Implementing Algebraic Effects in C
"Monads for Free in C"

Microsoft Research Technical Report MSR-TR-2017-23

Daan Leijen

Microsoft Research
daan@microsoft.com

Abstract. We describe a full implementation of algebraic effects and handlers as a library in standard and portable C99, where effect operations can be used just like regular C functions. We use a formal operational semantics to guide the C implementation at every step where an evaluation context corresponds directly to a particular C execution context. Finally we show a novel extension to the formal semantics to describe optimized tail resumptions and prove that the extension is sound. This gives two orders of magnitude improvement to the performance of tail resumptive operations (up to about 150 million operations per second on a Core i7@2.6GHz)

Updates:
2017-06-30: Added Section A.4 on C++ support.
2017-06-19: Initial version.

1. Introduction

Algebraic effects [35] and handlers [36, 37] come from category theory as a way to reason about effects. Effects come with a set of operations as their interface, and handlers to give semantics to the operations. Any free monad [2, 17, 40] can be expressed using algebraic effect handlers: the operations describe an algebra that gives rise to a free monad, whereas the handler is the *fold* over that algebra giving its semantics.

This makes algebraic effects highly expressive and practical, and they can describe many control flow constructs that are usually built into a language or compiler. Examples include, exception handling, iterators, backtracking, and async/await style asynchronous programming. Once you have algebraic effects, all of these abstractions can be implemented *as a library* by the user. In this article, we describe a practical implementation of algebraic effects and handlers as a library itself in C. In particular,

- We describe a full implementation of algebraic effects and handlers in standard and portable C99. Using effect operations is just like calling regular C functions. Stacks are always restored at the same location and regular C semantics are preserved.

- Even though the semantics of algebraic effects are simple, the implementation in C is not straightforward. We use a formal operational semantics to guide the C implementation at every step. In particular, we use context based semantics where a formal context corresponds directly to a particular C execution context.
- We show a novel extension to the formal semantics to describe optimized tail resumptions and prove that the extension is sound. This gives two orders of magnitude improvement to the performance of tail resumptive operations (up to about 150 million operations per second on a Core i7).

At this point using effects in C is nice, but defining handlers is still a bit cumbersome. Its interface could probably be improved by providing a C++ wrapper. For now, we mainly see the library as a target for library writers or compilers. For example, the P language [10] is a language for describing verifiable asynchronous state machines, and used for example to implement and verify the core of the USB device driver stack that ships with Microsoft Windows 8. Compiling to C involves a complex CPS-style transformation [19, 26] to enable async/await style programming [5] with a `receive` statement – using the effects library this transformation is no longer necessary and we can generate straightforward C code instead. Similarly, we hope to integrate this library with `libuv` [29] (the asynchronous C library underlying Node [43]) and improve programming with `libuv` directly from C or C++ using async/await style abstractions [12, 27].

The library is publicly available as `libhandler` under an open-source license [28]. For simplicity the description in this paper leaves out many details and error handling etc. but otherwise follows the real implementation closely.

2. Overview

We necessarily give a short overview here of using algebraic effects in C. For how this can look if a language natively supports effects, we refer to reader to other work [3, 18, 26, 27, 30]. Even though the theory of algebraic effects describes them in terms of monads, we use a more operational view in this article that is just as valid – and view effects as *resumable exceptions*. Therefore we start by describing how to implement regular exceptions using effect handlers.

2.1. Exceptions

We start by implementing exceptions as an algebraic effect. First we declare a new effect `exn` with a single operation `raise` that takes a `const char*` argument:

```
DEFINE_EFFECT1(exn, raise)
DEFINE_VOIDOP1(exn, raise, string)
```

Later we will show exactly what these macros expand to. For now, it is enough to know that the second line defines a new operation `exn_raise` that we can call as any other C function, for example:

```
int divexn( int x, int y ) {
  return (y!=0 ? x / y : exn_raise("divide by zero")); }
```

Since using an effect operation is just like calling a regular C function, this makes the library very easy to use from a user perspective.

Defining *handlers* is a bit more involved. Here is a possible handler function for our `raise` operation:

```
value handle_exn_raise(resume* r, value local, value arg) {
  printf("exception raised: %s\n", string_value(arg));
  return value_null; }
```

The `value` type is used here to simulate parametric polymorphism in C and is typedef'd to a `long long`, together with some suitable conversion macros; in the example we use `string_value` to cast the `value` back to the `const char*` argument that was passed to `exn_raise`.

