Accepted at Haskell '14

Effect Handlers in Scope

Nicolas Wu
University of Oxford
nicolas.wu@cs.ox.ac.uk

Tom Schrijvers
Ghent University
tom.schrijvers@ugent.be

Ralf Hinze
University of Oxford
ralf.hinze@cs.ox.ac.uk

Abstract

Algebraic effect handlers are a powerful means for describing effectful computations. They provide a lightweight and orthogonal technique to define and compose the syntax and semantics of different effects. The semantics is captured by handlers, which are functions that transform syntax trees.

Unfortunately, the approach does not support syntax for scoping constructs, which arise in a number of scenarios. While handlers can be used to provide a limited form of scope, we demonstrate that this approach constrains the possible interactions of effects and rules out some desired semantics.

This paper presents two different ways to capture scoped constructs in syntax, and shows how to achieve different semantics by reordering handlers. The first approach expresses scopes using the existing algebraic handlers framework, but has some limitations. The problem is fully solved in the second approach where we introduce higher-order syntax.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Languages

Keywords Haskell, effect handlers, modularity, monads, syntax, semantics

1. Introduction

Effect handlers [13] have established themselves as a lightweight and compositional means of describing effectful computations. At the heart of the solution is the idea that a program is composed out of fragments of syntax that are often orthogonal to one another. Those fragments can then in turn be given a semantics by handlers that systematically deal with different effects.

One aspect of handlers that has not received much attention are scoping constructs. Examples of this are abound: we see it in constructions for control flow, such as while loops and conditionals, but we also see this in pruning nondeterminsitic computations, exception handling, and multi-threading. The current work on effect handlers considers scoping to be in the province of handlers, who not only provide semantics but also delimit the scope of their effects.

However, as this paper illustrates, using handlers for scoping has an important limitation. The reason is that the semantics of handlers are not entirely orthogonal: applying handlers in different orders may give rise to different interactions between effects—perhaps the best known example is that of the two possible interactions between state and non-determinism. The flexibility of ordering handlers is of course crucial: we need control over the interaction of effects to obtain the right semantics for a particular application. However, if handlers double as scoping constructs, the two roles may be at odds: one order of the handlers provides the right scopes and the other order provides the right semantics. Unfortunately, we cannot have it both ways.

This paper solves the dilemma by shifting the responsibility of creating scopes from handlers to syntax. This way we can safely reorder handlers to control the interaction semantics while scoping is unaffected. Of course, handlers are still responsible for assigning a semantics to syntax that create scopes.

The specific contributions of this paper are:

1. We provide several examples that demonstrate the problem of scoping through handlers: pruning nondeterministic choices, exception handling, and multi-threading.
2. We provide two different approaches for handling scoping through syntax.
   (a) First, we use syntax within the existing effect handlers framework to delimit scopes, and show how to write a handler that works with this syntax. This solution is conceptually lightweight, since it makes use of syntax and is nothing other than another handler. However, it is not general enough to capture syntax that truly requires programs as arguments.
   (b) As a second solution we provide higher-order syntax that truly allows to embed programs within scoping constructs. This solution is more general, but requires a substantial adaptation of the effect handlers approach.
3. We illustrate both syntax scoping approaches on the examples and show how they effectively solve the problem.

The remainder of this paper is structured as follows. The first part provides background on the effect handlers approach through a number of examples and sets up the necessary infrastructure. We start with a gentle introduction to handling backtracking computations in Section 2. In Section 3 we prepare the ground for more modular syntax by using the datatypes à la carte approach. We demonstrate this modularity in Section 4, where we show how state can be added to nondeterministic computation. We then show how handlers can span different syntax signatures in Section 5.

The second part of this paper focuses on scoped effects. Section 6 builds grammars to parse input, and shows how using handlers to create local scopes imposes undesired semantics. In Section 7 we fix this problem by using syntax to delimit scope. Section 8 demonstrates exception handling as another example that requires scoped effects, which is resolved in Section 9. We show a more robust solution to the problem in Section 10, where higher-order

syntax is introduced. Section 11 gives an example where our first-order approach fails, but that can be solved with higher-order syntax. Finally, we discuss related work in Section 12 and conclude in Section 13.

2. Backtracking Computation

The effect handlers approach splits the problem of modelling behaviour into two parts. First syntax is introduced to represent the actions of interest; second, so-called handlers are written that interpret syntax trees into a semantic domain.

For instance, to model the behaviour of backtrackable computation, we use the datatype *Backtr a* to represent the syntax.

```
data Backtr a
  = Return a
  | Fail
  | Backtr a :|| Backtr a
```

Here, *Return x* represents a successful computation witnessed by *x*, *Fail* is for computations that have failed, and choice is given by $p :\!|\!|\ q$, where *p* and *q* are backtrackable computations.

This representation of backtrackable computations forms a monad, which will allow us to conveniently put syntax together.

```
instance Monad Backtr where
  return a = Return a

  Return a >>= r = r a
  Fail     >>= r = Fail
  (p :|| q) >>= r = (p >>= r) :|| (q >>= r)
```

As an example, consider how we might solve the well-known knapsack problem, where we choose with replacement elements from *vs* that sum to *w*. Assuming the values are all positive, the following is a naive solution to the problem that goes through all the different possibilities.

```
knapsack :: Int → [Int] → Backtr [Int]
knapsack w vs | w < 0  = Fail
              | w == 0 = return []
              | w > 0  = do v ← select vs
                            vs' ← knapsack (w − v) vs
                            return (v : vs')
```

This makes use of the *select* function that turns a list of values into backtrackable computations:

```
select :: [a] → Backtr a
select = foldr (:||) Fail · map Return
```

We fail when there are no values left to select, otherwise, we offer the choice between a given value and the remaining ones.

The resulting construction of *knapsack* 3 [1,2] is a tree that expresses the decisions that are made when choosing from the list of values. This is the syntactic tree we are interested in.

```
knapsack 3 [3,2,1] = Return [3] :|| ((Fail :|| (Fail :||
  (Return [2,1] :|| Fail))) :|| ((Fail :|| (Return [1,2] :||
  ((Fail :|| (Fail :|| (Return [1,1,1] :|| Fail))) :|| Fail))) :|| Fail))
```

We can extract successful computations by making use of the function *allsols p*, which produces a list of all the solutions that are generated by the program *p*. In fact, *allsols* is our first example of a *handler*: it takes the syntax of a backtrackable computation, and handles it to produce a list of all solutions.

```
allsols :: Backtr a → [a]
allsols (Return a) = [a]
allsols (Fail)     = []
allsols (p :|| q)  = allsols p ++ allsols q
```

Putting the different parts together, we can use *allsols* to capture the solutions to the *knapsack* problem:

```
allsols (knapsack 3 [3,2,1]) = [[3],[2,1],[1,2],[1,1,1]]
```

We characterize *allsols* as a handler since it turns the syntax tree of nondeterministic choices into the semantic domain of solutions.

3. Syntax Signatures

We can generalize away from backtrackable computations by defining a datatype that is parametric in the signature of the syntax. We factor syntax for programs into the *Return* constructor and a constructor *Op* that allows us to incorporate operations of interest from some signature *sig*.

```
data Prog sig a
  = Return a                  -- pure computations
  | Op (sig (Prog sig a))     -- impure computations
```

For instance, the operations that give rise to computations of type *Backtr a* are captured by the signature functor *Nondet*, such that $\mathit{Backtr}\ a \cong \mathit{Prog}\ \mathit{Nondet}\ a$.

```
data Nondet cnt
  = Fail'
  | cnt :||' cnt
```

The type argument *cnt* marks the recursive components, which in this context are continuations into some other syntactic construct. Here, and elsewhere in the paper, we assume that the functor instance is automatically derived.

