe*-monoid to types. We define a GADT parameterised by *Maybe* promoted to a kind:

$$\begin{aligned}&\mathbf{data}\ Eff\ (w :: Maybe\ *)\ \mathbf{where} \\ &\quad Put :: a \rightarrow Eff\ (Just\ a) \\ &\quad NoPut :: Eff\ Nothing\end{aligned}$$

The effect-parameterised version of the update monad is then:

$$\begin{aligned}&\mathbf{data}\ Update\ w\ a = U\ \{runUpdate :: (a, Eff\ w)\} \\ &\mathbf{instance}\ Effect\ Update\ \mathbf{where} \\ &\quad \mathbf{type}\ Unit\ Update = Nothing \\ &\quad \mathbf{type}\ Plus\ Update\ s\ Nothing = s \\ &\quad \mathbf{type}\ Plus\ Update\ s\ (Just\ t) = Just\ t \\ &\quad return\ x \qquad\quad = U\ (x, NoPut) \\ &\quad (U\ (a, w)) \gg\!= k = U\ (update\ w\ (runUpdate\ \$\ k\ a)) \\ &update :: Eff\ s \rightarrow (b, Eff\ t) \rightarrow (b, Eff\ (Plus\ Update\ s\ t)) \\ &update\ w\ (b, NoPut) = (b, w) \\ &update\ \_\ (b, Put\ w'') = (b, Put\ w'') \\ &put :: a \rightarrow Update\ (Just\ a)\ () \\ &put\ x = U\ ((), Put\ x)\end{aligned}$$

where *update* combines value- and type-level *Maybe* monoid behaviour. Note that we don't have to use the GADT approach. We could equivalently define two data types *Put* and *NoPut* and implement the type-dependent behaviour of *update* using a type class.

The effect-parameterised writer monad therefore provides a heterogeneously-typed memory cell, where the final type of the state for a computation is that of the last write, *e.g.*

$$\begin{aligned}&foo :: Update\ (Just\ String)\ () \\ &foo = \mathbf{do}\ \{put\ 42; put\ \texttt{"hello"}\}\end{aligned}$$

This parametric effect monad is a little baroque, but it serves to demonstrate the heterogeneous behaviour possible with parametric effect monads and gives an example effect system that is not based on sets (of which there are more examples later).

5. Reader effects

The classic *reader monad* provides a read-only value (or *parameter*) that is available throughout a computation. The data type of the reader monad is a function from the read-only state to a value:

$$\mathbf{data}\ Reader\ r\ a = Reader\ \{runReader :: r \rightarrow a\}$$

Similarly to the previous section, we can generalise this monad to a parametric effect monad providing an effect system for read effects and allowing multiple different reader values, solving the composition problem for multiple reader monads. The generalised type and parametric effect monad instance are defined:

$$\begin{aligned}&\mathbf{data}\ Reader\ s\ a = R\ \{runReader :: Set\ s \rightarrow a\} \\ &\mathbf{instance}\ Effect\ Reader\ \mathbf{where} \\ &\quad \mathbf{type}\ Inv\ Reader\ s\ t = (IsSet\ s, IsSet\ t, \\ &\qquad\qquad\qquad\qquad Split\ s\ t\ (Union\ s\ t)) \\ &\quad \mathbf{type}\ Unit\ Reader\ \ \ = \text{`}[] \\ &\quad \mathbf{type}\ Plus\ Reader\ s\ t = Union\ s\ t \\ &\quad return\ x = R\ (\lambda Empty \rightarrow x) \\ &\quad (R\ e) \gg\!= k = R\ (\lambda st \rightarrow \mathbf{let}\ (s, t) = split\ st \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (runReader\ \$\ k\ (e\ s))\ t)\end{aligned}$$

A pure computation therefore reads nothing, taking the empty set as an argument. For the composition of effectful computations, we define a computation that takes in a set $st :: Set\ (Union\ s\ t)$ and then splits it into two parts $s :: Set\ s$ and $t :: Set\ t$ which are passed to the subcomputations $e :: Set\ s \rightarrow a$ and $k\ (e\ s) : Set\ t \rightarrow b$.

Although set union is not an injective operation (*i.e.*, not invertible), the *split* operation here provides the inverse of $Union\ s\ t$

since s and t are known, provided by the types of the two subcomputations. We define *split* via a type class that is parameterised by its parameter set and return sets:

$$\textbf{class } Split\ s\ t\ st \textbf{ where } split :: Set\ st \to (Set\ s, Set\ t)$$

$$\textbf{instance } Split\ `[]\ `[]\ `[] \textbf{ where}$$
$$\quad split\ Empty = (Empty, Empty)$$

$$\textbf{instance } Split\ s\ t\ st \Rightarrow Split\ (e\ `:\ s)\ (e\ `:\ t)\ (e\ `:\ st) \textbf{ where}$$
$$\quad split\ (Ext\ x\ st) = \textbf{let } (s, t) = split\ st \textbf{ in } (Ext\ x\ s, Ext\ x\ t)$$

$$\textbf{instance } Split\ s\ t\ st \Rightarrow Split\ (x\ `:\ s)\ t\ (x\ `:\ st) \textbf{ where}$$
$$\quad split\ (Ext\ x\ st) = \textbf{let } (s, t) = split\ st \textbf{ in } (Ext\ x\ s, t)$$

$$\textbf{instance } Split\ s\ t\ st \Rightarrow Split\ s\ (x\ `:\ t)\ (x\ `:\ st) \textbf{ where}$$
$$\quad split\ (Ext\ x\ st) = \textbf{let } (s, t) = split\ st \textbf{ in } (s, Ext\ x\ t)$$

The first instance provides the base case. The second provides the case when an element of a $Union\ f\ g$ appears in both f and g. The third and fourth instances provide the cases when an element of $Union\ f\ g$ is only in f or only in g.

The constraint $Split\ s\ t\ (Union\ s\ t)$ in the *Effect* instance enforces that *Split* is the inverse of $Union$.