Using the new operation handler is done using the `handle` library function. It is a bit cumbersome as we need to set up a handler definition (`handlerdef`) that contains a table of all operation handlers[1]:

```
const operation _exn_ops[] = {
  { OP_NORESUME, OPTAG(exn,raise), &handle_exn_raise } };
const handlerdef _exn_def  = { EFFECT(exn), NULL, NULL, NULL, _exn_ops };

value my_exn_handle(value(*action)(value), value arg) {
  return handle(&_exn_def, value_null, action, arg); }
```

Using the handler, we can run the full example as:

```
value divide_by(value x) {
  return value_long(divexn(42,long_value(x)));
}
int main() {
  my_exn_handle( divide_by, value_long(0));
  return 0; }
```

When running this program, we'll see:

```
exception raised: divide by zero
```

A handler definition has as its last field a list of operations, defined as:

```
typedef struct _operation {
  const opkind  opkind;
  const optag   optag;
  value         (*opfun)(resume* r, value local, value arg);
} operation;
```

The operation tag `optag` uniquely identifies the operation, while the `opkind` describes the kind of operation handler:

```
typedef enum _opkind {
  OP_NULL,
```

[1] Ah, if only we had lambda expressions and virtual methods in C99 ;-)

```
  OP_NORESUME,  // never resumes
  OP_TAIL,      // only uses `resume` in tail-call position
  OP_SCOPED,    // only uses `resume` inside the handler
  OP_GENERAL    // `resume` is a first-class value
} opkind;
```

These operation kinds are used for optimization and restrict what an operation handler can do. In this case we used `OP_NORESUME` to signify that our operation handler never resumes. We'll see examples of the other kinds in the following sections.

The `DEFINE_EFFECT` macro defines a new effect. For our example, it expands into something like:

```
const char*  effect_exn[3]    = {"exn","exn_raise",NULL};
const optag  optag_exn_raise  = { effect_exn, 1 };
```

An effect can now be uniquely identified by the address of the `effect_exn` array, and `EFFECT(exn)` expands simply into `effect_exn`. Similarly, `OPTAG(exn,raise)` expands into `optag_exn_raise`. Finally, the `DEFINE_VOIDOP1` definition in our example expands into a small wrapper around the library `yield` function:

```
void exn_raise( const char* s ) {
  yield( optag_exn_raise, value_string(s) ); }
```

which "yields" to the innermost handler for `exn_raise`.

2.2. Ambient State

As we saw in the exception example, the handler for the `raise` operation took a `resume*` argument. This can be used to *resume* an operation at the point where it was issued. This is where the true power of algebraic effects come from (and why we can view them as resumable exceptions). As another example, we are going to implement *ambient state* [27].

```
DEFINE_EFFECT(state,put,get)
DEFINE_OP0(state,get,int)
DEFINE_VOIDOP1(state,put,int)
```

This defines a new effect `state` with the operations `void state_put(int)` and `int state_get()`. We can use them as any other C function:

```
void loop() {
  int i;
  while((i = state_get()) > 0) {
    printf("state: %i\n", i);
    state_put(i-1);
  }
}
```

We call this *ambient state* since it is dynamically bound to the innermost state handler – instead of being global or local state. This captures many common patterns in practice. For example, when writing a web server, the "current" request object needs to be passed around manually to each function in general; with algebraic effects you can just create a `request` effect that gives access to the current request without having to pass it explicitly to every function. The handler for `state` uses the `local` argument to store the current state:

```
value handle_state_get( resume* r, value local, value arg ) {
  return tail_resume(r,local,local);
}
value handle_state_put( resume* r, value local, value arg ) {
  return tail_resume(r,arg,value_null);
}
```

The `tail_resume` (or `resume`) library function resumes an operation at its yield point. It takes three arguments: the resumption object `r`, the new value of the local handler state `local`, and the return value for the `yield` operation. Here the `handle_state_get` handler simply returns the current local state, whereas `handle_state_put` returns a null value but resumes with its local state set to `arg`. The `tail_resume` operation can only be used in a tail-call position and only with `OP_TAIL` operations, but it is much more efficient than using a general `resume` function (as shown in Section 5).