This abstraction has bought us an important benefit: the *Prog sig* type forms a monad whenever *sig* is a functor (it is the free monad for the functor *sig*), which helps us to easily compose programs together from constituent parts.

```
instance (Functor sig) ⇒ Monad (Prog sig) where
  return v = Return v

  Return v >>= prog = prog v
  Op op    >>= prog = Op (fmap (>>=prog) op)
```

This monad instance allows us to conveniently write programs that piggyback on Haskell's **do** notation, which we will see later on.

In the monad instance, we can read $(\gg\!=)$ as substitution, where each fragment of syntax represents first-order terms with variables: *Return v* is a variable, and *Op op* is a compound term. Another reading is that *Prog sig a* represents computations or programs: *Return v* is a pure computation, *Op op* is an impure computation, and here $(\gg\!=)$ represents the sequential chaining of computations.

Syntax Infrastructure So far we have looked at only one syntax signature. More generally, we will be dealing with several different effects that work together in a single program, and so we need a means of flexibly composing signatures, where each signature captures syntax that encodes a particular effect.

Perhaps the simplest way to compose signatures is with the coproduct, where two signatures sig_1 and sig_2 are combined into a single one.

```
data (sig1 + sig2) cnt = Inl (sig1 cnt) | Inr (sig2 cnt)
```

Handlers over such signatures must be run one after the other, each dealing with part of the signature they recognize. In practice, this becomes a little cumbersome since the handlers have to carefully invoke the right mixture of *Inl* and *Inr* constructors to get access to the syntax they are interested in.

To fix this, we want to be able to inject and project constructors for some syntax into some larger language, and this is where the *datatypes à la carte* technique [14] shines, by precisely expressing the relationship between families of syntax:

```
class (Functor sub, Functor sup) ⇒ sub ⊂ sup where
  inj :: sub a → sup a
  prj :: sup a → Maybe (sub a)
instance Functor sig ⇒ sig ⊂ sig where
  inj = id
  prj = Just
```

The coproduct fits into this scheme nicely, as is evidenced by the following instances:

```
instance (Functor sig1, Functor sig2) ⇒
         sig1 ⊂ (sig1 + sig2) where
  inj           = Inl
  prj (Inl fa)  = Just fa
  prj _         = Nothing
instance (Functor sig1, sig ⊂ sig2) ⇒
         sig ⊂ (sig1 + sig2) where
  inj           = Inr · inj
  prj (Inr ga) = prj ga
  prj _         = Nothing
```

This gives a nice way of composing signatures, and makes it possible to inject syntax into programs over those signatures.

```
inject :: (sub ⊂ sup) ⇒ sub (Prog sup a) → Prog sup a
inject = Op · inj
```

As well as conveniently building operations of a program, we have a way of extracting operations from that program for inspection.

```
project :: (sub ⊂ sup) ⇒ Prog sup a → Maybe (sub (Prog sup a))
project (Op s) = prj s
project _      = Nothing
```

This projection returns the syntax of interest when we are dealing with an *Op*, and otherwise returns *Nothing*.

With smart constructors and destructors we can make the embedding of syntax into a wider context relatively painless. We make use of *pattern synonyms* and *view patterns*, which are recent extensions to Haskell. This allows us to create a new pattern called *Fail*, which works on programs of type *Prog sig a* for any signature where *Nondet* ⊂ *sig*. In the following pattern declaration, the function *project* is applied at the site of the pattern, and if the result is *Just Fail′*, the match is a success.

```
pattern Fail ← (project → Just Fail′)
fail :: (Nondet ⊂ sig) ⇒ Prog sig a
fail = inject Fail′
```

For choice we can again use the same approach:

```
pattern p :[] q ← (project → Just (p :[]′ q))
(:[]) :: (Nondet ⊂ sig) ⇒ Prog sig a → Prog sig a → Prog sig a
p :[] q = inject (p :[]′ q)
```

This time applying *project* to a program might yield *Just* $(p :[]' q)$, and if this is the case we bind the variables in that context to those in the pattern $p :[] q$.

We will be assembling programs using coproducts, Since our goal is to inject syntax into some accumulating collection of syntax functors we need a base case, and this is provided by *Void*, which is the signature for empty syntax:

```
data Void cnt
```

Using this syntax functor as a signature results in programs of type *Prog Void a*, where it is impossible to use the *Op* constructor. We can nevertheless extract values from such programs:

```
run :: Prog Void a → a
run (Return x) = x
```

This handler is usually the last one to be run, since it extracts a final value from a program with no more syntax.

When providing semantics for programs with signatures made up of coproducts, we will define handlers that deal with a specific part of that signature, and leaves the rest untouched. This is the key to modular semantics, allowing us to focus on the interesting details. We use *Other* to represent the other syntax that is not interesting in a given context:

```
pattern Other s = Op (Inr s)
```

For instance, we can evaluate a program with *Nondet* syntax on the left of its signature by using the *solutions* function, which is highly modular since *sig* can be an arbitrary signature:

```
solutions :: (Functor sig) ⇒ Prog (Nondet + sig) a → Prog sig [a]
solutions (Return a) = return [a]
solutions (Fail)     = return [ ]
solutions (p :[] q)  = liftM2 (++) (solutions p) (solutions q)
solutions (Other op) = Op (fmap solutions op)
```

This is a lifted, or monadized, version of *allsols* where there might be syntax other than that given by *Nondet* involved. We can recover *allsols* by noticing that *Backtr a* ≅ *Prog* (*Nondet* + *Void*) *a*, and adapting the definition to fit our more modular framework:

```
allsols :: Prog (Nondet + Void) a → [a]
allsols = run · solutions
```

This approach uses *run* to extract results from a program that has no more syntax.

4. Composing Semantics

The main point of the effect handlers approach is that not only the syntax of different effects, but also their semantics can be trivially composed. In other words, effect handlers provide modular semantics. This section illustrates that point by combining nondeterminism with state.

4.1 The State Effect

Stateful operations are modelled with the assumption that there exists some underlying state *s*, which can be updated with the operation *put s*, and retrieved with *get*. The corresponding syntax is as follows:

```
data State s cnt
   = Get′ (s → cnt)
   | Put′ s cnt
pattern Get k ← (project → Just (Get′ k))
get :: (State s ⊂ sig) ⇒ Prog sig s
get = inject (Get′ return)
pattern Put s k ← (project → Just (Put′ s k))
put :: (State s ⊂ sig) ⇒ s → Prog sig ()
put s = inject (Put′ s (return ()))
```

If we want to execute a stateful computation, then we can use the following handler, which takes an initial state, and a program that contains state manipulating syntax to return a residual program which returns an output state.

```
runState :: Functor sig ⇒
   s → Prog (State s + sig) a → Prog sig (s,a)
runState s (Return a) = return (s,a)
runState s (Get k)    = runState s (k s)
runState s (Put s′ k) = runState s′ k
runState s (Other op) = Op (fmap (runState s) op)
```

This works by carrying around the appropriate state in recursive calls: when a new state is inserted with *Put* s', then this new state s' replaces the previous one.

4.2 Combining State and Nondeterminism

Now we can assign a semantics to syntactic programs that combine nondeterminism and state by providing the semantics for both effects separately: we just compose both handlers. The first handler tackles one effect in the initial program while the second handler tackles the other in the residual program.

It is vital to note that we have a degree of freedom when composing two handlers: we can choose which handler to apply first. For instance, for the *runState* and *solutions* handlers we can choose between either *runLocal* or *runGlobal*:

```
runLocal :: Functor sig ⇒
  s → Prog (State s + Nondet + sig) a → Prog sig [(s, a)]
runLocal s = solutions · runState s

runGlobal :: Functor sig ⇒
  s → Prog (Nondet + State s + sig) a → Prog sig (s, [a])
runGlobal s = runState s · solutions
```

These two composite semantics are not equivalent; they differ in how the two effects interact. Here we get two flavors of nondeterministic state, *local* and *global* state. In *runLocal*, each branch of the nondeterministic computation has its own local copy of the state, while in *runGlobal* there is one state shared by all branches. The difference between the two is also apparent at the type level: *runLocal* returns a list of different final values with their associated states, one state for each solution in the backtracking. The type of *runGlobal* reveals that it produces a list of alternative solutions and only one final state.