Once we have the above parametric effect monad, the usual *ask* operation takes a variable as a parameter and produces a computation with a singleton effect for that variable:

$$ask :: Var\ v \to Reader\ `[v :\to t]\ t$$
$$ask\ Var = R\ (\lambda(Ext\ (Var :\to x)\ Empty) \to x)$$

The following gives an example program, whose type and effects are easily inferred by GHC, so we do not give a type signature here:

$$\begin{aligned} foo = \textbf{do}\ & x \leftarrow ask\ (Var :: (Var\ \texttt{"x"})) \\ & xs \leftarrow ask\ (Var :: (Var\ \texttt{"xs"})) \\ & x' \leftarrow ask\ (Var :: (Var\ \texttt{"x"})) \\ & return\ (x : x' : xs) \end{aligned}$$
$$init1 = Ext\ (Var :\to 1)\ (Ext\ (Var :\to [2,3])\ Empty)$$
$$runFoo = runReader\ foo\ init1$$

The inferred type is $foo :: Reader\ `[\texttt{"x"} :\to a, \texttt{"xs"} :\to [a]]\ [a]$ and *runFoo* evaluates to $[1, 1, 2, 3]$.

Note that we have not had to add a case for the *Nubable* type class with the *nub* method to removing duplicates in sets. This is because *Reader* does not use *union* (sets appear in a negative position, to the left of the function arrow). Instead, the idempotent behaviour is encoded by the definition of *split*/*Split*.

Sub effecting Since sets appear in negative positions, we can use the *subset* function defined earlier for subeffecting:

$$\textbf{instance } Subset\ s\ t \Rightarrow Subeffect\ Reader\ s\ t \textbf{ where}$$
$$\quad sub\ (R\ e) = R\ (\lambda st \to \textbf{let } s = subset\ st \textbf{ in } e\ s)$$

The following overapproximates the effects of the above example:

$$bar :: (Subset `[\texttt{"x"} :\to Int, \texttt{"xs"} :\to [Int]]\ t) \Rightarrow Reader\ t\ [Int]$$
$$bar = sub\ foo$$

This can be run by passing $\bot$ into the additional slot in the incoming reader set with initial reader state:

$$init2 :: Set `[\texttt{"x"} :\to Int, \texttt{"xs"} :\to [Int], \texttt{"z"} :\to a]$$
$$\begin{aligned} init2 = Ext\ & (Var :\to 1)\ (Ext\ (Var :\to [2,3]) \\ & (Ext\ (Var :\to \bot)\ Empty)) \end{aligned}$$

where $runReader\ bar\ init2$ evaluates to $[1, 1, 2, 3]$. The explicit signature on *init2* is required for the subeffecting function to be correctly resolved.

This effect system resembles the *implicit parameters* extension of Haskell [14], providing most of the same functionality. However, some additional structure is need to fully replicate the implicit parameter behaviour. This is discussed in Section 8 where we briefly discuss the dual notion of coeffect systems.

6. State effects

The earliest effect systems were designed specifically to track side-effects relating to state, with sets of triples marking read, write, and update effects on typed locations. We combine the approaches thus far for reader and writer effects to define a parametric state effect monad with state effect system. As before, we will use sets for effects, but this time with additional type-level information for distinguishing between reads, writes, and updates (read/write), given by the *Eff* type:

$$\textbf{data } Eff = R \mid W \mid RW$$
$$\textbf{data } Effect\ (s :: Eff) = Eff$$

where the *Effect* type uses *Eff* as a data kind and provides a data constructor that acts as a proxy for *Eff*. These effects markers are associated with types, describing the effect performed on a value of a particular type, with the constructor:

$$\textbf{data } (:!)\ (a :: *)\ (s :: Eff) = a :!\ (Effect\ s)$$

Effect annotations will be sets of mappings of the form $(v :\to t :! f)$ meaning variable v has type t and effect action f (drawn from *Eff*).

The parametric effect monad data type *State* is analogous to usual definition of state $s \to (a, s)$:

$$\begin{aligned} &\textbf{data } State\ s\ a = State \\ &\quad \{ runState :: Set\ (Reads\ s) \to (a, Set\ (Writes\ s))\} \end{aligned}$$

where *Reads* and *Writes* refine the set of effects into the read and write effects respectively. *Read* is defined:

$$\begin{aligned} &\textbf{type family } Reads\ t \textbf{ where} \\ &\quad Reads\ `[] &&= `[] \\ &\quad Reads\ ((v :\to a :!\ R)\ `:\ s) &&= (v :\to a :!\ R)\ `:\ (Reads\ s) \\ &\quad Reads\ ((v :\to a :!\ RW)\ `:\ s) &&= (v :\to a :!\ R)\ `:\ (Reads\ s) \\ &\quad Reads\ ((v :\to a :!\ W)\ `:\ s) &&= Reads\ s \end{aligned}$$

thus read-write effects RW are turned into read effects, and all write effects are ignored. The *Writes* operation (not shown here) removes R actions and turns RW actions into W actions.

Previously, set union combined effect sets, but now we need some additional behaviour in the case where both sets contain effects on a variable v but with different effect actions. For example, we require the behaviour that:

$$Union\ `[v :\to t :!\ R]\ `[v :\to t :!\ W] = `[v :\to t :!\ RW]$$

i.e., if one computation reads v and the other writes v the overall effect is a *read-write* effect (possible update). We thus redefine the previous *Nub* definition:

$$\begin{aligned} &\textbf{type family } Nub\ t \textbf{ where} \\ &\quad Nub\ `[] &&= `[] \\ &\quad Nub\ `[e] &&= `[e] \\ &\quad Nub\ (e\ `:\ e\ `:\ s) &&= Nub\ (e\ `:\ s) \\ &\quad Nub\ ((v :\to a :!\ f)\ `:\ (v :\to a :!\ g)\ `:\ s) &&= \\ &\qquad\qquad Nub\ ((v :\to a :!\ RW)\ `:\ s) \\ &\quad Nub\ (e\ `:\ f\ `:\ s) &&= e\ `:\ Nub\ (f\ `:\ s) \end{aligned}$$

Again, closed type families are used to match against types in the given order. The definition is the same as before in Section 3, apart from the third case which is new: if there are two different effects f and g on variable v then these are combined into one effect annotation with action RW. The value level is straightforward and analogous to the type-level (see `Control.Effect.State`) and is similar to the previous definition in Section 3. The union of two sets is defined as before, using sorting and the above version of *Nub*. To

distinguish this union from the previous (which is an actual union), we define $\mathbf{type}\ UnionS\ s\ t = Nub\ (Sort\ (Append\ s\ t))$.

