This work is supported by

- The University of Edinburgh
- School of Informatics
- EPSRC Centre for Doctoral Training in Pervasive Parallelism
- EPSRC Engineering and Physical Sciences Research Council
- University of Cambridge
- OCaml Labs
Compiling Links Effect Handlers to the OCaml Backend
ML Workshop '16

Daniel Hillerström ¹  Sam Lindley ¹  KC Sivaramakrishnan ²

¹The University of Edinburgh, UK
²University of Cambridge, UK

September 22, 2016
Meet Links (Cooper et al., 2006)

- a ML-like strict functional programming language,
- a single-source language for multi-tier web-programming,
- with a syntax reminiscent of JavaScript, e.g. `fun foo(x,y) { ... }`,
- and a strong type system including linear types,
- with effect typing based on row polymorphism,
- and it provides effect handlers for controlling effects (Hillerström, 2015).

Links has three backends, each written in OCaml:

- a JavaScript compiler for the client,
- an interpreter for the server,
- and an SQL generator for the database,
- and with this work a compiler for the server.

An algebraic effect is a collection of *abstract operations*, e.g.

\[ \text{Nondet} = \{ \text{Choose : Bool} \} \]

Using abstract operations we can define effectful computations *abstractly*, e.g.

``` Ocaml
fun toss() { if (do Choose) Heads else Tails }
```
Algebraic Effects and Abstract Computations

An algebraic effect is a collection of abstract operations, e.g.

\[ \text{Nondet} = \{ \text{Choose} : \text{Bool} \} \]

Using abstract operations we can define effectful computations abstractly, e.g.

```ocaml
sig toss : () {Choose:Bool|e}-> Toss
fun toss() { if (do Choose) Heads else Tails }
```
An algebraic effect is a collection of *abstract operations*, e.g.

\[
\text{Nondet} = \{ \text{Choose} : \text{Bool} \}
\]

Using abstract operations we can define effectful computations *abstractly*, e.g.

```ocaml
sig toss : () \{Choose:Bool\|e\}-> Toss
fun toss() { if (do Choose) Heads else Tails }
```

Evaluation of an abstract computation...

```ocaml
links> toss();
*** Error: Unhandled operation: Choose
```

...but, what is the semantics of \text{Choose}?
A handler instantiates abstract operations with concrete implementations, e.g.

```ocaml
handler randomResult {
  case Return(x) -> x
  case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.
A handler instantiates abstract operations with concrete implementations, e.g.

```
sig randomResult : (() {Choose:Bool|e}-> a) ->  
                     () {Choose{p} |e}-> a
handler randomResult {
    case Return(x) -> x
    case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.
A handler instantiates abstract operations with concrete implementations, e.g.

```
sig randomResult : (() {Choose:Bool|e}-> a) ->
                         (() {Choose{p} |e}-> a
```

```
handler randomResult {
    case Return(x) -> x
    case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Interpretation of `toss` with this handler:

```
links > randomResult(toss)();
Tails : Toss
```
A handler instantiates abstract operations with concrete implementations, e.g.

```
sig allChoices : (() {Choose:Bool|e}-> a) ->
    () {Choose{p} |e}-> [a]
handler allChoices {
    case Return(x) -> [x]
    case Choose(resume) -> resume(true) ++ resume(false)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Interpretation of `toss` with this handler:

```
links > allChoices(toss)();
[Heads, Tails] : [Toss]
```
Consider the following abstract handler:

```
sig flip : (() {Choose:Bool | e} -> a) ->
  () {Choose:Bool | e} -> a)
handler flip {
  case Return(x) -> x
  case Choose(resume) -> resume(not(do Choose))
}
```

We may use `allChoices` to interpret `flip(toss)`:

```
links > allChoices(flip(toss))();
[Tails, Heads] : [Toss]
```
Classification of Handlers

Handlers can be classified according to their continuation consumption.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example(s)</th>
<th>Cont. consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception handler</td>
<td>maybeResult</td>
<td>0</td>
</tr>
<tr>
<td>Linear handler</td>
<td>randomResult, flip</td>
<td>1</td>
</tr>
<tr>
<td>Multi-shot handler</td>
<td>allChoices</td>
<td>&gt; 1</td>
</tr>
</tbody>
</table>
Handlers are not only for coin tossing. In particular, we have a reconstruction of the concurrency model of Links using handlers (Hillerström, 2016).

Thus we are interested in making this abstraction efficient and safe while retaining modularity.
Compiler Backend

OCaml frontend

<table>
<thead>
<tr>
<th>OCaml backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte code</th>
</tr>
</thead>
</table>

| Custom backends |

| Links |

| Clambda |

| Native backends |
Multicore OCaml (Dolan et al., 2015) provides

- effect handlers as an abstraction for concurrency,
- an efficient, native implementation of linear effect handlers,
- an explicit copying construct for on demand multi-shot handlers.