The fact that we get different semantics through different compositions is a great benefit of the effect handlers approach: in return for writing the handlers in modular style, we get multiple interaction semantics for free!

The following example illustrates that both flavors of nondeterministic state are useful for different purposes. In fact, the example even shows the use of two different semantics for the same program and involves a third handler that counts how many number choices are made.

```
choices :: (Nondet ⊂ sig, State Int ⊂ sig)
        ⇒ Prog sig a → Prog sig a
choices (Return a) = return a
choices (Fail)     = fail
choices (p :[] q)  = incr ≫ (choices p [] choices q)
choices (Op op)    = Op (fmap choices op)

incr :: (State Int ⊂ sig) ⇒ Prog sig ()
incr = get ≫= put · (succ :: Int → Int)
```

The counting is performed by *incr*, which simply increments an *Int* stored in the state. We evaluate an *incr* every time we encounter a choice, and then recursively count choices in each branch.

In order to apply the *choices* handler to the *knapsack* example, we need to adapt the definitions of both *knapsack* and *select* to the modular setting. Thankfully, this involves only a little more than providing slightly different type signatures; the code body remains unchanged, except that we use the smart constructors *fail* and ([]):

```
knapsack :: (Nondet ⊂ sig) ⇒ Int → [Int] → Prog sig [Int]
select :: (Nondet ⊂ sig) ⇒ [a] → Prog sig a
```

Now we can observe that both global and local state give us different information. The global version tells us how many choice points are explored to find all solutions:

```
> (run · runGlobal (0 :: Int) · choices) (knapsack 3 [3, 2, 1])
(12, [[3], [2, 1], [1, 2], [1, 1, 1]])
```

In contrast, the local version tells us exactly how deep in the tree of choices each individual answer is found:

```
> (run · runLocal (0 :: Int) · choices) (knapsack 3 [3, 2, 1])
[(1, [3]), (5, [2, 1]), (5, [1, 2]), (9, [1, 1, 1])]
```

The information provided by the global version cannot be reconstructed from the local information, and vice versa.

Summary In this section we have seen how easy it is to compose the semantics of two orthogonal features, nondeterminism and state, with effect handlers. In fact, we can express the two interactions of state and nondeterminism simply by composing their handlers in different orders.

5. Cut and Call

The effect handlers approach does not force us to write orthogonal handlers. This section shows that we can extend nondeterminism with a non-orthogonal feature. The *Cutfail* operation immediately ends the search with failure, dropping all extant unexplored branches. Hence, there is clearly interaction with *Nondet*.

```
data Cut cnt = Cutfail'
pattern Cutfail ← (project → Just Cutfail')
cutfail :: (Cut ⊂ sig) ⇒ Prog sig a
cutfail = inject Cutfail'
```

The expression *call p*, defined in terms of the *go* handler, delimits the action of *Cutfail* in a program *p*.

```
call :: (Nondet ⊂ sig) ⇒ Prog (Cut + sig) a → Prog sig a
call p = go p fail where
  go :: (Nondet ⊂ sig) ⇒
        Prog (Cut + sig) a → Prog sig a → Prog sig a
  go (Return a)  q = return a [] q
  go (Fail)      q = q
  go (Cutfail)   q = fail
  go (p1 :[] p2) q = go p1 (go p2 q)
  go (Other op)  q = Op (fmap (flip go q) op)
```

The *go p q* handler accumulates in its second parameter *q* the unexplored alternatives to *p*. When *go* encounters a *Return* or a *Fail*, it explores the alternatives in *q*. When a *Cutfail* is encountered, the computation fails immediately, without exploring any alternatives. At a branching, *go* explores the left branch and adds the right branch to the unexplored alternatives.

Often *cutfail* is used in the form of *cut*, which can be defined as:

```
cut :: (Nondet ⊂ sig, Cut ⊂ sig) ⇒ Prog sig ()
cut = skip [] cutfail

skip :: Monad m ⇒ m ()
skip = return ()
```

This commits the computation to the current branch, pruning any unexplored alternatives. For example, *once p* commits to the first solution that is found in *p*.

```
once :: (Nondet ⊂ sig) ⇒ Prog (Cut + sig) b → Prog sig b
once p = call (do x ← p; cut; return x)
```

This way we can compute only the first *knapsack* solution as follows.

```
> (run · solutions · once) (knapsack 3 [3, 2, 1])
[[3]]
```

In summary, we can write non-orthogonal handlers like *call* just as easily as modular ones like *solutions* and they play nicely together.

However, there lurks a deep problem in these murky waters, where *call* does not always behave quite as we expect it to. We explore this in the next section with a different example.

6. Grammars

This section establishes the central problem tackled in this paper. We call a handler like *call* a *scoping* handler, because it not only provides the semantics for particular syntax, but also creates a local scope in which the impact of an effect is contained. The two roles of scoping handlers can be fundamentally at odds with one another: different orders of handlers affect the interaction semantics, while different scopes affect the extent of an effect's impact. With scoping handlers these two choices are not independent; we cannot affect one without the order. Yet, often we need to control both separately. This section illustrates that point on grammars.

Grammars Grammars can be expressed compactly using syntax signatures from a remarkably small base: the *Symbol* functor represents syntax that matches a single symbol from some source of characters:

```
data Symbol cnt = Symbol' Char (Char → cnt)

symbol :: (Symbol ⊂ sig) ⇒ Char → Prog sig Char
symbol c = inject (Symbol' c return)
```

The constructor *symbol c* attempts to match *c* with the current input, and if it succeeds, passes the value of *c* on to its continuation. For instance, we can build a digit recognizer with the following:

```
digit :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Char
digit = foldr (▯) fail (fmap symbol ['0'..'9'])
```

This nondeterministically attempts to match all of the digits, and fails if this is not possible.

The combinators *many* and $many_1$ will be familiar to readers who have worked with grammar libraries. Their definitions encode an accumulation of values from nondeterministic programs:

```
many, many1 :: (Nondet ⊂ sig) ⇒ Prog sig a → Prog sig [a]
many  p = many1 p ▯ return []
many1 p = do a ← p; as ← many p; return (a : as)
```

Both of these functions build nondeterminism into the output of a program that supports *Nondet* syntax.

The *parse xs* handler takes a grammar to a nondeterministic program. It resolves the *Symbol c k* constructors by matching *c* against the first element in the list of characters *xs*, turning it into failure when the match fails, or passing *c* on to the continuation *k* if it succeeds.

```
parse :: (Nondet ⊂ sig) ⇒
  [Char] → Prog (Symbol + sig) a → Prog sig a
parse []     (Return a)   = return a
parse (x:xs) (Return a)   = fail
parse []     (Symbol c k) = fail
parse (x:xs) (Symbol c k)
  | x == c                = parse xs (k x)
  | otherwise             = fail
parse xs     (Other op)   = Op (fmap (parse xs) op)
```

This handler also fails if the input is not entirely consumed, or if the grammar expects more symbols.

Parsing arithmetic expressions which are made up of sums and products can be done using the traditional recipe, where an *expr* deals with sums, and a *term* deals with with products. We return the result of evaluating the payload directly.

```
expr :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Int
expr = do i ← term; symbol '+'; j ← expr; return (i + j)
     ▯ do i ← term; return i
term :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Int
term = do i ← factor; symbol '*'; j ← term; return (i * j)
     ▯ do i ← factor; return i
```

The terminal case is in *factor*, which is either a string of digits, or an expression in parentheses:

```
factor :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Int
factor = do ds ← many1 digit; return (read ds)
       ▯ do symbol '('; i ← expr; symbol ')'; return i
```

To parse an expression, we simply handle the program with *parse*:

```
> (allsols · parse "2+8*5") expr
[42]
```

Grammar Refactoring We can left factor our grammars *expr* and *term* to improve efficiency. Focusing on *expr*, we factor out the common *term* prefix in the two branches.

```
expr1 :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Int
expr1 = do i ← term
           ( do symbol '+'; j ← expr1; return (i + j)
           ▯ do return i)
```

In the refactored $expr_1$ grammar, the two branches are mutually exclusive. The reason is that the first branch requires the next character in the input to be a `'+'`, while the second branch can only be followed by a `')'` or the end of the input. Hence, after seeing a `'+'` we can safely commit to the first branch and prune the second branch. Pruning the alternative should have a beneficial effect on performance because the parser will no longer unnecessarily explore the alternative.