A final operation is required to sequentially compose write effects of one computation with read effects of another. This amounts to a kind of intersection of two sets, between a set of write effects and a set of read $w \cap r$ where at the type level this equals r, but at the value level any reads in r that coincide with writes in w are replaced by the written values. We define this operation by first appending the two sets, sorting them, then filtering with $intersectR$:

$$\begin{aligned}&\mathbf{type}\ IntersectR\ s\ t = (Sortable\ (Append\ s\ t),\\ &\qquad\qquad Update\ (Append\ s\ t)\ t)\\ &intersectR :: (Writes\ s{\sim}s, Reads\ t{\sim}t, IntersectR\ s\ t)\\ &\qquad\Rightarrow Set\ s \to Set\ t \to Set\ t\\ &intersectR\ s\ t = update\ (bsort\ (append\ s\ t))\end{aligned}$$

The constraints here restrict us to just read effects in s and write effects in t. The $update$ function replaces any reader values with written values (if available). This is defined by the $Update$ class:

$$\begin{aligned}&\mathbf{class}\ Update\ s\ t\ \mathbf{where}\ update :: Set\ s \to Set\ t\\ &\mathbf{instance}\ Update\ xs\ \text{`}[]\ \mathbf{where}\ update\ \_ = Empty\\ &\mathbf{instance}\ Update\text{`}[e]\text{`}[e]\ \mathbf{where}\ update\ s = s\\ &\mathbf{instance}\ Update\ ((v :\to a :!\ R)\ \text{`}{:}\ as)\ as' \Rightarrow\\ &\quad Update\ ((v :\to a :!\ W)\ \text{`}{:}\ (v :\to b :!\ R)\ \text{`}{:}\ as)\ as'\ \mathbf{where}\\ &\quad update\ (Ext\ (v :\to (a :!\ \_))\ (Ext\ \_\ xs)) =\\ &\qquad update\ (Ext\ (v :\to (a :!\ (Eff :: (Effect\ R))))\ xs)\\ &\mathbf{instance}\ Update\ ((u :\to b :!\ s)\ \text{`}{:}\ as)\ as' \Rightarrow\\ &\quad Update\ ((v :\to a :!\ W)\ \text{`}{:}\ (u :\to b :!\ s)\ \text{`}{:}\ as)\ as'\ \mathbf{where}\\ &\quad update\ (Ext\ \_\ (Ext\ e\ xs)) = update\ (Ext\ e\ xs)\\ &\mathbf{instance}\ Update\ ((u :\to b :!\ s)\ \text{`}{:}\ as)\ as' \Rightarrow\\ &\quad Update\ ((v :\to a :!\ R)\ \text{`}{:}\ (u :\to b :!\ s)\ \text{`}{:}\ as)\\ &\qquad\qquad ((v :\to a :!\ R)\ \text{`}{:}\ as')\ \mathbf{where}\\ &\quad update\ (Ext\ e\ (Ext\ e'\ xs)) = Ext\ e\ (update\ (Ext\ e'\ xs))\end{aligned}$$

The first two instances provide the base cases. The third instance provides the intersection behaviour of replacing a read value with a written value. Since sorting is defined on the symbols used for variables, the ordering of write effects before read effects is preserved, hence we only need consider this case of a write preceding a read. The fourth instance ignores a write that has no corresponding read. The fifth instance keeps a read that has no overwriting write effect.

Finally, we can define the full state parametric effect monad:

$$\begin{aligned}&\mathbf{instance}\ Effect\ State\ \mathbf{where}\\ &\quad \mathbf{type}\ Unit\ State = \text{`}[]\\ &\quad \mathbf{type}\ Plus\ State\ s\ t = UnionS\ s\ t\\ &\quad return\ x = State\ (\lambda Empty \to (x, Empty))\\ &\quad (State\ e) \gg\!= k = State\ (\lambda st \to\\ &\qquad \mathbf{let}\ (sR, tR) = split\ st\\ &\qquad\quad (a, sW) = e\ sR\\ &\qquad\quad (b, tW) = (runState\ (k\ a))\ (sW\ \text{`}intersectR\text{`}\ tR)\\ &\qquad \mathbf{in}\ \ (b, sW\ \text{`}union\text{`}\ tW))\end{aligned}$$

Thus, a pure computation has no reads and no writes. When composing computations, an input state st is split into the reader states sR and tR for the two subcomputations. The first computation is run with input state sR yielding some writes sW as output state. These are then intersected with tR to give the input state to $(k\ a)$ which produces output state tW. This is then unioned with sW to get the final output state.

The definition for Inv is elided (see `Control.Effect.State`) since it is quite long but has no surprises. As before it constrains s and t to be in the set format, and includes various type-class constraints for $Unionable$, $Split$ and $IntersectR$.

We can now encode the examples of the introduction. For the stream processing example, we can define the operations as:

$$\begin{aligned}&varC = Var :: (Var\ \texttt{"count"})\\ &varS = Var :: (Var\ \texttt{"out"})\\ &incC :: State\ \text{`}[\texttt{"count"} :\to Int :!\ RW]\ ()\\ &incC = \mathbf{do}\ \{x \leftarrow get\ varC; put\ varC\ (x+1)\}\\ &writeS :: [a] \to State\ \text{`}[\texttt{"out"} :\to [a] :!\ RW]\ ()\\ &writeS\ y = \mathbf{do}\ \{x \leftarrow get\ varS; put\ varS\ (x +\!\!+ y)\}\\ &write :: [a] \to State\ \text{`}[\texttt{"count"} :\to Int :!\ RW,\\ &\qquad\qquad \texttt{"out"} :\to [a] :!\ RW]\ ()\\ &write\ x = \mathbf{do}\ \{writeS\ x; incC\}\end{aligned}$$

7. Monads as parametric effect monads

As explained in the introduction, all monads are parametric effect monads with a trivial *singleton* effect. This allows us to embed existing monads into parametric effect monads with a wrapper:

$$\begin{aligned}&\mathbf{import\ qualified}\ Prelude\ \mathbf{as}\ P\\ &\mathbf{data}\ Monad\ m\ t\ a\ \mathbf{where}\\ &\quad Wrap :: P.Monad\ m \Rightarrow m\ a \to Monad\ m\ ()\ a\\ &unWrap :: Monad\ m\ t\ a \to m\ a\\ &unWrap\ (Wrap\ m) = m\\ &\mathbf{instance}\ (P.Monad\ m) \Rightarrow Effect\ (Monad\ m)\ \mathbf{where}\\ &\quad \mathbf{type}\ Unit\ (Monad\ m)\ \ \ = ()\\ &\quad \mathbf{type}\ Plus\ (Monad\ m)\ s\ t = ()\\ &\quad return\ x = Wrap\ (P.return\ x)\\ &\quad (Wrap\ x) \gg\!= f = Wrap\ ((P.\gg\!=)\ x\ (unWrap \circ f))\end{aligned}$$

This provides a pathway to entirely replacing the standard $Monad$ class of Haskell with $Effect$.

