Extended Abstract

Compiling Links Effect Handlers to the OCaml Backend

Daniel Hillerström
The University of Edinburgh

Sam Lindley
The University of Edinburgh

KC Sivaramakrishnan
University of Cambridge

Abstract

Algebraic effects and handlers provide a modular abstraction for modelling and controlling computational effects. We present a compiler for the experimental language Links with effect handlers. Our compiler interfaces with the Multicore OCaml backend to take advantage of OCaml's implementation of efficient handlers.

1. Motivation

Algebraic effects and handlers [7] afford a compelling and modular alternative to monad transformers for controlling computational effects. In previous work, we extended the functional web programming language Links with algebraic effects and handlers [3, 4].

Links is a strict ML-like functional language for the web [1]. It has three backends: i) a JavaScript compiler for the client, ii) an interpreter for the server, iii) and an SQL generator for the database. Currently effect handlers are only supported on the server; in future we intend to extend the JavaScript compiler to support them too.

In order to improve performance, and to study efficient compilation of effect handlers in general, we have implemented a Links compiler that targets the Multicore OCaml backend [2]. By taking advantage of the OCaml backend we obtain both native code compilation and access to the OCaml toolchain for free.

Prior work focuses mainly on the design and expressiveness of handlers rather than performance. Nevertheless, several papers do address performance. Kammar et al. [5] take advantage of Haskell's aggressive fusion optimisations for an efficient Haskell library for handlers, as explained in detail by Wu and Schrijvers [9]. Kiselyov and Ishii [6] also optimise a different Haskell library for handlers, taking advantage of prior work on optimising monadic reflection [8]. Our work differs from these systems in that we compile effect handlers directly, rather than via library.

2. Affine and Multi-shot Effect Handlers

This section provides a short primer to effect handlers in Links. An algebraic effect is given by a signature of *abstract operations*. For example *nondeterminism* is an algebraic effect that is given by a nondeterministic choice operation called `Choose`. In Links, we may use this operation to implement a coin toss:

```
sig toss : Comp({Choose:Bool |e}, Toss)
fun toss() { if (do Choose) Heads else Tails }
```

This declares an *abstract computation* `toss`, which invokes an operation `Choose` using the `do` primitive. The `sig` keyword begins a signature, which reads: `toss` is a computation with effect signature `{Choose:Bool |e}` and return value `Toss`, whose constructors are `Heads` and `Tails`. Links employs row typing to support extensible effect signatures, thus `e` is an effect variable, which can be instantiated with additional operations.

Introduction of another operation causes the effect signature to grow accordingly. For example, if we introduce an exception operation `Fail : Zero`, then we can model a drunk coin toss:

```
sig drunkToss : Comp({Choose:Bool,Fail:Zero |e}, Toss)
fun drunkToss() { if (do Choose) toss()
                  else switch (do Fail) { } }
```

Here `Zero` is the empty type, and thus the `switch` pattern matching construct has no clauses.

An effect handler instantiates a subset of the operations of an abstract computation. For example, the following handler interprets `Choose` randomly:

```
sig randomResult : (Comp({Choose:Bool |e}, a)) ->
                    Comp({Choose{_}   |e}, a)
handler randomResult {
  case Return(x) -> x
  case Choose(k) -> k(random() > 0.5)
}
```

The signature conveys that the handler interprets the operation `Choose` and leaves any other operations uninterpreted. The notation `Choose{_}` denotes that the operation is polymorphic in its presence. The handler comprises two clauses: i) the `Return`-clause specifies how to handle the return value of the computation. ii) the `Choose`-clause specifies how to handle a `Choose` operation. The parameter `k` is the (delimited) continuation of the operation `Choose` in the computation. We say that `randomResult` is a *linear handler*, because it invokes every continuation exactly once.

Alternatively, we may define a handler for `Choose` that invokes its continuation twice to enumerate every possible outcome:

```
sig allResults : (Comp({Choose:Bool |e},  a)) ->
                  Comp({Choose{_}   |e}, [a])
handler allResults {
  case Return(x) -> [x]
  case Choose(k) -> k(true) ++ k(false)
}
```

Observe that the return value is lifted into a singleton list. The `Choose`-clause concatenates the outcomes obtained by interpreting the operation as `true` and `false`, respectively. We say that `allResults` is a *multi-shot handler*.

Finally, we have handlers that do not invoke continuations. These are familiar *exception handlers*. As an example consider the following handler, which returns `Just` the result of the computation or returns `Nothing` if the operation `Fail` is performed:

```
sig maybeResult : (Comp({Fail:Zero |e},       a)) ->
                   Comp({Fail{_}   |e}, Maybe(a))
handler maybeResult {
  case Return(x) -> Just(x)
  case Fail(_)   -> Nothing
}
```

The type system prevents invocation of the continuation in the `Fail`-clause, because the type `Zero` has zero inhabitants. Linear and exception handlers together constitute *affine handlers*.

