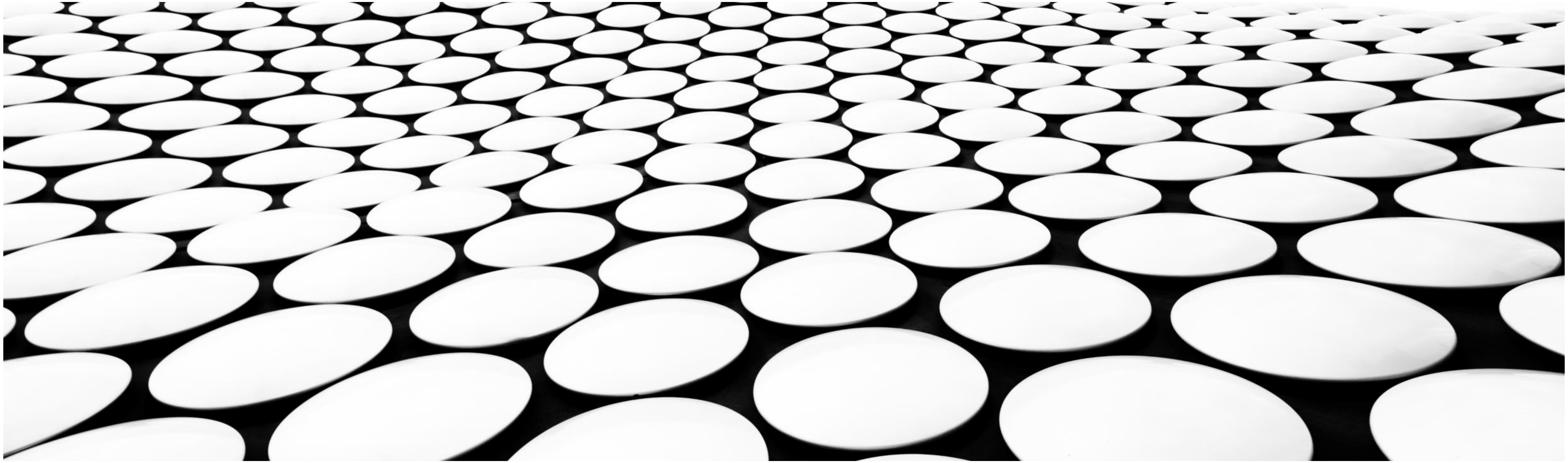
SUPPORTING MULTICORE OCAML EFFECTS

ROSS TATE



LANGUAGE FEATURE

DECLARING AN EFFECT

- effect Foo : input_type -> output_type
  with function Foo (input : input_type) : output_type = output_expr
  - Foo is now a constructor with input input_type and output (output_type eff)
  - output_expr is the “default” handler
    - If no default handler is specified, it is simply “raise Unhandled ()”

PERFORMING AN EFFECT

- perform : α eff -> α
  - Gives the (α eff) to the closest dynamically-scoped enclosing matching handler to get an α
  - If there is no such handler, call the default handler function
    - If that raises an exception, propagate that exception to the call to “perform”
    - Or if that performs an effect, handle it as if it had no enclosing handler
    - (These are designed to avoid allocating stacks for unhandled effects and default handlers.)

HANDLING AN EFFECT

- match expr with v → returner | exception e → catcher | effect e k → handler
  - Types
    - $\text{expr} : \alpha$
    - $v : \alpha \vdash \text{returner} : \beta$
    - $e : \text{exn} \vdash \text{catcher} : \beta$
    - $\gamma \mid e : \gamma\ \text{eff},\ k : (\gamma, \beta)\ \text{continuation} \vdash \text{handler} : \beta$
  - the continuation k is enclosed by this match (deep semantics for algebraic effects)
  - performs and raises done by returner/catcher/handler are not enclosed by this match

USING CONTINUATIONS

- continue : (α, β) continuation $\to \alpha \to \beta$
  - Runs the continuation within current dynamic scope resuming it with the given value
- discontinue : (α, β) continuation $\to$ exn $\to \beta$
  - Runs the continuation within current dynamic scope by throwing the given exception within it

NATIVE IMPLEMENTATION

VALUE REPRESENTATION

- α eff
  - Just an object with a tag identifying the effect name and a corresponding payload
- (α, β) continuation
  - A pair of stacks, one set to resume with an α value and one needing a parent expecting a β value
  - The former is an ancestor of the latter
- Stacks
  - At the root of every stack is a parent to return to (which is null if the stack is suspended)
  - At the root of the stack is a handler dictionary and a generic handler (each possibly null)
    - The dictionary is for the common case where the match names specific effects