2.3. Backtracking

You can enter a room once, yet leave it twice.
— Peter Landin [23, 24]

In the previous examples we looked at an abstractions that never resume (e.g. exceptions), and an abstractions that resumes once (e.g. state). Such abstractions are common in most programming languages. Less common are abstractions that can resume more than once. Examples of this behavior can usually only be found in languages like Lisp and Scheme, that implement some variant of `callcc` [42]. A nice example to illustrate multiple resumptions is the ambiguity effect:

```
DEFINE_EFFECT1(amb,flip)
DEFINE_BOOLOP0(amb,flip,bool)
```

which defines one operation `bool amb_flip()` that returns a boolean. We can use it as:

```
bool xor() {
  bool p = amb_flip();
  bool q = amb_flip();
  return ((p || q) && !(p && q)); }
```

One possible handler just returns a random boolean on every flip:

```
value random_amb_flip( resume* r, value local, value arg ) {
  return tail_resume(r, local, value_bool( rand()%2 )); }
```

but a more interesting handler resumes *twice*: once with a `true` result, and once with `false`. That way we can return a list of all results from the handler:

```
value all_amb_flip( resume* r, value local, value arg ) {
  value xs = resume(r,local, value_bool(true));
  value ys = resume(r,local, value_bool(false)); // resume again at `r`!
  return list_append(xs,ys); }
```

Note that the results of the `resume` operations are lists themselves since a resumption runs itself under the handler. When we run the `xor` function under the `all_amb` handler, we get back a list of all possible results of running `xor`, printed as:

```
[false,true,true,false]
```

In general, resuming more than once is a dangerous thing to do in C. When using mutable state or external resources, most C code assumes it runs at most once, for example closing file handles or releasing memory when exiting a lexical scope. Resuming again from inside such scope would give invalid results.

Nevertheless, you can make this work safely if you for example manage state using effect handlers themselves which take care of releasing resources correctly. Multiple resumptions are also needed for implementing async/await interleaving where the resumptions are safe by construction.

Combining the `state` and `amb` handlers is also possible; if we put `state` as the outermost handler, we get a "global" state per ambiguous strand, while if we switch the order, we get a "local" state per ambiguous strand. We refer to other work for a more in-depth explanation [3, 26].

2.4. Asynchronous Programming

Recent work shows how to build `async`/`await` abstractions on top of algebraic effects [12, 27]. We plan to use a similar approach to implement a nice interface to programming `libuv` directly in C. This is still work in progress and we only sketch here the basic approach to show how algebraic effects can enable this. We define an asynchronous effect as:

```
DEFINE_EFFECT1(async,await)
int  await( uv_req_t* req );
void async_callback(uv_req_t* req);
```

The handler for `async` only needs to implement `await`. This operation receives an asynchronous `libuv` request object `uv_req_t` where it only stores its resumption in the request custom `data` field. However, it does not resume itself! Instead it returns directly to the outer `libuv` event loop which invokes the registered callbacks when an asynchronous operation completes.

```
value handle_async_await( resume* r, value local, value arg ) {
  uv_req_t* req = (uv_req_t*)ptr_value(arg);
  req->data = r;
  return value_null; }
```

We ensure that the asynchronous `libuv` functions all use the same `async_callback` function as their callback. This in turn calls the actual resumption that was stored in the `data` field by `await`:

Expressions	e	$::=$	$e(e)$	application
		$\mid$	$\mathsf{val}\; x = e;\, e$	binding
		$\mid$	$\mathsf{handle}_h(e)$	handler
		$\mid$	v	value
Values	v	$::=$	$x \mid c \mid op \mid \lambda x.\, e$	
Clauses	h	$::=$	$\mathsf{return}\; x \rightarrow e$	
		$\mid$	$op(x) \rightarrow e;\, h$	$op \notin h$

Figure 1. Syntax of expressions in λ_{eff}

```
void async_callback( uv_req_t* req ) {
  resume* r = (resume*)req->data;
  resume(r, req->result); }
```

In other words, instead of explicit callbacks with the current state encoded in the `data` field, the current execution context is fully captured by the first-class resumption provided by our library. We can now write small wrappers around the `libuv` asynchronous API to use the new `await` operation, for example, here is the wrapper for an asynchronous file stat:

```
int async_stat( const char* path, uv_stat_t* stat ) {
  uv_fs_t* req = (uv_fs_t*)malloc(sizeof(uv_fs_t));
  uv_stat(uv_default_loop(), req, path, async_callback);  // register
  int err = await((uv_req_t*)req);                         // and await
  *stat = req->statbuf;
  uv_fs_req_cleanup(req);
  free(req);
  return content; }
```

The asynchronous functions can be called just like regular functions:

```
uv_stat_t stat;
int err = async_stat("foo.txt", &stat);  // asynchronous!
printf("Last access time: %li\n", (err < 0 ? 0 : stat.st_atim.tv_sec));
```

This would make it as easy to use `libuv` as using the standard C libraries for doing I/O.