8. Implicit parameters and coeffects

The parametric effect reader monad of Section 5 essentially embeds an effect system for *implicit parameters* into Haskell, an existing extension of Haskell [14]. Implicit parameters provide dynamically scoped variables. For example, the following function sums three numbers, two of which are passed implicitly (dynamically):

$$\begin{aligned}&sum3 :: (Num\ a, ?x :: a, ?y :: a) \Rightarrow a \to a\\ &sum3\ z = ?x + ?y + z\end{aligned}$$

where implicit parameters are syntactically introduced by a preceding question mark. Any implicit parameters used in an expression are represented in the expression's type as constraints (shown above). These implicit parameter constraints are a kind of effect analysis, similar to that of our reader effect monad. In our approach, a similar definition to $sum3$ is:

$$\begin{aligned}&sum3 :: (Num\ a) \Rightarrow a \to Reader\ \text{`}[\texttt{"?x"} :\to a, \texttt{"?y"} :\to a]\ a\\ &sum3\ z = \mathbf{do}\ x \leftarrow ask\ (Var :: (Var\ \texttt{"?x"}))\\ &\qquad\qquad y \leftarrow ask\ (Var :: (Var\ \texttt{"?y"}))\\ &\qquad\qquad return\ (x + y + z)\end{aligned}$$

This is longer than the implicit parameter approach since the **do**-notation is needed to implement effect sequencing and the symbol encoding of variables is required, but the essence is the same.

However, the two approaches have a significant difference. Our effect-parameterised reader monad provides fully dynamically scoped variables, that is, they are bound only when the computation is run. In contrast, implicit parameters allow a mix of dynamic and static (lexical) scoping. For example, we can write:

$$\begin{aligned}&sum2 :: (Num\ a, ?y :: a) \Rightarrow a \to a\\ &sum2 = \mathbf{let}\ ?x = 42\ \mathbf{in}\ \lambda z \to ?x + ?y + z\end{aligned}$$

where the **let** binds the lexically scoped $?x$ inside of the λ-expression, but $?y$ remains dynamically scoped, as shown by the type. Without entering into the internals of *Reader* we cannot (yet) implement the same behaviour with the monadic approach. This illustrates how the implicit parameters extension is *not* an instance of an effect system or monadic semantics approach, in the traditional sense. The main difference is in the treatment of λ-abstraction.

Recall the standard type-and-effect rule for λ-abstraction [9], which makes all effects *latent*. Unifying effect systems with monads via parametric effect monads gives the semantics [12]:

$$\frac{[\![\Gamma, x : \sigma \vdash e : \tau, F]\!] = g : \Gamma \times \sigma \to \mathsf{M}_F\tau}{[\![\Gamma \vdash \lambda x.e : \sigma \xrightarrow{F} \tau, \emptyset]\!] = \mathit{return}(\mathit{uncurry}\, g) : \Gamma \to \mathsf{M}_\emptyset(\sigma \to \mathsf{M}_F\tau)}$$

where the returned function is pure (as defined by $return$).

This contrasts with the abstraction rule for implicit parameters [14]. Lewis *et al.* describe implicit parameter judgments of the form $C; \Gamma \vdash e : \tau$ where C augments the usual typing relation with a set of constraints. The rule for abstraction is then:

$$(abs)\frac{C; \Gamma, v : \sigma \vdash e : \tau}{C; \Gamma \vdash \lambda v.e : \tau}$$

If constraints C are thought of as effect annotations, then we see that the λ-abstraction is *not* pure in the sense that the constraints of the body e are now the constraints of the λ-abstraction (no latent effects). When combined with their rule for discharging implicit parameters, this allows lexically scoped implicit parameters.

The semantics of these implicit parameters has been described separately in terms of a *comonadic* semantics for implicit parameters with a *coeffect* system [21].

Comonads and coeffects Comonads dualise monads, revealing a structure of the following form (taken from `Control.Comonad`):

$$\begin{array}{l}\textbf{class}\ \mathit{Comonad}\ c\ \textbf{where}\\ \quad \mathit{extract} :: c\ a \to a\\ \quad \mathit{extend} :: (c\ a \to b) \to c\ a \to c\ b\end{array}$$

where $extract$ is the dual of $return$ and $extend$ is the infix dual of $(\gg\!=)$. Comonads can be described as capturing *input impurity*, *input effects*, or *context-dependent* notions of computation.

Recently, *coeffect systems* have been introduced as the comonadic analogues of effect systems for analysing resource usage and context-dependence in programs [4, 8, 21]. The semantics of theses systems each include a dual to parametric effect monads (in various forms), which we call here *parametric coeffect comonads* (earlier called *indexed comonads* [21]).

We write coeffect judgments as $\Gamma ? R \vdash e : \tau$, meaning an expression e has coeffects (or *requirements*) R. The key distinguishing feature between (simple) coeffect systems, shown in [21], and effect systems is the abstraction rule, which has the form:

$$(abs)\frac{\Gamma, x : \sigma ? F \otimes G \vdash e : \tau}{\Gamma ? F \vdash \lambda x.e : \sigma \xrightarrow{G} \tau}$$

for some binary operation $\otimes$ on coeffects. Thus, in a coeffect system, λ-abstraction is not "pure". Instead, reading the rule top-down, coeffects of the body are split between the declaration site (immediate coeffects) and the call site (latent coeffects); reading bottom up, the contexts available at the declaration site and call site are merged to give the context of the body.

