Effect Handlers in Multicore OCaml

Daniel Hillerström, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, KC Sivaramakrishnan
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - Concurrency expressed through *effect handlers*
  - Will land upstream in Q2 2021
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  
- Concurrency expressed through *effect handlers*
  
- Will land upstream in Q2 2021

```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
```
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*

- Concurrency expressed through *effect handlers*

- Will land upstream in Q2 2021

```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
```
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - Concurrency expressed through *effect handlers*
  - Will land upstream in Q2 2021

```ocaml
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    try
      comp ()
    with effect E k ->
      print_string "1 ";
      continue k "2 ";
      print_string "4 ";
```

**effect declaration**

**computation**
Effect Handlers

• Multicore OCaml is an OCaml extension with native support for concurrency and shared-memory parallelism

✦ Concurrency expressed through effect handlers

✦ Will land upstream in Q2 2021

```ocaml
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
    with effect E k ->
      print_string "1 ";
      continue k "2 ";
      print_string "4 ";
```
Effect Handlers

• Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*

  ✦ Concurrency expressed through *effect handlers*

  ✦ Will land upstream in Q2 2021

```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
    with effect E k ->
      print_string "1 ";
      continue k "2 ";
      print_string "4 ";
```

effect declaration suspends current computation

computation

handler
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for concurrency and shared-memory parallelism
  - Concurrency expressed through effect handlers
  - Will land upstream in Q2 2021

```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

- effect declaration
- suspends current computation
- delimited continuation
- handler
Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*

  - Concurrency expressed through *effect handlers*
  - Will land upstream in Q2 2021

```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
```
Compilation

effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 "

let main () =
    try
        comp ()
        with effect E k ->
            print_string "1 ";
            continue k "2 ";
            print_string "4 "
```ocaml
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

**Compilation**

- `pc` refers to the program counter, indicating the current instruction.
- `sp` refers to the stack pointer.
- The `main` function is the entry point of the program, handling effects and printing strings.
- The `comp` function is called with the `E` effect, performing side effects and printing strings.
- The `try` block catches any effects and handles them accordingly.
```

effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 ";

let main () =
    try
        comp ()
        with effect E k ->
            print_string "1 ";
            continue k "2 ";
            print_string "4 ";
```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
Compilation

```ocaml

effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
```

parent
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

0
Compilation

effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 "

let main () =
    try
        comp ()
    with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 "

effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 ";

let main () =
    try
        comp ()
    with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 ";
effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 ";

let main () =
    try
        comp ()
    with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 ";
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 ";
effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 

let main () =
    try
        comp ()
    with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 ";
effect E : string

let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 ";

let main () =
    try
        comp ()
        with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 "

Handlers can be nested

```
effect A : unit
effect B : unit

let baz () = perform A

let bar () = try baz () with effect B k -> continue k()

let foo () = try bar () with effect A k -> continue k()
```
effect A : unit
effect B : unit

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A k ->
    continue k ()
effect A : unit
effect B : unit

let baz () =
  perform A

let bar () =
  try
    baz ()
    with effect B k ->
    continue k ()

let foo () =
  try
    bar ()
    with effect A k ->
    continue k ()

Handlers can be nested
Handlers can be nested

```
let baz () = 
    perform A

let bar () = 
    try 
        baz ()
        with effect B k -> 
            continue k ()

let foo () = 
    try 
        bar ()
        with effect A k -> 
            continue k ()
```

- Linear search through handlers
- *Handler stacks shallow in practice*
Deep-dive into perform
Deep-dive into `perform`

- Full power of pattern matching for matching effects
  - Tag test + branching is compiled to a function
Deep-dive into perform

• Full power of pattern matching for matching effects
  ✦ Tag test + branching is compiled to a function

https://github.com/ocaml-multicore/ocaml-multicore/blob/parallel_minor_gc/runtime/amd64.S#L865
Performance

- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - For reference, memory read latency is 90 ns (local NUMA node) and 145 ns (remote NUMA node)
Performance

- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - For reference, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

```ocaml
let foo () =
  (* a *)
try
  (* b *)
  perform E
  (* d *)
with effect E k ->
  (* c *)
  continue k ()
  (* e *)
```
Performance

- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - For reference, memory read latency is 90 ns (local NUMA node) and 145 ns (remote NUMA node)

```haskell
let foo () =
  (* a *)
try
  (* b *)
perform E
  (* d *)
with effect E k ->
  (* c *)
continue k ()
  (* e *)
```

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>a to b</td>
<td>Create a new stack &amp; run the computation</td>
</tr>
<tr>
<td>b to c</td>
<td>Performing &amp; handling an effect</td>
</tr>
<tr>
<td>c to d</td>
<td>Resuming a continuation</td>
</tr>
<tr>
<td>d to e</td>
<td>Returning from a computation &amp; free the stack</td>
</tr>
</tbody>
</table>

- Each of the instruction sequences involves a stack switch
Performance

- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - For reference, memory read latency is 90 ns (local NUMA node) and 145 ns (remote NUMA node)

```ocaml
let foo () =
  (* a *)
try
  (* b *)
  perform E
  (* d *)
with effect E k ->
  (* c *)
  continue k ()
  (* e *)
```

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Significance</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a to b</td>
<td>Create a new stack &amp; run the computation</td>
<td>2479</td>
</tr>
<tr>
<td>b to c</td>
<td>Performing &amp; handling an effect</td>
<td>122</td>
</tr>
<tr>
<td>c to d</td>
<td>Resuming a continuation</td>
<td>189</td>
</tr>
<tr>
<td>d to e</td>
<td>Returning from a computation &amp; free the stack</td>
<td>155</td>
</tr>
</tbody>
</table>

- Each of the instruction sequences involves a stack switch
Performance: Generators
Performance: Generators

- Traverse a complete binary-tree of depth 25
Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
Performance: Generators

• Traverse a complete binary-tree of depth 25

• Iterator — idiomatic recursive traversal

• Generator — `next()` function to consume elements on-demand
  ✦ Hand-written generator (hw-generator)
    ✦ CPS translation + defunctionalization to remove intermediate closure allocation
  ✦ Generator using effect handlers (eh-generator)
    ✦ $2 \times (2^{25} - 1) + 2 = 2^{26}$ stack switches
Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
- Generator — `next()` function to consume elements on-demand
  - Hand-written generator (hw-generator)
    - CPS translation + defunctionalization to remove intermediate closure allocation
  - Generator using effect handlers (eh-generator)
    - $2 \times (2^{25} - 1) + 2 = 2^{26}$ stack switches

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>202</td>
</tr>
<tr>
<td>hw-generator</td>
<td>761 (3.76x)</td>
</tr>
<tr>
<td>eh-generator</td>
<td>1879 (9.30x)</td>
</tr>
</tbody>
</table>

Multicore OCaml
Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
- Generator — `next()` function to consume elements on-demand

  - Hand-written generator (hw-generator)
    - CPS translation + defunctionalization to remove intermediate closure allocation
  - Generator using effect handlers (eh-generator)
    - $2 \times (2^{25} - 1) + 2 = 2^{26}$ stack switches

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>202</td>
</tr>
<tr>
<td>hw-generator</td>
<td>761 (3.76x)</td>
</tr>
<tr>
<td>eh-generator</td>
<td>1879 (9.30x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>492</td>
</tr>
<tr>
<td>generator</td>
<td>43842 (89.1x)</td>
</tr>
</tbody>
</table>
Performance: WebServer

- Effect handlers for asynchronous I/O
- Variants
  - Go + net/http
  - OCaml + http/af + Async (explicit callbacks)
  - OCaml + http/af + Effect handlers
- Latency measured using wrk2
Performance: WebServer

- Effect handlers for asynchronous I/O
- Variants
  - Go + net/http
  - OCaml + http/af + Async (explicit callbacks)
  - OCaml + http/af + Effect handlers
- Latency measured using wrk2
Thank you!

- Multicore OCaml
  - https://github.com/ocaml-multicore/ocaml-multicore

- A collection of effect handlers examples
  - https://github.com/ocaml-multicore/effects-examples

- JS generator example
  - https://github.com/kayceesrk/wasmfx/tree/master/cg_4_aug_20