3. Operational Semantics

An attractive feature of algebraic effects and handlers is that they have a simple operational semantics that is well-understood. To guide our implementation in C we define a tiny core calculus for algebraic effects and handlers that is well-suited to reason about the operational behavior.

Figure 1 shows the syntax of our core calculus, λ_{eff}. This is equivalent to the definition given by Leijen [26]. It consists of basic lambda calculus extended with handler definitions h and operations op. The calculus can also be typed

Evaluation contexts:

$$\begin{array}{lcll} \mathsf{E} & ::= & [\,] \mid \mathsf{E}(e) \mid v(\mathsf{E}) \mid op(\mathsf{E}) \mid \mathsf{val}\; x \;=\; \mathsf{E};\; e \mid \mathsf{handle}_h(\mathsf{E}) & \\ \mathsf{X}_{op} & ::= & [\,] \mid \mathsf{X}_{op}(e) \mid v(\mathsf{X}_{op}) \mid \mathsf{val}\; x \;=\; \mathsf{X}_{op};\; e & \\ & \mid & \mathsf{handle}_h(\mathsf{X}_{op}) & \mathsf{if}\; op \notin h \end{array}$$

Reduction rules:

$$\begin{array}{llcll} (\delta) & c(v) & \longrightarrow & \delta(c, v) & \text{if } \delta(c, v) \text{ is defined} \\ (\beta) & (\lambda x.\, e)(v) & \longrightarrow & e[x \mapsto v] & \\ (let) & \mathsf{val}\; x \;=\; v;\; e & \longrightarrow & e[x \mapsto v] & \\ & & & & \\ (return) & \mathsf{handle}_h(v) & \longrightarrow & e[x \mapsto v] & \\ & & & \text{with} & \\ & & & \quad (\mathsf{return}\; x \rightarrow e) \;\in h & \\ & & & & \\ (handle) & \mathsf{handle}_h(\mathsf{X}_{op}[op(v)]) & \longrightarrow & e[x \mapsto v,\; resume \mapsto \lambda y.\, \mathsf{handle}_h(\mathsf{X}_{op}[y])] & \\ & & & \text{with} & \\ & & & \quad (op(x) \rightarrow e) \;\in h & \end{array}$$

Figure 2. Reduction rules and evaluation contexts

using regular static typing rules [18, 26, 38]. However, we can still give a dynamic *untyped* operational semantics: this is important in practice as it allows an implementation algebraic effects without needing explicit types at runtime.

Figure 2 defines the semantics of λ_{eff} in just five evaluation rules. It has been shown that well-typed programs cannot go 'wrong' under these semantics [26]. We use two evaluation contexts: the E context is the usual one for a call-by-value lambda calculus. The X_{op} context is used for handlers and evaluates down through any handlers that do *not* handle the operation *op*. This is used to express concisely that the 'innermost handler' handles particular operations.

The E context concisely captures the entire evaluation context, and is used to define the evaluation function over the basic reduction rules: $\mathsf{E}[e] \longmapsto \mathsf{E}[e']$ iff $e \longrightarrow e'$. The first three reduction rules, (δ), (β), and (let) are the standard rules of call-by-value evaluation. The final two rules evaluate handlers. Rule $(return)$ applies the return clause of a handler when the argument is fully evaluated. Note that this evaluation rule subsumes both lambda- and let-bindings and we can define both as a reduction to a handler without any operations:

$$\begin{array}{lcl} (\lambda x.\, e_1)(e_2) & \equiv & \mathsf{handle}\{\mathsf{return}\; x \rightarrow e_1\}(e_2) \\ \mathsf{val}\; x \;=\; e_1;\; e_2 & \equiv & \mathsf{handle}\{\mathsf{return}\; x \rightarrow e_2\}(e_1) \end{array}$$

These equivalences are used in the Frank language [30] to express everything in terms of handlers.

The next rule, $(handle)$, is where all the action is. Here we see how algebraic effect handlers are closely related to delimited continuations as the evaluation rules captures a delimited 'stack' $\mathsf{X}_{op}[op(v)]$ under the handler h. Using a X_{op} context ensures by construction that only the innermost handler containing a clause for op, can handle the operation $op(v)$. Evaluation continues with the expression ϵ but besides binding the parameter x to v, also the *resume* variable is bound to the continuation: $\lambda y.\,\mathsf{handle}_h(\mathsf{X}_{op}[y])$. Applying *resume* results in continuing evaluation at X_{op} with the supplied argument as the result. Moreover, the continued evaluation occurs again under the handler h.