In the semantics of coeffect systems, coeffect judgments are interpreted as morphisms: $[\![\Gamma ? F \vdash e : \tau]\!] : \mathsf{D}_F[\![\Gamma]\!] \to [\![\tau]\!]$ where D_F is a parametric coeffect comonad. The semantics of abstraction requires an additional monoidal operation on D of type $\mathit{merge} : \mathsf{D}_F A \times \mathsf{D}_G B \to \mathsf{D}_{F\otimes G}(A \times B)$, giving the rule:

$$\frac{[\![\Gamma, x : \sigma ? F \otimes G \vdash e : \tau]\!] = g : \mathsf{D}_{F\otimes G}(\Gamma \times \sigma) \to \tau}{[\![\Gamma ? F \vdash \lambda x.e : \sigma \xrightarrow{G} \tau]\!] = \mathit{uncurry}(g \circ \mathit{merge}) : \mathsf{D}_F\Gamma \to (\mathsf{D}_G\sigma \to \tau)}$$

Implicit parameters as coeffects A coeffect system (with the above abstraction rule) with coeffects as sets of variable-type pairs provides the constraints behaviour of implicit parameters [21]. This allows the $sum2$ example for implicit parameters to be typed, with additional syntax for binding implicit parameters:

$$(let?)\frac{(abs)\dfrac{\Gamma, z : a\ ?\ \{?x : a, ?y : a\} \vdash ?x + ?y + z : a}{\Gamma\ ?\ \{?x : a\} \vdash \lambda z.?x + ?y + z : a \xrightarrow{\{?y:a\}} a} \qquad \Gamma\ ?\ \emptyset \vdash \textbf{let?}\ ?x = e\ \textbf{in}\ (\lambda z.?x + ?y + z) : a \xrightarrow{\{?y:a\}} a}$$

Thus, the requirements of the function body are split, with $\{?x : a\}$ becoming an immediate coeffect which is discharged by the **let?** binding, and $\{?y : a\}$ remaining latent.

The semantics can be given in terms of a coeffect-parameterised *product* comonad on $\mathsf{P}_F A = A \times F$, and an operation $merge : \mathsf{P}_F A \times \mathsf{P}_G B \to \mathsf{P}_{F\cup G}(A \times B)$ taking the union of the coeffects.

Reader as a monad or comonad By (un)currying, functions of type $\mathsf{P}_F A \to B$ (*e.g.*, denotations of the coeffect semantics) are isomorphic to functions $A \to \mathit{Reader}\ F\ B$ of our parametric effect reader, *i.e.*, $curry :: ((A \times F) \to B) \to (A \to (F \to B))$, and vice versa by *uncurry*. Thus, we can structure sequential reader computations using either the comonadic or monadic approach. The difference is in the treatment of abstraction, as we have seen above with $merge$. However, we can recover the mixed lexical/dynamic behaviour of the implicit parameters extension by providing the isomorphic version of $merge$ for the $Reader$ type:

$$\begin{array}{l}\mathit{merge} :: (\mathit{Unionable}\ s\ t) \Rightarrow (a \to \mathit{Reader}\ (\mathit{Union}\ s\ t)\ b)\\ \qquad\qquad\qquad\qquad \to \mathit{Reader}\ s\ (a \to \mathit{Reader}\ t\ b)\\ \mathit{merge}\ k = R\ (\lambda s \to \lambda a \to\\ \qquad\qquad R\ (\lambda t \to \mathit{runReader}\ (k\ a)\ (\mathit{union}\ s\ t)))\end{array}$$

This merges the immediate requirements/effects that occur before the function is applied and latent requirements/effects for when the function is applied, providing requirements $Union\ s\ t$. We see here the merging behaviour described above in the coeffect setting, where the union of two implicit parameter environments is taken.

Therefore, $merge$ allows mixed lexical/dynamic scoping of implicit parameters with $Reader$. For example, $sum2$ (which used implicit parameters) can now be equivalently expressed as:

$$\begin{array}{l}\mathit{sum2} :: \mathit{Num}\ a \Rightarrow a \to \mathit{Reader}\text{`}[\texttt{"?y"} :\!\to a]\ a\\ \mathit{sum2} = \textbf{let}\ x = (\mathit{Ext}\ ((\mathit{Var} :: (\mathit{Var}\ \texttt{"?x"})) :\!\to 42)\ \mathit{Empty})\\ \qquad\textbf{in}\ \mathit{runReader}\\ \qquad\quad (\mathit{merge}\ (\lambda z \to \textbf{do}\ x \leftarrow \mathit{ask}\ (\mathit{Var} :: (\mathit{Var}\ \texttt{"?x"}))\\ \qquad\qquad\qquad\qquad\qquad\quad y \leftarrow \mathit{ask}\ (\mathit{Var} :: (\mathit{Var}\ \texttt{"?y"}))\\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (x + y + z)))\ x\end{array}$$

Thus, we lexically scope `?x` via $merge$ with our original $sum3$ definition, leaving only the requirement for `?y`.

We have seen here that Haskell's implicit parameters are a kind of coeffect analysis, or an effect analysis with some additional structure borrowed from the coeffect/comonadic approach. Furthermore, we can use the same approach to encode type class constraints, where dictionaries are encoded via the effect-parameterised reader monad. The mechanism for implicitly discharging constraints is not provided here, but our discussion shows how parametric effect monads could be used to emulate implicit parameters and type-class constraints or to give their semantics.

9. Program analysis and specification

In our examples so far, effect indices have had value-level counterparts. For example, the effect set for the reader monad corresponds to the set of values being read. However, we may not necessarily want, or need, to have a semantic, value-level counterpart to our

indices – they may be purely syntactic, used for analysis of programming properties and subsequent specifications for verifying program invariants. We show two examples in this section.