In the previous section we introduced the control operator *cut* to commit to a successful branch. In this case, we want to commit to the first branch when a *symbol* `'+'` is encountered, so we might try the following:

```
expr2 :: (Nondet ⊂ sig, Symbol ⊂ sig) ⇒ Prog sig Int
expr2 = do i ← term
           call ( do symbol '+'; cut; j ← expr2; return (i + j)
                ▯ do return i)
```

At a first glance, this *seems* sensible: the locally placed *call* handler is needed to delimit the scope of the *cut*. After all the *cut* is only meant to prune the one alternative, and should not affect other alternatives made elsewhere in the grammar.

The Problem Alas, while the above grammar syntactically captures the desired pruning, it may come as a surprise that the handlers do not provide the desired semantics:

```
> (allsols · parse "1") expr2
[]
```

We expect the result [1], but the parse fails instead. In order to understand why this happens, we need to carefully consider the impact of the following clause in the definition of the subsidiary *go* handler of *call*.

```
go (Other op) q = Op (fmap (flip go q) op)
```

After one recursive invocation of *call* in $expr_2$, this clause matches the "other" operation *symbol* `'+'`. In effect, we can think of this execution as rewriting the body of $expr_2$ to:

```
do i ← term
   symbol '+'
   go (do cut; j ← expr2; return (i + j))
      (return i ▯ fail)
```

In other words, *symbol* `'+'` has been hoisted out of the left branch and now happens before the *call*. Hence, the input always has to contain a `'+'`; this is obviously not what we want.

A Non-Solution The problem is that we have chosen the wrong order for the *parse* and *call* handlers, which leads to the undesired interaction. The appropriate interaction is obtained by first applying *parse* and then *call*. This way there is no more *symbol* for *call* to hoist out of a branch.

Unfortunately, we cannot reorder *call* and *parse* for other reasons: *call* creates a local scope. We cannot put it anywhere else without risking that *cut* prunes more alternatives than it should. Conversely, it obviously does not make sense to apply *parse* only in the local scope of *call*. Hence, we are stuck and *call* is to blame because it unnecessarily couples scoping and semantics.

Summary This section has shown that the coupling of scoping and semantics in scoping handlers is problematic. In the following sections we look at two different solutions to this problem; both solutions decouple scoping from semantics by making scoping the province of syntax. The first solution uses lightweight syntax that fits naturally into the first-order framework of effect handlers we have been describing so far, but is prone to user errors. The second solution we look at is more robust and expressive, but requires much heavier machinery.

7. Scoped Syntax

One solution to the problem we encounter with *call* is to explicitly delimit the beginning and end of the scope of the *call*. This can be managed by giving the user syntax to explicitly set these boundaries:

```
data Call cnt = BCall' cnt | ECall' cnt
pattern BCall p ← (project → Just (BCall' p))
pattern ECall p ← (project → Just (ECall' p))
```

We will want to ensure that each *BCall* is paired with an *ECall*. Hence, they should only be exposed to the user in the form of *call'*.

```
call' :: (Call ⊂ sig) ⇒ Prog sig a → Prog sig a
call' p = do begin; x ← p; end; return x where
  begin = inject (BCall' (return ()))
  end   = inject (ECall' (return ()))
```

With these changes, the left factored expression remains syntactically the same as $expr_2$, except that *call* has been replaced by *call'*, and the signature is now more elaborate, since the program explicitly incorporates *Cut* and *Call* syntax.

```
expr3 :: (Nondet ⊂ sig, Symbol ⊂ sig, Call ⊂ sig, Cut ⊂ sig) ⇒
  Prog sig Int
expr3 = do i ← term
           call' ( do symbol '+'; cut; j ← expr3; return (i+j)
                  [] do return i)
```

We can run this with the *runCut* handler, that provides semantics to *cut* in a way that respects the scope set out by *BCall* and *ECall*.

```
> run · solutions · runCut · parse "1" $ expr3
[1]
```

While on the surface not much has changed, there is a lot more going on behind the scenes.

The *runCut* handler is now used to eliminate *Call* and *Cut* from the signature:

```
runCut :: (Nondet ⊂ sig) ⇒
  Prog (Call + Cut + sig) a → Prog sig a
runCut p = call (bcall p)
```

The definition of *runCut* is in terms of two helper functions, *bcall* and *ecall*. The interesting case for *bcall* is when it encounters a *BCall p*. In this case, the handler *call* is used to handle the code in the continuation *p* up until the matching *ecall q*, which is found by the function *ecall*.

```
bcall :: (Nondet ⊂ sig) ⇒
         Prog (Call + Cut + sig) a → Prog (Cut + sig) a
bcall (Return a) = return a
bcall (BCall p)  = upcast (call (ecall p)) >>= bcall
bcall (ECall p)  = error "Mismatched ECall!"
bcall (Other op) = Op (fmap bcall op)
```

If an *ECall* is found during the execution of *begin*, then an error is raised, since this must be a mismatched *ECall p*. An alternative to raising an error is to simply ignore the spurious *ECall*, and continue with *p*.

The function *ecall* takes a program with scoped syntax and modifies it so that any scope context is removed. The code outside of that scope is found in *ECall p*, where $p :: Prog\ (Call + Cut + sig)\ a$ is a program in its own right.

```
ecall :: (Nondet ⊂ sig) ⇒
  Prog (Call + Cut + sig) a →
  Prog (Cut + sig) (Prog (Call + Cut + sig) a)
ecall (Return a) = return (Return a)
ecall (BCall p)  = upcast (call (ecall p)) >>= ecall
ecall (ECall p)  = return p
ecall (Other op) = Op (fmap ecall op)
```

Since *call* removes all the syntax given by *Cut* from the signature, we use *upcast* to ensure that the types match our expectations. The function *upcast* simply extends a signature so that it contains an additional syntax functor. It works by shifting operations in the original signature into the right of the resulting coproduct.

```
upcast :: (Functor f, Functor sig) ⇒ Prog sig a → Prog (f + sig) a
upcast (Return x) = return x
upcast (Op op)    = Op (Inr (fmap upcast op))
```

In summary, the idea of the tagging technique is to mark the beginning and the end of a scope with syntactic operations, the tags. Handlers should takes these tags into account to determine the impact of effects.

While this tagging of scope seems like a neat solution, we find it lacking in several regards. For one, we require the user to carefully ensure that scopes are nested correctly with *begin* and *end*. We have solved this with some syntactic sugar, but this could be circumvented unless the constructors are removed from the programmer's vocabulary. This can be achieved by abstraction.

Another criticism is that this solution is perhaps not as general as we would hope. For instance, it is insufficient to solve the related problem of expressing the scope of code with exception handling, which we explore more carefully in the following section.

8. Exceptions

This section presents a second instance of the scoping handler problem in the form of exception handling. Exception handling is a fundamental feature of many programming languages. It allows a block of code to terminate abruptly in a way that throws an exception value from which the overall program resumes computation. We will model this with effect handlers.

The syntax for exceptions is as follows, where an exception value of type *e* is thrown by the syntax *Throw' e*:

```
data Exc e cnt = Throw' e
pattern Throw e ← (project → Just (Throw' e))
```

```
throw :: (Exc e ⊂ sig) ⇒ e → Prog sig a
throw e = inject (Throw′ e)
```

To handle a thrown exception, we use *runExc*:

```
runExc :: Functor sig ⇒
  Prog (Exc e + sig) a → Prog sig (Either e a)
runExc (Return x) = return (Right x)
runExc (Throw e)  = return (Left e)
runExc (Other op) = Op (fmap runExc op)
```

This handler uses the standard approach of encoding exceptions into values of type *Either e a*: computations normally place their results in *Right a*, unless an exception *e* is thrown, in which case this is signalled with the value *Left e*.