Resuming under the same handler is important as it ensures that our semantics correspond to the original categorical interpretation of algebraic effect handlers as a *fold* over the effect algebra [37]. If the continuation is not resumed under the same handler, it behaves more like a *case* statement doing only one level of the *fold*. Such handlers are sometimes called *shallow handlers* [21, 30].

For this article we do not formalize parameterized handlers as shown in Section 2.2. However the reduction rule is straightforward. For example, a handler with a single parameter p is reduced as:

$$\mathsf{handle}_h(p\ =\ v_p)(\mathsf{X}_{op}[op(v)]) \quad\longrightarrow\quad \{\ op(v) \to e\ \in h\,\}$$
$$e[x \mapsto v,\ p \mapsto v_p,\ resume \mapsto \lambda q\,y.\,\mathsf{handle}_h(p\ =\ q)(\mathsf{X}_{op}[y])]$$

3.1. Dot Notation

The C implementation closely follows the formal semantics. We will see that we can consider the contexts as the current evaluation context in C, i.e. the call stack and instruction pointer. To make this more explicit, we use *dot* notation to express the notion of a context as call stack more clearly. We write $\cdot$ as a right-associative operator where $e\cdot\, e' \equiv e(e')$ and $\mathsf{E}\cdot\, e \equiv \mathsf{E}[e]$. Using this notation, we can for example write the (*handle*) rule as:

$$\mathsf{handle}_h\cdot\,\mathsf{X}_{op}\cdot\,op(v) \;\longrightarrow\; e[x \mapsto v,\ resume \mapsto \lambda y.\,\mathsf{handle}_h\cdot\,\mathsf{X}_{op}\cdot\,y]$$

where $(op(x) \to e) \in h$. This more clearly shows that we evaluate $op(v)$ under a current "call stack" $\mathsf{handle}_h\cdot\,\mathsf{X}_{op}$ (where h is the innermost handler for op as induced by the grammar of X_{op}).

4. Implementing Effect Handlers in C

The main contribution of this paper is showing how we can go from the operational semantics on an idealized lambda-calculus to an implementation as a C library. All the regular evaluation rules like application and let-bindings are already part of the C language. Of course, there are no first-class lambda expressions either so we must make do with top-level functions only. So, our challenge is to implement the (*handle*) rule:

$$\mathsf{handle}_h\cdot\,\mathsf{X}_{op}\cdot\,op(v) \longrightarrow e[x \mapsto v,\ resume \mapsto \lambda y.\,\mathsf{handle}_h\cdot\,\mathsf{X}_{op}\cdot\,y]$$

where $(op(x) \to e)\ \in h$. For this rule, we can view "$\mathsf{handle}_h\,.\,\mathsf{X}_{op}$" as our current execution context, i.e. as a *stack* and instruction pointer. In C, the execution context is represented by the current call stack and the current register context, including the instruction pointer. That means:

1. When we enter a handler, push a handle_h frame on the stack.
2. When we encounter an operation $op(v)$, walk down the call stack "$\mathsf{E}\cdot \mathsf{handle}_h \cdot \mathsf{X}_{op}$" until we find a handler for our operation.
3. Capture the current execution context "$\mathsf{handle}_h \cdot \mathsf{X}_{op}$" (call stack and registers) into a `resume` structure.
4. Jump to the handler h (restoring its execution context), and pass it the operation op, the argument v, and the captured resumption.

In the rest of this article, we assume that a stack always grows up with any parent frames "below" the child frames. In practice though, most platforms have downward growing stacks and the library adapts dynamically to that.

4.1. Entering a Handler

When we enter a handler, we need to push a handler frame on the stack. Effect handler frames are defined as:

```
typedef struct _handler {
  jmp_buf           entry;       // used to jump back to a handler
  const handlerdef* hdef;        // operation definitions
  volatile value    arg;         // the operation argument is passed here
  const operation*  arg_op;      // the yielded operation is passed here
  resume*           arg_resume;  // the resumption function
  void*             stackbase;   // stack frame address of the handler function
} handler;
```

Each handler needs to keep track of its `stackbase` – when an operation captures its resumption, it only needs to save the stack up to its handler's `stackbase`. The `handle` function starts by recording the `stackbase`:

```
value handle( const handlerdef* hdef,
              value (*action)(value), value arg ) {
  void* base = NULL;
  return handle_upto( hdef, &base, action, arg );
}
```