IMPLEMENTING MATCH

- Create a new stack
  - with expr as its first frame
  - with the current stack as its parent
  - with the appropriate handler dictionary and/or generic handler
- Switch to the new stack

IMPLEMENTING CONTINUE/DISCONTINUE

- continue
  - Set the current stack as the parent of the appropriate stack
  - Switch to the appropriate stack with the given value
- discontinue
  - Set the current stack as the parent of the appropriate stack
  - Switch to the appropriate stack and throw the given exception

IMPLEMENTING PERFORM (HANDLED)

- Let stack be the current stack
- while stack is not null and does not have a matching handler
  - Let stack be stack’s parent
- If stack is not null
  - Switch to and clear stack’s parent and call the appropriate handler with the given payload on that stack (using the pair of stack and the current stack as the continuation)

IMPLEMENTING PERFORM (UNHANDLED)

- … Otherwise
  - Store current stack’s parent and handlers in local frame
  - Clear current stack’s parent and handlers
  - Call default handler for the given effect with given payload
  - Restore current stack’s parent and handlers from local

SUPPORTING MULTICORE OCAML USING ALGEBRAIC EFFECTS

VALUE REPRESENTATION

- eff is lowered to ref (struct dataref)
  - The dataref is the effect’s tag (every effect also has an associated rtt)
- continuation is lowered to ref (struct (ref $handlers) (cont ([eqref] -> [eqref])))
  - Where $handlers describes the (internal) data structure for effect handlers, e.g. hashmap

EFFECTS

- effect $ocaml_eff : [(ref (struct dataref))] -> [eqref]
  - Need a single effect to support first-class effect handlers

IMPLEMENTING PERFORM (HANDLED)

suspend $ocaml_eff

IMPLEMENTING CONTINUE [PRESENTED VERSION]

```
(local $k (ref (struct (ref $handlers)) (cont ([eqref] -> [eqref]))))        (local $v eqref)                                   ;; the given inputs
(local $hs (ref $handlers))     (local $c (ref (cont ([eqref] -> [eqref]))))     (local $h (ref $handler))      (local $e (ref (struct dataref))) ;; temporaries
(local.set $hs (struct.get 0 (local.get $k))) (local.set $c (struct.get 1 (local.get $k)))
(block $done ([] -> [eqref])
        (loop $retry ([] -> [eqref])
                (block $suspended ([] -> [(ref (struct dataref)) (cont ([eqref] -> [eqref]))])
                        (resume (tag $ocaml_eff $suspended) (local.get $v) (local.get $c)) ;; eqref put onto the stack
                        (br $done) ;; eqref already on the stack
                )
                (local.set $c) ;; continuation was on the stack
                (local.set $e) ;; effect was on the stack
                (local.set $h (call $get_handler (local.get $hs) (struct.get 0 (local.get $e))))
                (br_if $retry (ref.is_null $h))
                (call $call_handler (local.get $h) (local.get $e) (local.get $handlers) (local.get $c)) ;; eqref put onto the stack
        )
)
```

Performs a stack switch just to find out that this stack did not need to be switched to.

IMPLEMENTING CONTINUE [CORRECTED VERSION]

```
(local $k (ref (struct (ref $handlers)) (cont ([eqref] -> [eqref]))))            (local $v eqref)                                        ;; the given inputs
(local $hs (ref $handlers))              (local $c (ref (cont ([eqref] -> [eqref])))    (local $h (ref $handler))          (local $e (ref (struct dataref)))    ;; temporaries
(local.set $hs (struct.get 0 (local.get $k))) (local.set $c (struct.get 1 (local.get $k)))
(block $done ([] -> [eqref])
        (loop $retry ([] -> [eqref])
                (block $suspended ([] -> [(ref (struct dataref)) (cont ([eqref] -> [eqref]))])
                        (resume (tag $ocaml_eff $suspended) (local.get $v) (local.get $c)) ;; eqref put onto the stack
                        (br $done) ;; eqref already on the stack
                )
                (local.set $c) ;; continuation was on the stack
                (local.set $e) ;; effect was on the stack
                (local.set $h (call $get_handler (local.get $hs) (struct.get 0 (local.get $e))))
                (if (ref.is_null $h)
                        (suspend $ocaml_eff (local.get $e)) ;; eqref put onto stack
                        (local.set $v) ;; eqref was on the stack
                        (br $retry)
                else
                        (call $call_handler (local.get $h) (local.get $e) (local.get $handlers) (local.get $c)) ;; eqref put onto the stack
)       )       )
```

Performs a stack switch just to find out that this stack did not need to be switched to.