In addition to propagating exceptions into the outer program, we are also interested in catching exceptions in code, and handling them with some computation that can recover. This can be modelled by the following handler, where *catch p h* >>= *k* executes *p*, and continues with *k* unless an exception *e* is thrown, in which case *h e* >>= *k* is invoked.

```
catch :: (Exc e ⊂ sig) ⇒ Prog sig a → (e → Prog sig a) → Prog sig a
catch (Return x) h = return x
catch (Throw e)  h = h e
catch (Op op)    h = Op (fmap (λp → catch p h) op)
```

While this handler is perfectly reasonable at first glance, it suffers from the same problem as our initial version of *call*: its does not compose as flexibly as it could.

As a simple example, consider the interaction of exceptions and state. The following code attempts to decrement the state counter three times, and if an exception is thrown it is handled with *return*:

```
tripleDecr :: (State Int ⊂ sig, Exc () ⊂ sig) ⇒ Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return

decr :: (State Int ⊂ sig, Exc () ⊂ sig) ⇒ Prog sig ()
decr = do x ← get
          if x > (0 :: Int) then put (pred x)
                            else throw ()
```

The *decr* decrements a counter held in state. However, if the decrement would result in a negative value, then the state is left unchanged and an exception is thrown.

If we run *tripleDecr* on a state that initially contains 2, then an exception will be raised by the third *decr*. In this scenario, there are two different reasonable final states to expect: a global interpretation would result in a final state of 0, where the first two *decr*s persist; a more local interpretation would lead us to expect all of the effects within the *catch* to be rolled back, so that the final state 1 is the result of the first *decr* only.

Obtaining these different behaviours should in principle be possible by reordering handlers. However, because *catch* is a scoping handler that creates a local scope, we can only express the global interpretation.

```
> (run · runExc · runState 2) tripleDecr
Right (0, ())
```

Exchanging *catch* and *runState* does not make sense, because it would change the scope created by *catch*.

9. Scoped Syntax Revisited

We already noted that the scoped syntax in Section 7 is insufficient to capture the behaviour of exceptions. The issue is that a catch block has two different continuations in addition to the body that is to be executed: one continuation in the case where no exceptions are thrown, and another for the exception handler.

We can solve the problem by extending the idea of using tags to delineate the different blocks of code involved.

```
data Catch e cnt = BCatch′ cnt (e → cnt) | ECatch′ cnt
pattern BCatch p q ← (project → Just (BCatch′ p q))
pattern ECatch p   ← (project → Just (ECatch′ p))
```

Instead of exposing smart constructors *bcatch* and *ecatch*, we instead introduce the following syntactic sugar that ensures the tags are matched appropriately:

```
catch′ :: ∀sig e a . (Catch e ⊂ sig) ⇒
  Prog sig a → (e → Prog sig a) → Prog sig a
catch′ p h = begin (do x ← p; end; return x) h where
  begin p q = inject (BCatch′ p q)
  end = inject (ECatch′ (return ()) :: Catch e (Prog sig ()))
```

Notice that the constructor for *bcatch* does not make use of *return*, which is what we have done in every continuation parameter so far: instead, the syntax *bcatch p q* takes two continuations explicitly, where *p* represents the code that is to be tried, and *q* is the code that handles potential exceptions.

The *runCatch* function is similar to *runCut*: it handles exceptions, and is defined in terms of two subsidiary functions *bcatch* and *ecatch*.

```
runCatch :: (Functor sig) ⇒
  Prog (Catch e + (Exc e + sig)) a → Prog sig (Either e a)
runCatch p = runExc (bcatch p)
```

The function *bcatch* searches for a *BCatch p q*, and when one is encountered, it recursively runs exception handling on *p*. If an exception is raised, then the handling code *q* is used for the continuation.

```
bcatch :: (Functor sig) ⇒
  Prog (Catch e + (Exc e + sig)) a → Prog (Exc e + sig) a
bcatch (Return a)    = return a
bcatch (BCatch p q) =
  do r ← upcast (runExc (ecatch p))
     case r of
       Left e    → bcatch (q e)
       Right p′ → bcatch p′
bcatch (ECatch p)    = error "Mismatched ECatch!"
bcatch (Other op)    = Op (fmap bcatch op)
```

The scope is delimited by an *ECatch*, which are handled by *ecatch*. This implementation mirrors that of *bcatch*.

```
ecatch :: (Functor sig) ⇒
          Prog (Catch e + (Exc e + sig)) a →
          Prog (Exc e + sig) (Prog (Catch e + (Exc e + sig)) a)
ecatch (Return a)    = return (Return a)
ecatch (BCatch p q) =
  do r ← upcast (runExc (ecatch p))
     case r of
       Left e    → ecatch (q e)
       Right p′ → ecatch p′
ecatch (ECatch p)    = return p
ecatch (Other op)    = Op (fmap ecatch op)
```

All that needs to change in our example is the type signature, since we now use *Catch* markers before the exception syntax.

```
tripleDecr :: (State Int ⊂ sig, Exc () ⊂ sig, Catch ⊂ sig) ⇒
  Prog sig ()
```

We are now able to change the behaviour by composing *runCatch* and *runState* in different orders:

$> (run \cdot runCatch \cdot runState\ 2)\ tripleDecr$
$Right\ (1,())$

$> (run \cdot runState\ 2 \cdot runCatch)\ tripleDecr$
$(0, Right\ ())$

Here we see that local state behaviour occurs when *runState* is run first, and global state behaviour when it is last.

10. Higher-Order Syntax

The previous sections used syntax to carefully mark the beginning and end of syntax blocks that should be handled in some self-contained context. A more direct solution is to model scoping constructs with higher-order syntax, where the syntax carries those syntax blocks directly.

For the handler *catch p h*, we introduce the following signature, where the syntax $Catch'\ p\ h\ k$ carries the program contained in p directly as an argument, as well as the exception handler h. The continuation is k, which takes the result of either a successful program p, or from the exception handler h, depending on whether an exception is thrown.

```
data HExc e m a
  = Throw' e
  | ∀x . Catch' (m x) (e → m x) (x → m a)
```

This differs from signatures we have seen previously in two important ways. First, higher-order signatures refine the single type parameter *cnt* into two parts, m and a. By having m and a as two separate type parameters we have tighter control over the type of continuation that is allowed. For instance, in $Catch'\ p\ h\ k$ the scoped computation p has type $m\ x$ while the continuation k turns the result x into a computation of type $m\ a$.

Higher-order signatures are functorial in both type parameters. In the last parameter a they clearly satisfy the ordinary functor laws when m is a functor.

```
instance Functor m ⇒ Functor (HExc e m) where
  fmap f (Throw' e)     = Throw' e
  fmap f (Catch' p h k) = Catch' p h (fmap f · k)
```

Functoriality in the parameter m makes such signatures higher-order functors, which are functors in the category of functors and natural transformations. This is captured by the *HFunctor* class.

```
type f ·→ g = ∀x . f x → g x

class HFunctor h where
  hmap :: (Functor f, Functor g) ⇒ (f ·→ g) → (h f ·→ h g)

instance HFunctor (HExc e) where
  hmap t (Throw' x)     = Throw' x
  hmap t (Catch' p h k) = Catch' (t p) (t · h) (t · k)
```

This allows us to transform the type constructor m with a natural transformation.

Higher-order signatures depart from ordinary ones in an important way: they support more precise control over how they compose, and how handlers must traverse them. In first-order signatures this control is determined entirely by the functor instance, but on deeper inspection the function $fmap :: (cnt \to cnt') \to (sig\ cnt \to sig\ cnt')$ plays two roles: first, to extend the continuation captured by the syntax, and second, to thread handlers through the syntax.