The stack base is found by taking the address of the local variable `base` itself; this is a good conservative estimate of an address just below the frame of the handler. We mark `handle_upto` as `noinline` to ensure it gets its own stack frame just above `base`:

```
noinline value handle_upto( hdef, base, action, arg ) {
  handler* h = hstack_push();
  h->hdef = hdef;
  h->stackbase = base;
  value res;
  if (setjmp(h->entry) == 0) {
    // (H1): we recorded our register context
    ...
 }
  else {
    // (H2): we long jumped here from an operation
    ...
  }
  // (H3): returning from the handler
  return res; }
```

This function pushes first a fresh handler on a *shadow* handler stack. In principle, we could have used the C stack to "push" our handlers simply by declaring it as a local variable. However, as we will see later, it is more convenient to maintain a separate shadow stack of handlers which is simply a thread-local array of handlers. Next the handler uses setjmp to save its execution context in h->entry. This is used later by an operation to longjmp back to the handler. On its invocation, setjmp returns always 0 and the (H1) block is executed next. When it is long jumped to, the (H2) block will execute.

For our purposes, we need a standard C compliant setjmp/longjmp implementation; namely one that just saves all the necessary registers and flags in setjmp, and restores them all again in longjmp. Since that includes the stack pointer and instruction pointer, longjmp will effectively "jump" back to where setjmp was called with the registers restored. Unfortunately, we sometimes need to resort to our own assembly implementations on some platforms. For example, the Microsoft Visual C++ compiler (msvc) will unwind the stack on a longjmp to invoke destructors and finalizers for C++ code [33]. On other platforms, not always all register context is saved correctly for floating point registers. We have seen this in library code for the ARM Cortex-M for example. Fortunately, a compliant implementation of these routines is straightforward as they just move registers to and from the entry block. Appendix A.1 shows an example of the assembly code for setjmp on 32-bit x86.

4.1.1. Handling Return

The (H1) block in handle_upto is executed when setjmp finished saving the register context. It starts by calling the action with its argument:

```
  if (setjmp(h->entry) == 0) {
    // (H1): we recorded our register context
    res = action(arg);
    hstack_pop();             // pop our handler
    res = hdef->retfun(res);  // invoke the return handler
  }
```

If the action returns normally, we are in the (*return*) rule:

$$\mathsf{handle}_h \cdot v \longrightarrow e[x \mapsto v] \quad \text{with}\,(\mathsf{return} \to e) \in h$$

We have a handler h on the handler stack, and the result value v in res. To proceed, we call the return handler function retfun (i.e. e) with the argument res (i.e. $x \mapsto v$) – but only after popping the handle_h frame.

4.1.2. Handling an Operation The `(H2)` block of `handle_upto` executes when an operation long jumps back to our handler `entry`:

```
else {
  // we long jumped here from an operation
  value   arg = h->arg;         // load our parameters
  const operation* op = h->arg_op;
  resume* resume = h->arg_resume;
  hstack_pop();                 // pop our handler
  res = op->opfun(resume,arg);  // and call the operation
}
```

This is one part of the (*handle*) rule:

$$\mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot op(v) \longrightarrow e[x \mapsto v,\ resume \mapsto \lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y]$$

where $(op(x) \to e) \in h$. At this point, the yielding operation just jumped back and the X_{op} part of the stack has been "popped" by the long jump. Moreover, the yielding operation has already captured the resumption *resume* and stored it in the handler frame `arg_resume` field together with the argument v in `arg` (Section 4.2). We store them in local variables, pop the handler frame handle_h, and execute the operation handler function e, namely `op->opfun`, passing the resumption and the argument.

4.2. Yielding an Operation

Calling an operation $op(v)$ is done by a function call `yield(OPTAG(op),`v`)`:

```
value yield(const optag* optag, value arg) {
  const operation* op;
  handler* h = hstack_find(optag,&op);
  if (op->opkind==OP_NORESUME)  yield_to_handler(h,op,arg,NULL);
                        else  return capture_resume_yield(h,op,arg); }
```

First we call `hstack_find(optag,&op)` to find the first handler on the handler stack that can handle `optag`. It returns the a pointer to the handler frame and a pointer to the operation description in `&op`. Next we make our first optimization: if the operation handler does not need a resumption, i.e. `op->opkind==OP_NORESUME`, we can pass `NULL` for the resumption and not bother capturing the execution context. In that case we immediately call `yield_to_handler` with a `NULL` argument for the resumption. Otherwise, we capture the resumption first using `capture_resume_yield`. The `yield_to_handler` function just long jumps back to the handler:

```
noreturn void yield_to_handler( handler* h, const operation* op,
                                value oparg, resume* resume ) {
  hstack_pop_upto(h);      // pop handler frames up to `h`
  h->arg = oparg;          // pass the arguments in then handler fields
  h->arg_op = op;  h->arg_resume = resume;
  longjmp(h->entry,1); }   // and jump back down! (to (H2))
```