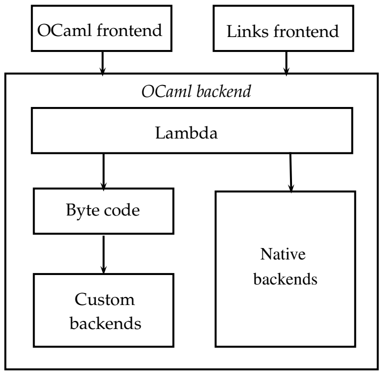

Figure 1. OCaml Backend

3. Compiler Infrastructure

We reuse most of the previous Links infrastructure. The Links frontend is type-checked and translated into a small, typed intermediate language in *A-normal form* (ANF). The Links interpreter implements a generalised CEK machine [4], which interprets ANF code.

Our compilation strategy is to translate the Links ANF language into the OCaml *Lambda* language, which is a small, untyped lambda calculus. The OCaml backend exposes a hierarchy of intermediate representations (IRs), where the top representation is known as Lambda. As shown in the Figure 1, the Lambda IR offers two different compilation options: byte code and native code. Therefore by targeting Lambda rather than a lower level IR, we achieve maximum flexibility as a translation into byte code, in principle, enables us to take advantage of custom backends such as `js_of_ocaml` to produce efficient JavaScript.

There are several semantic differences between Links and OCaml, e.g. Links employs structural typing, whilst OCaml predominantly employs nominal typing. In particular, Links employs row typing for effects, records, and variants, whereas OCaml only supports row typing for the latter. Exhibiting a faithful translation from Links to OCaml amounts to a lot of value boxing. Thus, we target Lambda for greater flexibility and control. We effectively subvert OCaml's typechecker by targeting Lambda, however the translation is safe as Links programs are already typechecked.

4. Runtime Representation

By using the OCaml backend we naturally inherit the OCaml runtime. OCaml implements effect handlers as heap-managed stack data structures, and as a consequence composition of handlers gives rise to n-element stacks at run-time. For example, the composition `randomResult(maybeResult(·))` is represented as a two-element stack. Thus, an invocation of an abstract operation amounts to performing a dynamic lookup for a suitable handler in a stack.

Since the primary use of handlers in OCaml is to express concurrency, OCaml handlers are affine; continuations can only be resumed at most once, and multiple invocations of a continuation causes a run-time error. Multi-shot handlers can be simulated by manually cloning continuations using `Obj.clone_continuation`. The cost of cloning is linear in the size of the handler stack. However, cloning is a fragile abstraction; if the handler stack contains at least one multi-shot handler, then every affine handler in the stack must be demoted to a multi-shot handler to be safe, because a multi-shot handler may consume a linear continuation more than once. Consequently, multi-shot handlers in OCaml break modularity.

In the Links compiler we use the cloning capability under the hood to implement multi-shot handlers. For example, our encoding of `allResults` amounts to the following in plain OCaml:

	State	8-Queens	20-Queens
Links Interpreter	76167	242	411517
Links Compiler	1619	1	1059
OCaml (native)	829	1	200

Table 1. Micro-benchmarks (execution times are in milliseconds)

```
let all_results m = match m () with
 | x -> [x]
 | effect Choose k ->
   let k' arg =
     continue (Obj.clone_continuation k) arg
   in k' true @ k' false
```

OCaml provides a unified syntax for pattern-matching on regular, effect, and exception patterns. The keyword `effect` begins an operation-clause. Essentially, we create a local function `k'`, which wraps the actual continuation `k`. An invocation of `k'` passes its argument to a fresh copy of the actual continuation. The `continue` function is provided by the standard library; given a continuation and a value, it invokes the continuation with that particular value. By default we implement every handler as a multi-shot handler.

5. Optimisations

Table 1 contains the results of a few micro-benchmarks from Kammar et al. [5]. Links Compiler corresponds to this work. While the results are promising, it is rather conservative to implement every handler as multi-shot. We would like to recover the efficiency of affine handlers, but without sacrificing abstraction.

Linearisation Handlers that use their continuations linearly should be promoted to linear handlers at compile time. We are currently working on applying the existing linear type system of Links to track the linearity of handlers.

Handler Resolution Traversing a large handler stack is likely to be costly. If the handler stack, or part of it, is known statically, then we can instantiate abstract operations at compile time.

6. Acknowledgements

The first and second authors were supported by EPSRC grants CDT in Pervasive Parallelism (EP/L01503X/1) and EP/K034413/1, respectively. This work is in collaboration with OCaml Labs.

References

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. FMCO '06, 2006.

[2] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective Concurrency through Algebraic Effects, 9 2015. OCaml Workshop.

[3] D. Hillerström. Handlers for Algebraic Effects in Links. Master's thesis, 2015.

[4] D. Hillerström and S. Lindley. Liberating Effects using Rows and Handlers. Draft, June, 2016.

[5] O. Kammar, S. Lindley, and N. Oury. Handlers in Action. ICFP, 2013.

[6] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. Haskell Symposium, 2015.

[7] G. D. Plotkin and M. Pretnar. Handling Algebraic Effects. Logical Methods in Computer Science, 2013.

[8] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. 2014.

[9] N. Wu and T. Schrijvers. Fusion for Free - Efficient Algebraic Effect Handlers. MPC, 2015.