We separate these two roles into two different functions for higher-order syntax, where they will not always coincide. These are captured in the *Syntax* class:

```
class HFunctor sig ⇒ Syntax sig where
  emap :: (m a → m b) → (sig m a → sig m b)
  weave :: (Monad m, Monad n, Functor s) ⇒
           s () → Handler s m n → (sig m a → sig n (s a))
type Handler s m n = ∀x . s (m x) → n (s x)
```

Thus we use *emap* to extend the continuation, and *weave* to determine how handlers should be threaded through the syntax. We describe these functions in more detail below.

Extending the Continuation The *emap* method plays to first role, extending the continuation, and has the following type signature.

$$emap :: Syntax\ sig \Rightarrow (m\ a \to m\ b) \to (sig\ m\ a \to sig\ m\ b)$$

This type is obtained by refining *cnt* to $m\ a$ in the signature of *fmap*. The two are closely related, and *emap* is subject to similar laws:

$$emap\ id = id \tag{1}$$

$$emap\ f \cdot emap\ g = emap\ (f \cdot g) \tag{2}$$

Those versed in category theory will notice that these are the functor laws for the action on arrows, and indeed, we can think of *emap* as a functor from the sub-category obtained through the image of the functor m. This gives the following condition:

$$fmap = emap \cdot fmap \tag{3}$$

The *fmap* on the left hand side is from the functor instance of $sig\ m$, and this should agree with the *fmap* for m when extended by *emap*.

Since higher-order signatures and programs are a generalization of first-order ones, we will redefine all of the infrastructure so that it works in this setting. The type *Prog* now becomes slightly different, since $sig :: (* \to *) \to * \to *$ is of a different kind. The *Op* constructor is also adjusted accordingly.

```
data Prog sig a
  = Return a
  | Op (sig (Prog sig) a)
```

We use *emap* in the definition of the free monad *Prog* over higher-order signatures, since this is where continuations get plugged together.

```
instance Syntax sig ⇒ Monad (Prog sig) where
  return v = Return v

  Return v >>= prog = prog v
  Op op    >>= prog = Op (emap (>>=prog) op)
```

The restricted type of *emap* precisely captures our requirements here, where m is *Prog sig*.

Distributing Handlers Handlers of higher-order syntax have to be formulated in a more modular form than first-order handlers. This is because scoped syntax such as $Catch'\ p\ h\ k$ represents a sequential computation $p >\!\!>\!\!= k$ where p is isolated in a scope. However, this scope is only relevant for the interpretation of $Throw'\ e$, and not for other effects.

The generic threading of another handler *hdl* through $Catch'\ p\ h\ k$ results in $Catch'\ p'\ h'\ k'$. When no exception is thrown in p, then $p' >\!\!>\!\!= k'$ should be equivalent to running the *hdl* on $p >\!\!>\!\!= k$. This means that it should be possible to suspend handlers and resume them from an intermediate point like inbetween p and k.

Because a handler may be stateful, we need to capture its intermediate state when it suspends and make it available when resuming. We capture these requirements in the *Handler s m n* synonym. Here s is a functor that captures the computation state of the handler. A higher-order handler is then a function that transforms a state-annotated computation in one monad m into a computation in another monad n whose value is annotated with the final state.

The type of a handler $hdl :: Handler\ s\ m\ n$ reveals that it is a natural transformation, and by imposing the following conditions it

is a distributive law which composes nicely with monads:

$$hdl \cdot fmap\ return = return \tag{4}$$

$$hdl \cdot fmap\ join = join \cdot fmap\ hdl \cdot hdl \tag{5}$$

These laws arise naturally as the coherence conditions that govern the interaction between *hdl* and monadic computations.

The first law expresses that the handler preserves a pure computation without modifying its state. The second law expresses that it makes no difference whether a composite computation is transformed before or after composition, and makes use of *join*, which is interdefinable with $(\gg\!=)$:

$$join\ mx = mx \gg\!= id \qquad mx \gg\!= f = join\ (fmap\ f\ mx)$$

While the second distributivity law is better expressed in terms of *join*, an operational understanding might be more easily obtained when we rewrite it using these equivalences and name values:

$$hdl\ (fmap\ (\gg\!= k)\ sm) = hdl\ sm \gg\!= hdl \cdot fmap\ k$$

The *weave* method generically threads a handler through a higher-order signature. In addition to the handler, the method also takes the initial state of s, which is represented by $s\ ()$.

```
weave :: (Monad m, Monad n, Functor s) ⇒
  s () → Handler s m n → (sig m a → sig n (s a))
```

Its interpretation is perhaps best understood with an example: here is the *Call* instance of the *Syntax* class.

```
instance Syntax (HExc e) where
  emap f (Throw' e)     =  Throw' e
  emap f (Catch' p h k) =  Catch' p h (f · k)
  weave f hdl (Throw' x)     = Throw' x
  weave f hdl (Catch' p h k) =
    Catch' (hdl (fmap (const p) f))
           (λe → hdl (fmap (const (h e)) f))
           (hdl · fmap k)
```

The definition of *emap* is straightforward: the *Throw'* case is trivial since it does not carry a continuation, and in the *Catch' p h k* case we apply f to the continuation k.

The definition of *weave* for *Throw'* is equally trivial, but *Catch'* is more involved. We derive its implementation as follows. Assuming that *weave s hdl* (*Catch' p h k*) yields *Catch' p' h' k'*. In the absence of exceptions in p we require that

$$p' \gg\!= k' = hdl\ (fmap\ (\gg\!= k)\ (fmap\ (const\ p)\ s))$$

Based on the second handler law, the latter is equivalent to

$$hdl\ (fmap\ (const\ p)\ s) \gg\!= hdl \cdot fmap\ k$$

which gives us the following solutions for p' and k' in terms of p and k respectively:

$$p' = hdl\ (fmap\ (const\ p)\ s) \qquad k' = hdl \cdot fmap\ s$$

Similarly, if $p = throw\ e$, we require that

$$h'\ e \gg\!= k' = hdl\ (fmap\ (\gg\!= k)\ (fmap\ (const\ (h'\ e)\ s))$$

which gives us, by way of the second handler law, a solution for h':

$$h' = \lambda e \to hdl\ (fmap\ (const\ (h\ e))\ s)$$

10.1 Infrastructure

The infrastructure that supports higher-order syntax is for the most part an adapted version of what was presented in earlier sections, and its definition is fairly routine. We need only change two things: the *Functor* constraints now become *Syntax* instead, and continuation parameters *cnt* now become *m a*.

As far as the datatypes à la carte machinery is concerned, the only changes are the signatures of the class:

```
class (Syntax sub, Syntax sup) ⇒ sub ⊂ sup where
  inj :: sub m a → sup m a
  prj :: sup m a → Maybe (sub m a)
```

All of the instances need only have their *Functor* constraints turned into *Syntax*, and the bodies remain identical. We can provide patterns for higher-order syntax in just the same way as before:

```
pattern Throw e ← (project → Just (Throw' e))
throw :: (HExc e ⊂ sig) ⇒ e → Prog sig a
throw e = inject (Throw' e)

pattern Catch p h k ← (project → Just (Catch' p h k))
catch :: (HExc e ⊂ sig) ⇒
  Prog sig a → (e → Prog sig a) → Prog sig a
catch p h = inject (Catch' p h return)
```

The composition of higher-order syntax needs a little more attention, since *weave* must correctly thread handlers through subparts:

```
data (sig1 + sig2) (m :: * → *) a = Inl (sig1 m a) | Inr (sig2 m a)
instance (HFunctor sig1, HFunctor sig2) ⇒
         HFunctor (sig1 + sig2) where
  hmap t (Inl op) = Inl (hmap t op)
  hmap t (Inr op) = Inr (hmap t op)
instance (Syntax sig1, Syntax sig2) ⇒
         Syntax (sig1 + sig2) where
  emap f (Inl op) =  Inl (emap f op)
  emap f (Inr op) =  Inr (emap f op)
  weave s hdl (Inl op) = Inl (weave s hdl op)
  weave s hdl (Inr op) = Inr (weave s hdl op)
```

In addition, we can easily lift our existing first-order signatures to higher-order signatures, by performing the refinement of *cnt* to *m a*. In this lifting, the first-order *fmap* provides the definitions needed for all the higher-order methods.

```
newtype (Lift sig) (m :: * → *) a = Lift (sig (m a))
instance Functor sig ⇒ HFunctor (Lift sig) where
  hmap t (Lift op) = Lift (fmap t op)
instance Functor sig ⇒ Syntax (Lift sig) where
  emap f (Lift op) = Lift (fmap f op)
  weave s hdl (Lift op) =
    Lift (fmap (λp → hdl (fmap (const p) s)) op)
```

Providing signatures and syntax for first-order syntax is now simple boilerplate. For instance, running higher-order programs with an empty signature is performed in the same way as first-order programs, except the signature is now lifted:

```
type HVoid = Lift Void
run :: Prog HVoid a → a
run (Return x) = x
```

The lifted signatures and syntax for *State* are outlined in Figure 1.