4.2.1. Capturing a Resumption At this point we have a working implementation of effect handlers without *resume* and basically implemented custom exception handling. The real power comes from having first-class resumptions though. In the (*handle*) rule, the resumption is captured as:

$resume \mapsto \lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y$

This means we need to capture the current execution context, "$\mathsf{handle}_h \cdot Xop$", so we can later resume in the context with a result y. The execution context in C would be the stack up to the handler together with the registers. This is done by `capture_resume_yield`:

```
value capture_resume_yield(handler* h, const operation* op, oparg ) {
  resume* r = (resume*)malloc(sizeof(resume));
  r->refcount = 1;  r->arg = lh_value_null;
  // set a jump point for resuming
  if (setjmp(r->entry) == 0) {
    // (Y1) we recorded the register context in `r->entry`
    void* top = get_stack_top();
    capture_cstack(&r->cstack, h->stackbase, top);
    capture_hstack(&r->hstack, h);
    yield_to_handler(h, op, oparg, r); } // back to (H2)
  else {
    // (Y2) we are resumed (and long jumped here from (R1))
    value res = r->arg;
    resume_release(r);
    return res;
  } }
```

A resumption structure is allocated first; it is defined as:

```
typedef struct _resume {
  ptrdiff_t refcount; // resumptions are heap allocated
  jmp_buf   entry;    // jump point where the resume was captured
  cstack    cstack;   // captured call stack
  hstack    hstack;   // captured handler stack
  value     arg;      // the argument to `resume` is passed through `arg`.
} resume;
```

Once allocated, we initialize its reference count to `1` and record the current register context in its `entry`. We then proceed to the `(Y1)` block to capture the current call stack and handler stack.
These structures are defined as:

```
typedef struct _hstack {
  ptrdiff_t count;    // number of valid handlers in hframes
  ptrdiff_t size;     // total entries available
  handler*  hframes;  // array of handlers (0 is bottom frame)
} hstack;

typedef struct _cstack {
  void*     base;     // The base of the stack part
  ptrdiff_t size;     // The byte size of the captured stack
  byte*     frames    // The captured stack data (allocated in the heap)
} cstack;
```

Capturing the handler stack is easy and `capture_hstack(&r->hstack,h)` just copies all handlers up to and including `h` into `r`'s `hstack` field (allocating as necessary). Capturing the C call stack is a bit more subtle. To determine the current top of the stack, we cannot use our earlier trick of just taking the address of a local variable, for example as:

```
void* top = (void*)&top;
```

since that may underestimate the actual stack used: the compiler may have put some temporaries above the `top` variable, and ABI's like the System V `amd64` include a *red zone* which is a part of the stack above the stack pointer where the compiler can freely spill registers [32,3.2.2]. Instead, we call a child function that captures its stack top instead as a guaranteed conservative estimate:

```
noinline void* get_stack_top() {
  void* top = (void*)&top;
  return top; }
```

The above code is often found on the internet as a way to get the stack top address but it is wrong in general; an optimizing compiler may detect that the address of a local is returned which is undefined behavior in C. This allows it to return any value! In particular, both `clang` and `gcc` always return 0 with optimizations enabled. The trick is to use a separate identity function to pass back the local stack address:

```
noinline void* stack_addr( void* p ) {
  return p;
}
noinline void* get_stack_top() {
  void* top = NULL;
  return stack_addr(&top);
}
```

This code works as expected even with aggressive optimizations enabled.

The piece of stack that needs to be captured is exactly between the lower estimate of the handler `stackbase` up to the upper estimate of our stack `top`. The `capture_cstack(&r->cstack,h->stackbase,top)` allocates a `cstack` and `memcpy`'s into that from the C stack.

At this point the resumption structure is fully initialized and captures the delimited execution context. We can now use the previous `yield_to_handler` to jump back to the handler with the operation, its argument, and a first-class `resume` structure.