Consider the following example in Links and OCaml:

```ocaml
links> allChoices(flip(toss))();
[Tails, Heads] : [Toss]
```

```ocaml
ocaml# allChoices (flip toss) ();
```
Multicore OCaml (Dolan et al., 2015) provides

- effect handlers as an abstraction for concurrency,
- an efficient, native implementation of linear effect handlers,
- an explicit copying construct for on demand multi-shot handlers.

Consider the following example in Links and OCaml:

```
links> allChoices(flip(toss))()
[Tails, Heads] : [Toss]
```

```
ocaml# allChoices (flip toss) ();
Exception: Invalid_argument "continuation already taken".
```
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of allChoices(flip(toss)):
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:

```
  do Choose
  do
    Choose
    resume(false)
    sp
  end
  resume(true)
  resume(false)
```

Conservative solution: implement every handler as a multi-shot handler.
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:

- `H_1`: allChoices
- `H_2`: flip

Reference vs. call chain:
- `reference`: solid arrow
- `call chain`: dotted arrow

Conservative solution: implement every handler as a multi-shot handler.
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of \texttt{allChoices(flip(toss))}:

\begin{itemize}
    \item \texttt{H1} \texttt{handler} \texttt{allChoices}
    \item \texttt{H2} \texttt{resume(true)} \texttt{sp} \texttt{flip}
    \item \texttt{toss} \texttt{computation}
\end{itemize}

Conservative solution: implement every handler as a multi-shot handler.
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of allChoices(flip(toss)):
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of $\text{allChoices}(\text{flip}(\text{toss}))$:

Conservative solution: implement every handler as a multi-shot handler.

D. Hillerström (University of Edinburgh)
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:

Conservative solution: implement every handler as a multi-shot handler.
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:

```
do
  Choose resume(false)
  Choose resume(true)
  resume(false)
sp sp sp
```

Conservative solution: implement every handler as a multi-shot handler.

D. Hillerström (University of Edinburgh)

Compiling Links Effect Handlers to the OCaml Backend

22-09-2016 11/20
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of \( \text{allChoices}(\text{flip}(\text{toss})) \):

Conservative solution: implement every handler as a multi-shot handler.
What is the Penalty?

Dynamic process generation (Sieve)

Number of processes

Median execution time [ms]

Concurrency impl.
- Compiler/handlers
- Interpreter/built-in

D. Hillerström (University of Edinburgh)
**Idea:** let’s use the linear type system to track the linearity of handlers.
Scoping of Handler Promotion

Initial idea: use the effect system to propagate linearity information.

Ideally, we want this:

Legend

<table>
<thead>
<tr>
<th>$h_n^*$</th>
<th>Multi-shot handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_n$</td>
<td>Linear handler</td>
</tr>
</tbody>
</table>
Initial idea: use the effect system to propagate linearity information.

But, this is what really happens:

```
[horizontal diagram]

\[ h_\perp \quad \ldots \quad h_2^* \quad h_1 \]

\[ \Rightarrow \]

\[ h_\perp^* \quad \ldots^* \quad h_2^* \quad h_1^* \]
```

Need some way to capture the structure of the handler stack at the type-level.

<table>
<thead>
<tr>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_n^* )</td>
</tr>
<tr>
<td>( h_n )</td>
</tr>
</tbody>
</table>
The linear type system is not expressive enough to capture tombstones.

Use Case: Channel Selection

The linear type system is not expressive enough to capture tombstones.
Use Case: Channel Selection

The linear type system is not expressive enough to capture tombstones.
The linear type system is not expressive enough to capture tombstones.

\[ \text{resume} \]

\[ \text{resume'} \]

This is not the desired semantics!
Compilation options

- Continuation monad (Kammar et al., 2013)
- Free monad (Kiselyov et al., 2013; Bauer and Pretnar, 2015)
- Direct-style like Multicore OCaml (Dolan et al., 2015)
- Selective CPS translation (Leijen, 2016)
- Shift/reset control operators (Saleh and Schrijvers, 2016)
Algebraic effects and handlers provide a modular abstraction for effectful programming.

OCaml backend gives you a native code generator (almost) for free.

Regard Links as an experimental frontend to OCaml with effect typing and linear types.

Type-and-effect directed optimisations of handlers is promising.

To capture common use cases we need a more expressive linear type system.
Daniel Hillerström.
Handlers for algebraic effects in Links.
Master’s thesis, School of Informatics, the University of Edinburgh, Scotland, August 2015.

Daniel Hillerström.
Compilation of effect handlers and their applications in concurrency.
Master’s thesis, School of Informatics, the University of Edinburgh, Scotland, August 2016.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop.
Links: Web programming without tiers.
Amr Hany Saleh and Tom Schrijvers.
Efficient algebraic effect handlers for Prolog.
Submitted to TPLP, 2016.

Oleg Kiselyov, Amr Sabry, and Cameron Swords.
Extensible effects: an alternative to monad transformers.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy.
Effective concurrency through algebraic effects.
OCaml Workshop, 2015.

Andrej Bauer and Matija Pretnar.
Programming with algebraic effects and handlers.
Ohad Kammar, Sam Lindley, and Nicolas Oury.  
Handlers in action.  

Daan Leijen.  
Type directed compilation of row-typed algebraic effects.  