10.2 Higher-Order Handlers

With the infrastructure for higher-order syntax in place, we are now in a position to define handlers.

State The first higher-order handler we look at is *runState*, which is almost identical to its first-order counterpart. Two things have changed: the type signature is modified to use *HState*, and the clause for *Other op* involves *weave*.

```
type HState s = Lift (State s)
pattern Get k ← (project → Just (Lift (Get' k)))
get :: (HState s ⊂ sig) ⇒ Prog sig s
get = inject (Lift (Get' return))
pattern Put s k ← (project → Just (Lift (Put' s k)))
put :: (HState s ⊂ sig) ⇒ s → Prog sig ()
put s = inject (Lift (Put' s (return ())))
```

Figure 1. Lifted *State* signatures and syntax.

```
runState :: Syntax sig ⇒
  s → Prog (HState s + sig) a → Prog sig (s, a)
runState s (Return a) = return (s, a)
runState s (Get k)    = runState s (k s)
runState s (Put s' k) = runState s' k
runState s (Other op) = Op (weave (s, ()) (uncurry runState) op)
```

Here we see that the *fmap* (*runState s*) *op* in the original definition has been replaced with *weave* (*s*, ()) (*uncurry runState*) *op*.

Exceptions Exceptions have a more complex handler. This is to be expected since we must take care of the intermediate state that is to be woven through higher-order syntax. The clauses for *Return* and *Throw* remain identical, and only *Catch* and *Other* are of interest.

```
runExc :: Syntax sig ⇒
  Prog (HExc e + sig) a → Prog sig (Either e a)
runExc (Return x)     = return (Right x)
runExc (Throw x)      = return (Left x)
runExc (Catch p h k) =
  do r ← runExc p
     case r of
       Left e  → do r ← runExc (h e)
                    case r of
                      Left e  → return (Left e)
                      Right a → runExc (k a)
       Right a → runExc (k a)
runExc (Other op) = Op (weave (Right ()) hdl op) where
  hdl :: Syntax sig ⇒
    Handler (Either e) (Prog (HExc e + sig)) (Prog sig)
  hdl = either (return · Left) runExc
```

The case for *Catch p h k* attempts to execute *p*. If an exception *e* is thrown, then this is used to execute the exception handler *h*. If that fails again, then this is an uncaught exception that propagates its way out. In all other cases, some valid computation *a* is returned, which is fed into *runExc* (*k a*), since *k a* may itself throw exceptions.

The *Other* case starts computations with a successful computation *Right* (). The distributive handler simply propagates errors when they are encountered with *return* · *Left*, and otherwise recursively applies *runExc*.

Recreating the example of the previous section requires very little work indeed. For instance, here is the definition of *tripleDecr*:

```
tripleDecr :: (HState Int ⊂ sig, HExc () ⊂ sig) ⇒ Prog sig ()
tripleDecr = decr ≫ (catch (decr ≫ decr) return)
```

The only change is in the *types* of the functions, since we have moved to a higher-order setting!

11. Multi-Threading

Cooperative multi-threading allows a thread to suspend its computation with a *yield* operation: this relinquishes control to the scheduler, which may decide to run a different thread. We assume the existence of an initial master thread, and new child threads are created by calling *fork*.

The examples of scoped effects we have seen so far—pruning non-deterministic computations, and exception handling—have been solved both by clever use of first-order syntax tagging, and also by higher-order syntax. We might, therefore, expect both techniques to be equally expressive. In this section we put that expectation to rest and show that cooperative multi-threading can only be solved using the higher-order approach.

11.1 Signature

The multi-threading effect *Thread* concerns two operations: *fork d* spawns a new thread *d*, and *yield* relinquishes control. While *yield* is a plain algebraic operation, *fork* is clearly a scoping construct that delimits the new thread.

It would be wrong to capture the signature as the following first-order syntax.

```
data Thread cnt    -- BOGUS!
  = Yield' cnt
  | Fork' cnt cnt
```

Here *Fork' p q* would represent a computation that spawns a new thread *p*, while the master thread continues with *q*. The problem is that, in the first-order framework, we have that

$$Op\ (Fork'\ p\ q) \gg\!= k = Op\ (Fork'\ (p \gg\!= k)\ (q \gg\!= k))$$

This is clearly not the desired semantics, since forking would result in in the continuation *k* being run in both the parent thread and its child. Instead we want the following, where the continuation is only applied to the remainder of the parent *q*.

$$Op\ (Fork'\ p\ q) \gg\!= k = Op\ (Fork'\ p\ (q \gg\!= k))$$

In other words, we should distinguish between the subcomputation for the child thread and the one for the continuation of the parent thread. First-order syntax does not have this capability, only higher-order syntax does:

```
data Thread m a
  = Yield' (m a)
  | ∀x . Fork' (m x) (m a)
pattern Yield p ← (project → Just (Yield' p))
yield :: (Thread ⊂ sig) ⇒ Prog sig ()
yield = inject (Yield' (return ()))
pattern Fork p q ← (project → Just (Fork' p q))
fork :: (Thread ⊂ sig) ⇒ Prog sig a → Prog sig ()
fork d = inject (Fork' d (return ()))
```

The *Syntax* instance shown below distinguishes between the two subcomputations, where *emap* marks the continuation and ensures the desired semantics for (≫=).

```
instance Syntax Thread where
  emap f (Yield' p)   =  Yield' (f p)
  emap f (Fork' d p)  =  Fork' d (f p)
  weave s hdl (Yield' p)   = Yield' (hdl (fmap (const p) s))
  weave s hdl (Fork' d p)  = Fork' (hdl (fmap (const d) s))
                                   (hdl (fmap (const p) s))
```

Note that the result type of the new thread is existentially quantified:

$$Fork' :: \forall x\ .\ m\ x \to m\ a \to Thread\ m\ a$$

This is in line with the notion that in *Fork' d p* ≫= *k* the continuation *k* does not interact with the child's result: there is no direct communication between the child and the master thread, and no need to constrain its return type.

We call a thread with an existentially quantified result type a *daemon*, in contrast with the master thread of the program.

```
data Daemon sig = ∀x . Daemon (Prog (Thread + sig) x)
```

11.2 Handler

We adopt the semantics that a thread suspends at every *fork* and *yield* in favour of running another thread. Hence, a thread can be in one of three different states, two of which are named after the corresponding syntax:

```
data SThread sig r
  = SYield  (Prog (Thread + sig) r)
  | SFork   (Daemon sig) (Prog (Thread + sig) r)
  | SActive r

instance Syntax sig ⇒ Functor (SThread sig) where
  fmap f (SActive x) = SActive (f x)
  fmap f (SYield p)  = SYield (liftM f p)
  fmap f (SFork d p) = SFork d (liftM f p)
```

The default state is *SActive*, which denotes an ongoing thread. The *SFork d p* state denotes a thread that has suspended at a *fork* that spawns a daemon *d* and should continue with *p*. Similarly, *SYield p* denotes a thread that has suspended at a *yield* and should continue with *p*. The similarity between *Thread* and *SThread* is no coincidence: *SThread* is very nearly *Prog* (*Thread* + *sig*) *a*, except that the latter has an additional constructor *Other* for syntax.