9.1 Data access

Stencil computations are a common idiom for array programming, in which an array is calculated by applying a function at each possible index of the array to compute a new cell value, possibly based on the neighbouring cells related to the current index. For example, convolution operations and the Game of Life are stencil computations. One-dimensional stencil computations can be captured by functions of type $(Array\ Int\ a, Int) \to b$ which describe the local behaviour of the stencil, *e.g.* (ignoring boundary cases here):

$$\begin{aligned}&localMean :: (Array\ Int\ Float, Int) \to Float\\&localMean\ (x, c) = (x\ !\ (c+1) + x\ !\ c + x\ !\ (c-1)) \setminus 3.0\end{aligned}$$

Promoting this operation to work over all indices of an array is provided by the *extend* operation of a comonad (see the previous section) on "cursored arrays" [18]. Stencil computations can be a source of low-level errors, especially when stencils are large, performing many indexing operations (as is common). Here we use our approach to embed an effect system that tracks the indexing operations relative to the cursor index (c above). We define the following parameterised, cursored array data type $CArray$ and $Stencil$ which captures stencil computations on $CArray$:

$$\begin{aligned}&\mathbf{data}\ CArray\ (r :: [*])\ a = A\ (Array\ Int\ a, Int)\\&\mathbf{data}\ Stencil\ a\ (r :: [*])\ b = S\ (CArray\ r\ a \to b)\end{aligned}$$

The parameter r has no semantic meaning; we will use effect annotations purely for analysis, and not for any computation. $Stencil$ has a parametric effect monad definition with the set union monoid over indices and the standard reader definition at the value level.

$$\begin{aligned}&\mathbf{instance}\ Effect\ (Stencil\ a)\ \mathbf{where}\\&\quad\mathbf{type}\ Plus\ (Stencil\ a)\ s\ t = Union\ s\ t\\&\quad\mathbf{type}\ Unit\ (Stencil\ a) \quad = `[]\\&\quad return\ a = A\ (\backslash\_ \to a)\\&\quad(S\ f) \gg\!= k = S\ (\lambda a \to \mathbf{let}\ (S\ f') = k\ (f\ a)\ \mathbf{in}\ f'\ a)\end{aligned}$$

Our key effectful operation is an operation for relative indexing which induces an effect annotation containing the relative index:

$$\begin{aligned}&ix :: (Val\ (IntT\ x)\ Int) \Rightarrow IntT\ x \to Stencil\ a\ `[IntT\ x]\ a\\&ix\ n = S\ (\lambda(A\ (a, c)) \to a\ !\ (c + toVal\ n))\end{aligned}$$

with lifting of the kind Nat of natural numbers types to a type of integers $IntT$ with a sign kind over Nat:

$$\begin{aligned}&\mathbf{data}\ Sign\ n = Pos\ n \mid Neg\ n\\&\mathbf{data}\ IntT\ (n :: Sign\ Nat) = IntT\end{aligned}$$

Thus, the effect system collects a set of relative indices. We can then redefine $localMean$ as:

$$\begin{aligned}&localMean :: Stencil\ Float\\&\quad`[IntT\ (Neg\ 1), IntT\ (Pos\ 0), IntT\ (Pos\ 1)]\ Float\\&localMean = \mathbf{do}\ a \leftarrow ix\ (IntT :: (IntT\ (Pos\ 0)))\\&\qquad\qquad b \leftarrow ix\ (IntT :: (IntT\ (Pos\ 1)))\\&\qquad\qquad c \leftarrow ix\ (IntT :: (IntT\ (Neg\ 1)))\\&\qquad\qquad return\ \$\ (a + b + c)\ /\ 3.0\end{aligned}$$

We observe that, in practice, many stencils have a very regular shape to some fixed depth. We can therefore define type-level functions for generating stencil specifications of these shapes. For example, we define "forward" oriented stencils to depth d as:

$$\begin{aligned}&\mathbf{type}\ Forward\ d = AsSet\ ((IntT\ (Pos\ 0))\ `:\ (Fwd\ d))\\&\mathbf{type\ family}\ Fwd\ d\ \mathbf{where}\\&\quad Fwd\ 0 = `[]\\&\quad Fwd\ d = (IntT\ (Pos\ n))\ `:\ (Fwd\ (d-1))\end{aligned}$$

We can similarly define a backwards definition, and together form the common symmetrical stencil pattern:

$$\begin{aligned}&\mathbf{type}\ Symm\ d =\\&AsSet\ ((IntT\ (Pos\ 0))\ `:\ (Append\ (Fwd\ d)\ (Bwd\ d)))\end{aligned}$$

We can then give $localMean$ the shorter signature:

$$localMean :: Stencil\ Float\ (Symm\ 1)\ Float$$

Such signatures provide specifications on stencils from which the type system checks whether the stencil function is correctly implemented, *i.e.*, not missing any indices. The type system will reveal to us any omissions. For example, the following buggy definition raises a type error since the negative index -1 is missing:

$$\begin{aligned}&localMean :: Stencil\ Float\ (Symm\ 1)\ Float\\&localMean = \mathbf{do}\ a \leftarrow ix\ (Pos\ Z)\\&\qquad\qquad b \leftarrow ix\ (Pos\ (S\ Z))\\&\qquad\qquad return\ \$\ (a + b + b)\ /\ 3.0\end{aligned}$$

In this effect system, effects are ordered by the superset relation since we want to recognise when indices are omitted. For example, the effect of $localMean$ is a subset of $(Symm\ 1)$ as an index is missing, therefore $localMean$'s effect is *not* a subeffect of $(Symm\ 1)$ hence cannot be 'upcast' to it. Thus, effects are overapproximated here by the subset.

9.2 Counter

Prior to the work on effect parameterised monads, Danielsson proposed the $Thunk$ "annotated monad" type [6], which is parameterised with natural numbers: 0 for $return$, and addition on the natural number parameters for $(\gg\!=)$. We call this the *counter* effect monad as it can be used for counting aspects of computation, such as time bounds in the case of Danielsson, or computation steps.

$$\begin{aligned}&\mathbf{data}\ Counter\ (n :: Nat)\ a = Counter\ \{forget :: a\}\\&\mathbf{instance}\ Effect\ Counter\ \mathbf{where}\\&\quad\mathbf{type}\ Unit\ Counter = 0\\&\quad\mathbf{type}\ Plus\ Counter\ n\ m = n + m\\&\quad return\ a = Counter\ a\\&\quad(Counter\ a) \gg\!= k = Counter\ .\ forget\ \$\ k\ a\\&tick :: a \to Counter\ 1\ a\\&tick\ x = Counter\ x\end{aligned}$$

Thus we can use $tick$ to denote some increment in computation steps or time. This effect system can be used to prove complexity bounds on our programs. For example, we can prove that the map function over a sized vector is linear in its size:

$$\begin{aligned}&\mathbf{data}\ Vector\ (n :: Nat)\ a\ \mathbf{where}\\&\quad Nil\ \ :: Vector\ 0\ a\\&\quad Cons :: a \to Vector\ n\ a \to Vector\ (n+1)\ a\\&map :: (a \to Count\ m\ b)\\&\qquad\to Vector\ n\ a \to Count\ (n * m)\ (Vector\ n\ b)\\&map\ f\ Nil = return\ Nil\\&map\ f\ (Cons\ x\ xs) = \mathbf{do}\ x' \leftarrow f\ x\\&\qquad\qquad\qquad\qquad xs' \leftarrow map\ f\ xs\\&\qquad\qquad\qquad\qquad return\ (Cons\ x'\ xs')\end{aligned}$$

i.e., if we apply a function which takes m steps to a list of n elements, then this takes $n * m$ steps.

