MSc dissertation progress presentation

Handlers for Algebraic Effects in Links

Daniel Hillerström
s1467124@sms.ed.ac.uk

School of Informatics
University of Edinburgh

July 16, 2015

Project description

Motivation

- Monads are great abstractions for programming with *explicit* effects. But in general monads do not compose.
- Monad Transformers (MT) provide compositionality for monads. But MTs impose an implicit ordering on effects.

Problem statement

How may we obtain modular, composable and unordered effects?

Project description

Motivation

- Monads are great abstractions for programming with *explicit* effects. But in general monads do not compose.
- Monad Transformers (MT) provide compositionality for monads. But MTs impose an implicit ordering on effects.

Problem statement

How may we obtain modular, composable and unordered effects?

Answer: Handlers for algebraic effects using row polymorphism!

Project description

Motivation

- Monads are great abstractions for programming with *explicit* effects. But in general monads do not compose.
- Monad Transformers (MT) provide compositionality for monads. But MTs impose an implicit ordering on effects.

Problem statement

How may we obtain modular, composable and unordered effects?

Answer: Handlers for algebraic effects using row polymorphism!

Project aim

Provide an implementation of first-class effect handlers in the web-oriented functional programming language Links [1].

Effects and handlers

Definition (Algebraic effect)

An effect is a collection of abstract operations, e.g. $\{Op_i : a_i \rightarrow b_i\}$

Definition (Effect handler)

An effect handler interprets operations.

Computations as trees

Intuitively, we can think of operations as *nodes* in computation trees, and handlers as computation tree transformers [2].

Design

Handlers

Essentially, handlers embody a collection of case-statements, e.g.

```
handle(x) {
  case Op1(p,k)  -> ...
  case OpN(p,k)  -> ...
  case Return(x) -> ...
}
```

This style is adapted from Plotkin and Pretnar [3, 4].

Operation discharge

Operations are discharged using the "do" primitive, e.g. `do Op(arg)`

Make a choice!

Live demo or backup slides?

How does row polymorphism fit into this?

- Handlers get typed as

$$\texttt{fun} : (() \xrightarrow{\{Op_1 : a_1 \to b_1, \ldots, Op_n : a_n \to b_n\}} c) \to c$$

that is, they are closed.

How does row polymorphism fit into this?

- Handlers get typed as

$$\texttt{fun} : (() \xrightarrow{\{Op_1 : a_1 \to b_1, \ldots, Op_n : a_n \to b_n\}} c) \to c$$

that is, they are closed.

- We want *open* handlers too, e.g.

$$\texttt{fun} : (() \xrightarrow{\{Op_1 : a_1 \to b_1, \ldots, Op_n : a_n \to b_n \mid \rho\}} c) \to c$$

Consequently, we obtain a high-degree of modularity.

Wrap up

At this stage

- Front end stuff.
- Added user-defined effects.
- Type checking handlers and operations.
- Almost done implementing closed handlers.

Todo in the near future

- Add open handlers.
- Programming with handlers and effects.
- Explore generalisations.
- Refactor.
- Write-up.

References

Philip Wadler, Sam Lindley, Garrett Morris, et al.
Links: Linking theory to practice for the web.
http://groups.inf.ed.ac.uk/links/, 2005.

Sam Lindley.
Algebraic effects and effect handlers for idioms and arrows.
In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 47–58, 2014.

Gordon D. Plotkin and Matija Pretnar.
Handling algebraic effects.
Logical Methods in Computer Science, 9(4), 2013.

Ohad Kammar, Sam Lindley, and Nicolas Oury.
Handlers in action.
In *ICFP'13*, pages 145–158, 2013.

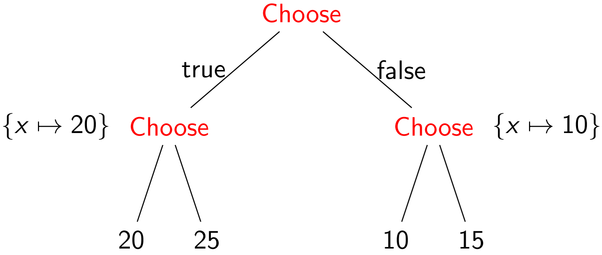
Computations as trees (I)

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) -> k(true)
   case Return(x)   -> x
  }
}
```



Computations as trees (I)

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) -> k(true)
   case Return(x)   -> x
  }
}
```

$\Rightarrow 20$

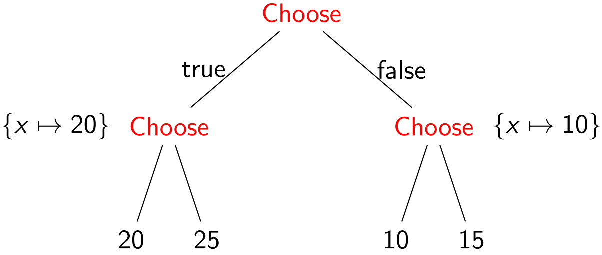
Computations as trees (II)

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) -> k(false
    )
   case Return(x)   -> x
  }
}
```



Computations as trees (II)

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) -> k(false
    )
   case Return(x)   -> x
  }
}
```

$\Rightarrow 15$

Computations as trees (III)

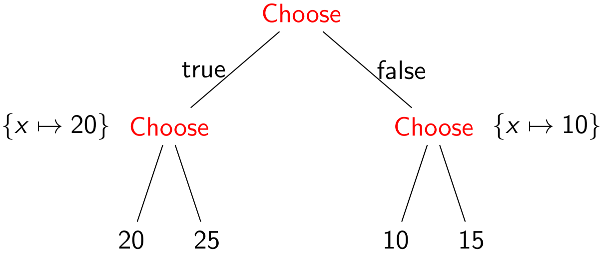
Handler

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) ->
     k(true) ++ k(false)
   case Return(x)    -> [x]
  }
}
```



Computations as trees (III)

Handler

Computation

```
fun choiceC() {
 var x =
  if (do Choose()) { 20 }
  else { 10 }
 var y =
   if (do Choose()) { 0 }
   else { 5 }
 x + y
}
```

Handler

```
fun handler(x) {
  handle(x) {
   case Choose(_,k) ->
     k(true) ++ k(false)
   case Return(x)   -> [x]
  }
}
```

$$\Rightarrow [20, 25, 10, 15]$$

Go back

Go back