The *runThread* handler runs a thread and returns its resulting state in the obvious way.

```
runThread :: Syntax sig ⇒
  Prog (Thread + sig) x → Prog sig (SThread sig x)
runThread (Return x) = return (SActive x)
runThread (Yield q)  = return (SYield q)
runThread (Fork d q) = return (SFork (Daemon d) q)
runThread (Other op) = Op (weave (SActive ()) thread op)
```

The helper function *thread* decides how to continue from an intermediate state. It calls *runThread* for an active thread, and extends the continuation for both kinds of suspended threads.

```
thread :: Syntax sig ⇒
  Handler (SThread sig) (Prog (Thread + sig)) (Prog sig)
thread (SActive p) = runThread p
thread (SYield p)  = return (SYield (join p))
thread (SFork d p) = return (SFork d (join p))
```

Finally, the top-level *schedule* handler runs the master thread and daemons in round-robin fashion. It switches from one thread to another at every *fork* and *yield*.

```
schedule :: Syntax sig ⇒ Prog (Thread + sig) a → Prog sig a
schedule p = master p [] where
  master p ds =
    do r ← runThread p
       case r of
         SActive x  → return x
         SYield p   → daemons ds [] p
         SFork d p → daemons (d : ds) [] p
  daemons [] ds' p = master p (reverse ds')
  daemons (Daemon q : ds) ds' p =
    do r ← runThread q
       case r of
         SActive _  → daemons ds (ds' ++ [Daemon q]) p
         SYield q'  → daemons ds (ds' ++ [Daemon q']) p
         SFork d' q' → daemons (d' : ds) (ds' ++ [Daemon q']) p
```

The *schedule* function adopts the termination condition of the Go language:[1] the whole program ends when the master thread ends; any unfinished daemons are discarded.

11.3 Threading in Action

In order to show an example of multi-threading behaviour, we will introduce syntax for communicating to the outside world with IO. The syntax *Out′ x* expresses that *x* should be written out.

```
data Out cnt = Out' String cnt
type HOut = Lift Out
pattern Out s p ← (project → Just (Lift (Out' s p)))
out :: (HOut ⊂ sig) ⇒ String → Prog sig ()
out s = inject (Lift (Out' s (return ())))
```

The *io* handler turns the syntax into semantics by invoking the appropriate OS system call, which we model by *putStrLn*.

```
io :: Prog HOut a → IO a
io (Return x) = return x
io (Out s p)  = do putStrLn s; io p
```

With this in place, the following program combines the multi-threading effect with state and output.

```
prog :: (Thread ⊂ sig, HState Int ⊂ sig, HOut ⊂ sig) ⇒ Prog sig ()
prog = do logIncr "master"
          fork (logIncr "daemon" ≫ logIncr "daemon")
          logIncr "master"
```

The master thread increments the state twice, but spawns a new thread in between that increments the state twice too.

```
log :: (HState Int ⊂ sig, HOut ⊂ sig) ⇒ String → Prog sig ()
log x = do (n :: Int) ← get; out (x ++ ": " ++ show n)
logIncr :: (HState Int ⊂ sig, HOut ⊂ sig) ⇒ String → Prog sig ()
logIncr x = log x ≫ incr
```

The *logIncr* function outputs the state before incrementing it.

With our for multi-threading we can obtain two different semantics for *prog*: one where the state is shared among all threads, and one where it is local to each thread. By running *runThread* before *runState* we share the state between the master and daemon:

```
> (io · runState (0 :: Int) · schedule) prog
master : 0
daemon : 1
daemon : 2
master : 3
(4, ())
```

The other way around, *fork* creates a local copy of the state for the daemon and updates are not shared:

```
> (io · schedule · runState (0 :: Int)) prog
master : 0
daemon : 1
daemon : 2
master : 1
(2, ())
```

We have successfully shown how higher-order syntax allows effects to be scoped, which has resulted in interesting compositional semantics through the simple reordering of handlers.

[1] http://golang.org/

12. Related Work

12.1 Effect Handlers

Plotkin and Power were the first to explore effect operations [11], and gave an algebraic account of effects [12] and their combination [4]. Subsequently, Plotkin and Pretnar [13] have added the concept of handlers to deal with exceptions. This has led to many implementations.

Languages Based on this idea, two entirely new programming languages have been created from the ground up around algebraic effect handlers.

- Eff [1] is an ML-variant that does not track effect signatures in its static type system. Hence, its type system does not rule out higher-order syntax. For instance, Bauer and Pretnar show how to implement the multi-threading example in Eff, but can only get the global state interpretation.
- Frank [10] does track effect signatures in its static type system and does not allow higher-order syntax.

Libraries More recently, three proposals show how to implement algebraic effect handlers on top of existing functional programming languages:

- Brady [2] provides an effect handlers approach in the Idris language. The approach supports only one special built-in syntactic scoping construct, *catch*, but does not support additional higher-order syntax.
- Kammar et al. [7] present several different implementations in Haskell, OCaml, SML and Racket. These are based on different implementation techniques: the free monad and a continuation-based approach are considered in Haskell, and delimited continuations for the other languages. Scoping syntax is not covered. How delimited continuations can be used to implement higher-order syntax requires further investigation.
- Kiselyov et al. [8] provide a Haskell implementation in terms of the free monad, in combination with two optimizations: 1) the codensity transformer improves the performance of $(\gg\!=)$, and 2) their *Dynamic*-based open unions have a better time complexity than nested co-products. We believe that both of these optimizations can be applied to our first-order and higher-order scoped syntax. They do not cover scoping syntax.

  As far as we are aware Kiselyov et al. are the first to provide a handler for *Cut*,[2] inspired by Hinze's monad transformer [3]. However, they do not discuss the scoping problem.

12.2 Monad Transformers

The issue of scoping operations already arises in the more general setting of monad transformers [9] where different effects can be composed in different ways to obtain different semantics. The *lift* operation is used to combine operations from different transformers in the same program. While lifting algebraic operations is typically easy, lifting scoping operations is typically not. This problem is addressed by Jaskelioff and Moggi [6] for a class of *functorial* operations and available in the Monatron library [5].

13. Conclusion

We have shown that using effect handlers for scoping fixes the interaction between effects. Our main message is that, in order to regain control over the semantics of interaction, syntax should determine scope.

We provide two approaches for scoping syntax, scope markers and higher-order syntax, each with their merits. Scope markers play nicely with all existing effect handler frameworks and can piggy-back on existing optimizations and convenience infrastructure (e.g., Template Haskell macros). In contrast, it is an open question how higher-order syntax can be implemented on top of delimited continuations, which is the basis for effect handlers in strict languages. The advantages of higher-order syntax are that it is strictly more expressive and that is a more natural way to denote scoping; the markers run the risk of being unbalanced.

For future work we believe that it should be possible to generically lift first-order handlers to the higher-order setting provided that they are expressed in terms of algebras.

[2] Our variant in Section 5 simplifies theirs from n-way to 2-way choice and omits the codensity optimization.

Acknowledgments

This work has been funded by EPSRC grant number EP/J010995/1 on Unifying Theories of Generic Programming, and by the Flemish Fund for Scientific Research (FWO).

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers, 2012.

[2] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 133–144. ACM, 2013.

[3] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 186–197, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. . URL `http://doi.acm.org/10.1145/351240.351258`.

[4] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

[5] M. Jaskelioff. Monatron: An extensible monad transformer library. In S.-B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2008.

[6] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, Dec. 2010.

[7] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 145–158. ACM, 2013.

[8] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013.

[9] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, 1995.

[10] C. McBride. The Frank manual, May 2012. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/TFM.pdf`.

[11] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[12] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[13] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

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