The above is a slight simplification of the actual implementation (which can be found in `Control.Effect.Counter`) since type-checking operations on type-level natural numbers are currently a little underpowered: the above does not type check. Instead, if we

implement our own inductive definitions of natural numbers, and the corresponding $+$ and $*$ operations, then the above type checks, and the type system gives us a kind of complexity proof. The only difference to the implementation and the above is that we do not get the compact natural number syntax in the types.

10. Category theory definition

Previous theoretical work introduced parametric effect monads [12, 19] (where in [19] we called them indexed monads). For completeness we briefly show the formal definition, which shows that parametric effect monads arise as a mapping between a monoid of effects $(\mathcal{I}, \bullet, I)$ and the monoid of endofunctor composition (which models sequential composition).

Parametric effect monads comprise a functor $\mathsf{T} : \mathcal{I} \to [\mathcal{C}, \mathcal{C}]$ (*i.e.*, an indexed family of endofunctors) where $\mathcal{I}$ is the category providing effect annotations. This category $\mathcal{I}$ is taken as a strict monoidal category $(\mathcal{I}, \bullet, I)$, *i.e.*, the operations on effect annotations are defined as a binary functor $\bullet : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ and an object $I \in \mathcal{I}$. The T functor is then a parametric effect monad when it is a *lax monoidal functor*, mapping the strict monoidal structure on $\mathcal{I}$ to the strict monoid of endofunctor composition $([\mathcal{C}, \mathcal{C}], \circ, I_C)$. The operations of the lax monoidal structure are thus:

$$\eta_1 : I_{\mathcal{C}} \stackrel{\cdot}{\to} \mathsf{T}1 \qquad \mu_{F,G} : \mathsf{T}F \circ \mathsf{T}G \stackrel{\cdot}{\to} \mathsf{T}(F \bullet G)$$

These lax monoidal operations of T match the shape of the regular monad operations. Furthermore, the standard associativity and unitality conditions of the lax monoidal functor give coherence conditions to η_1 and $\mu_{F,G}$ which are analogous to the regular monad laws, but with added indices, *e.g.*, $\mu_{1,G} \circ (\eta_1)_{\mathsf{T}G} = id_{\mathsf{T}G}$.

In our definition here, we have used the "extension form" in terms of $(\gg\!=)$, as is traditional in Haskell. This is derived from the μ (join) operation by $x \gg\!= f = (\mathsf{T}f \circ \mu)x$.

Indexed monads collapse to regular monads when $\mathcal{I}$ is a single-object monoidal category. Thus, indexed monads generalise monads. Note that indexed monads are *not* indexed families of monads. That is, for all indices $F \in obj(\mathcal{I})$ then $\mathsf{T}F$ may not be a monad.

11. Related notions

Parameterised monads and indexed monads Katsumata used the phrase *parametric effect monads* [12], which we adopted here. In previous work, we referred to such structures as *indexed monads* [19], but we recognise this clashes with other earlier uses of the term. Most notably, Haskell already has an "indexed monad" library (`Control.Monad.Indexed`) which provides an interface for Atkey's notion of *parameterised monad* [1] with operations:

$$ireturn :: a \to m\ i\ i\ a$$
$$ibind :: (a \to m\ j\ k\ b) \to m\ i\ j\ a \to m\ i\ k\ b$$

The second and third indices on m can be read like Hoare triples (which McBride shows when embedding a similar definition to the above in Haskell [16]), where $m\ i\ j\ a$ is the triple $\{i\}\, a\, \{j\}$, *i.e.*, a computation starts with pre-condition i, and computes a value of type a providing post-condition j. An alternate view is that m here is indexed by the source and target types of morphisms, where $ireturn$ is indexed by identities and $ibind$ exhibits composition.

We can encode the same approach with our *Effect* class. Using data kinds, we define a kind of morphisms $Morph$ inhabited by either the identity Id or a morphism $M\ a\ b$ with source a and target b. The type, together with the effect monad are defined as:

$$\mathbf{data}\ Morph\ a\ b = M\ a\ b \mid Id$$
$$\mathbf{newtype}\ T\ (i :: Morph * *)\ a = T\ a$$
$$\mathbf{instance}\ Effect\ (T :: ((Morph * *) \to * \to *))\ \mathbf{where}$$
$$\quad \mathbf{type}\ Unit\ T = Id$$
$$\mathbf{type}\ Plus\ T\ (M\ a\ b)\ (M\ c\ d) = M\ a\ d$$
$$\mathbf{type}\ Plus\ T\ Id\ (M\ a\ b) = M\ a\ b$$
$$\mathbf{type}\ Plus\ T\ (M\ a\ b)\ Id = M\ a\ b$$
$$\mathbf{type}\ Inv\ \ T\ (M\ a\ b)\ (M\ c\ d) = c{\sim}d$$
$$return\ a = T\ a$$
$$(T\ x) \gg\!= k = \mathbf{let}\ T\ y = k\ x\ \mathbf{in}\ T\ y$$

We use the Inv constraint family to force the target type of the left morphism to match the source type of the right morphism. Thus, Hoare logic-style reasoning can be encoded in our framework, but further exploring program logics is a topic for future work.

Effect handlers *Algebraic effects handlers* provide a representation of effects in terms of effectful operations (rather than an encoding as with monads) and equations on these (*e.g.*, [2, 23]). This is a change of perspective. The monadic approach tends to start with the encoding of effects, and later consider the effect-specific operations. The algebraic effects approach starts with the operations and later considers the encoding as the free structure arising from the operations and their equations. This provides a flexible solution to the problems of granularity and compositionality for monads.