IMPLEMENTING DISCONTINUE

- Cannot be done with existing instruction set
  - resume_throw does not allow resumer to handle effects
- Fix by adding resume_throw_with_handler
  - can only propagate OCaml exceptions (not foreign exceptions)

IMPLEMENTING PERFORM (UNHANDLED)

- Cannot be done straightforwardly
  - (suspend $ocaml_eff) traps if no handler is found
- All entry points must allocate a continuation and resume it with “default” handler
  - That “default” handler must allocate a continuation to run the effect-specific default handler on and likewise resume it with the “default” handler
    - Requires recursion, as well as resume_throw_with_handler for propagating OCaml exceptions

Default handlers were specifically designed to avoid the need to allocate stacks.

IMPLEMENTING MATCH

- Allocate a ref $handlers appropriately
- Allocate a cont with the function for the expr (with unbound variables)
- Resume the cont with the value of those variables
  - Using a handler that then enters the same loop as the implementation of continue

SUPPORTING MULTICORE OCAML USING FIRST-CLASS STACKS

TYPES

- OCaml stacks have a bunch of fields at the root for stack-specific data
- stack_extend $struct_type $label_type $label_type+
  - Defines a new stack type with fields from the specified struct type
  - The first label type is its return type
  - The remaining label types are its resumption types
  - Stacks are either mounted (onto some parent) or suspended (no parent but labels defined)
- $ocaml_stack = stack_extend (struct (ref $ocaml_stack) (ref $handlers)) [eqref] [eqref]

VALUE REPRESENTATION

- continuation lowers to ref (struct (ref $ocaml_stack) (ref $ocaml_stack)))
  - The first is the stack in need of a parent
  - The second is the stack waiting to be resumed

INSTRUCTIONS

- stack.current
  - Returns a reference to (the root of) the current stack
  - Null if the current stack is not allowed to be switched away from
  - Function signature (typically) defines the type of the current stack (if switchable)
- stack.switch index $label+
  - Transfers control to the target stack via the label at the specified index
  - The target stack must have the same return type as the current stack
  - Transfers mounting point, making current stack suspended and the target stack mounted
  - The labels are the resumption points for the current suspended stack
- stack.switch_call $func index $label+
  - Transfers like stack.switch, except it calls $func (with args from value stack) on the target stack with the label at the specified index as the call’s return address

IMPLEMENTING PERFORM

```
$current, $stack := stack.current
while ($stack != null) {
    $handlers := struct.get 1 (local.get $stack)
    $handler := call $get_handler (locals.get $handlers $eff)
    $parent := struct.get 0 (local.get $stack)
    if ($handler != null) {
        struct.set 0 (ref.null $ocaml_stack) $stack
        stack.switch_call $call_handler 0 $performed_label (locals.get $handler $eff $stack $current $parent)
    } ;; else $handler is null
    $stack := $parent
}
stack.conceal { ;; makes the current stack unswitchable, so that stack.current returns null within this block
    call $call_default_handler (local.get $eff) ;; puts an eqref onto the stack
}
$performed_label: ;; has label type [eqref]
```

IMPLEMENTING CONTINUE

```
$root := struct.get 0 (local.get $k)
$leaf := struct.get 1 (local.get $k)
$current := stack.current
if ($current != null) {
    struct.set 0 (local.get $current) (local.get $root) ;; set $root’s parent to $current
    stack.switch 0 $continued_label (local.get $v) (local.get $leaf)
} ;; else $current is null
stack.mount 0 (local.get $v) (local.get $leaf) ;; sets the given stack to return to this point
$continued_label: ;; has label type [eqref]
```

IMPLEMENTING DISCONTINUE

```
$root := struct.get 0 (local.get $k)
$leaf := struct.get 1 (local.get $k)
$current := stack.current
if ($current != null) {
    struct.set 0 (local.get $current) (local.get $root) ;; set $root's parent to $current
    stack.switch_call $ocaml_throw 0 $continued_label (local.get $exn) (local.get $leaf)
} ;; else $current is null
stack.mount_call $ocaml_throw 0 (local.get $exn) (local.get $leaf)
$continued_label: ;; has label type [eqref]
```

DESIGN NOTES

- All instructions are constant time
- Ensures both composition and (strong) abstraction
- Instructions were designed prior to considering Multicore OCaml
  - no changes were necessary to accommodate full feature set
  - Additional optimizations like for tail-resumptive handlers also already supported
- Admits a convenient and efficient JS API
  - Except that all exceptions must be converted between JS and $ocaml_exn at boundary