Recent work by Kammar, Lindley, and Oury embeds a system of effect handlers in Haskell with a DSL [11]. The aims are similar to ours, but the approach is different. Our approach can be embedded in GHC as is, without any additional macros for encoding handlers as in the approach of Kammar *et al.*, and it provides rich type system information, showing the effects of a program. There are also some differences in power. For example, the heterogeneous typing of state provided by parametric effect monads is not possible with the current handler approach; we could not encode the update writer example from Section 4.1. However, effect handlers offer much greater compositionality, easily allowing different kinds of effect to be combined in one system.

It is our view that parametric effect monads are an intermediate approach between using monads and full algebraic effects.

As mentioned in the introduction, an alternate solution to the coarse-granularity of monads is to introduce type classes for each effectful operations where type class constraints act as effect annotations (see *e.g.* [15]). A similar approach is taken by Kiselyov *et al.* in their library for *extensible effects*, which has similarities to the effect handlers approach [13]. By the type-class constraint encoding, these effect systems are based on sets with union and ordering by subsets. Our approach allows effect systems based on different foundations (an arbitrary monoid with a preorder), *e.g.*, the number-indexed counter monad (Section 9.2), the $Maybe$-indexed update monad (Section 4.1), and the ordering of effects by supersets for array indexing effects (Section 9.1).

12. Epilogue

A whole menagerie of type system features were leveraged in this paper to give a shallow embedding of effect systems in Haskell types (without macros or custom syntax). The newest *closed family* extension to GHC was key to embedding sets in types, which was core to some of our examples.

While there is a great deal of power in the GHC type system, a lot of boilerplate code was required. Frequently, we have made almost identical type- and value-level definitions. Languages with richer dependent types are able to combine these. Going forward, it seems likely, and prudent, that such features will become part of Haskell, although care must be taken so that they do not conflict with other aspects of the core language. We also advocate for built-in type-level sets which would significantly simplify our library.

Further work is to extend our approach to allow different kinds of effect to be combined. One possible approach may be to define

a single monad type, parameterised by a set of effect annotations whose elements each describe different notions of effect.

Acknowledgements Thanks to the anonymous reviewers for their helpful feedback, Alan Mycroft for subeffecting discussions, Andrew Rice for stencil computation discussion, Michael Gale for comments on an earlier draft of this manuscript, and participants of *Fun in the Afternoon* 2014 (Facebook, London) for comments on a talk based on an early version. Thanks also to Andy Hopper for his support. This work was partly supported by CHESS.

References

[1] Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*. Cambridge Univ. Press, 2006.

[2] Andrej Bauerm and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 2014.

[3] Max Bolingbroke. Constraint Kinds for GHC, 2011. `http://blog.omega-prime.co.uk/?p=127` (Retreived 24/06/14).

[4] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proceedings of ESOP*, volume 8410 of *LNCS*, pages 351–370. Springer, 2014.

[5] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of 10th International Conference on Functional Programming*, pages 241–253. ACM, 2005.

[6] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, volume 43, pages 133–144. ACM, 2008.

[7] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of POPL 2014*, pages 671–684, 2014.

[8] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proceedings of ESOP*, volume 8410 of *LNCS*, pages 331–350. Springer, 2014.

[9] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986.

[10] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the symposium on Principles of Programming Languages*, pages 303–310. ACM, 1991.

[11] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th International Conference on Functional Programming*, pages 145–158. ACM, 2013.

[12] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of symposium Principles of Programming Languages*, pages 633–646. ACM, 2014.

[13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of 2013 symposium on Haskell*, pages 59–70. ACM, 2013.

[14] J.R. Lewis, J. Launchbury, E. Meijer, and M.B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of Principles of Programming Languages*, page 118. ACM, 2000.

[15] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of 22nd symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

[16] Conor McBride. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (to appear)*, 2011.

[17] Flemming Nielson and Hanne Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.

[18] Dominic Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *Proceedings of 5th workshop on Declarative Aspects of Multicore Programming*, pages 15–24. ACM, 2010.

[19] Dominic Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *arXiv:1401.5391*, 2014.

[20] Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *Functional and Logic Programming*, volume 6009/2010, pages 56–71. Springer Berlin, 2010.

[21] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *ICALP (2)*, volume 7966 of *LNCS*, pages 385–397. Springer, 2013.

[22] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of ICFP*, pages 50–61. ACM, 2006.

[23] G. Plotkin and M. Pretnar. A logic for algebraic effects. In *Logic in Computer Science, 2008. LICS'08*, pages 118–129. IEEE, 2008.

[24] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

[25] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

A. Typed value-level list sorting

The following gives the value-level definitions of sorting to normalise lists for the set representation, referenced from Section 3.

The top-level *bubble* function is defined by a type class:

```
class Bubbler s s' where
  bubble :: Set s → Set s' → Set (Bubble s s')

instance Bubbler s '[] where
  bubble s Empty = s

instance (Bubbler s t, Passer (Bubble s t))
         ⇒ Bubbler s (e ': t) where
  bubble s (Ext _ t) = pass (bubble s t)
```

The individual bubble sort pass is defined also by a type class, so that the embedded constraints in the 'swapping' case are captured:

```
class Passer s where pass :: Set s → Set (Pass s)

instance Passer '[] where pass Empty = Empty

instance Passer '[e] where
  pass (Ext e Empty) = Ext e Empty

instance (Passer ((Max e f) ': s), OrdH e f)
         ⇒ Passer (e ': f ': s) where
  pass (Ext e (Ext f s)) =
    Ext (minH e f) (pass (Ext (maxH e f) s))
```

B. Value comparison of variable-value mappings

Section 4 uses mappings $v :\rightarrow t$ between variables and values. Here, we add the value-level comparison operation. Scoped type variables are used along with the data type *Proxy* used for giving a value-level proxy to a type of kind k, *i.e.*, $Proxy :: Proxy\ k$.

```
select :: forall j k a b . (Chooser (CmpSymbol j k)) ⇒
          Var j → Var k → a → b → Select j k a b
select _ _ x y = choose
  (Proxy :: (Proxy (CmpSymbol j k))) x y

instance (Chooser (CmpSymbol u v)) ⇒
         OrdH (u :→ a) (v :→ b) where
  minH (u :→ a) (v :→ b) = Var :→ (select u v a b)
  maxH (u :→ a) (v :→ b) = Var :→ (select u v b a)

class Chooser (o :: Ordering) where
  choose :: (Proxy o) → p → q → (Choose o p q)

instance Chooser LT where choose _ p q = p
instance Chooser EQ where choose _ p q = p
instance Chooser GT where choose _ p q = q
```