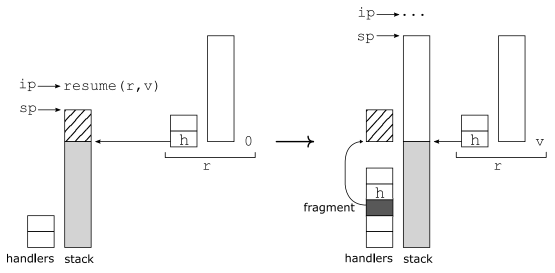

Figure 3. Resuming a resumption `r` that captured the stack up to a handler `h`. The captured stack will overwrite the striped part of the existing stack, which is saved by a fragment handler. The argument `v` is passed in the `arg` field of the resumption `r`.

4.3. Resuming

Now that we can capture a resumption, we can define how to resume one. In our operational semantics, a resumption is just a lambda expression:

$$\mathit{resume} \mapsto \lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y$$

and resuming is just application, $\mathsf{E} \cdot \mathit{resume}(v) \longrightarrow \mathsf{E} \cdot \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot v$. For the C implementation, this means pushing the captured stack onto the main stacks and passing the argument v in the `arg` field of the resumption. Unfortunately, we cannot just push our captured stack on the regular call stack. In C, often local variables *on the stack* are passed by reference to child functions. For example,

```
char buf[N];  snprintf(buf,N,"address of buf: %p", buf);
```

Suppose inside `snprintf` we call an operation that captures the stack. If we resume and restore the stack at a different starting location, then all those stack relative addresses are wrong! In the example, `buf` is now at a different location in the stack, but the address passed to `snprintf` is still the same.

Therefore, we must *always restore a stack at the exact same location*, and we need to do extra work in the C implementation to maintain proper stacks. In particular, when jumping back to an operation (H2), the operation may call the resumption. At that point, restoring the original captured stack will need to overwrite part of the current stack of the operation handler!

4.3.1. Fragments This is situation is shown in Figure 3. It shows a resumption `r` that captured the stack up to a handler `h`. The arrow from `h` points to the `stackbase` which is below the current stack pointer. Upon restoring the saved stack in `r`, the striped part of the stack is overwritten. This means:

1. We first save that part of the stack in a `fragment` which saves the register context and part of a C stack.
2. We push a special `fragment` handler frame on the handler stack just below the newly restored handler `h`. When `h` returns, we can now restore the original part of the call stack from the `fragment`.

The implementation of resuming becomes:

```
value resume(resume* r, value arg) {
  fragment* f = (fragment*)malloc(sizeof(fragment));
  f->refcount = 1;  f->res = value_null;
  if (setjmp(f->entry) == 0) {
    // (R1) we saved our register context
    void* top = get_stack_top();
    capture_cstack(&f->cstack, cstack_bottom(&r->cstack), top);
    hstack_push_fragment(f);       // push the fragment frame
    hstack_push_frames(r->hstack); // push the handler frames
    r->arg = arg;                  // pass the argument to resume
    jumpto(r->cstack, r->entry); } // and jump (to (Y2))
  else {
    // (R2) we jumped back to our fragment from (H3).
    value res = f->res; // save the resume result to a local
    hstack_pop(hs);     // pop our fragment frame
    return res;         // and return the resume result
  } }
```

The `capture_cstack` saves the part of the current stack that can be overwritten into our fragment. Note that this may capture an "empty" stack if the stack happens to be below the part that is restored. This only happens with resumptions though that escape the scope of an operation handler (i.e. *non-scoped* resumptions). The `jumpto` function restores an execution context by restoring a C stack and register context. We discuss the implementation in the next section.

First, we need to supplement the handler function `handle_upto` to take `fragment` handler frames into account. In particular, every handler checks whether it has a fragment frame below it: if so, it was part of a resumption and we need to restore the original part of the call stack saved in the fragment. We add the following code to `(H3)`:

```
noinline value handle_upto( hdef, base, action, arg ) {
  ...
  // (H3): returning from the handler
  fragment* f = hstack_try_pop_fragment();
  if (f != NULL) {
    f->res = res;                // pass the result
    jumpto(f->cstack,f->entry); // and restore the fragment (to (R2))
  }
  return res; }
```

Here we use the same `jumpto` function to restore the execution context. Unwinding through fragments also needs to be done with care to restore the stack correctly; we discuss this in detail in Appendix A.2.

4.3.2. Jumpto: Restoring an Execution Context The `jumpto` function takes a C stack and register context and restores the C stack at the original location and long jumps. We cannot implement this directly though as:

Compiler	Native (s)	Effects (s)	Slowdown	Operation Cost	Ops/s
`msvc` 2015 `/O2`	0.00057	0.1852	326×	162 · `sqrt`	$1.158 \cdot 10^6$
`clang` 3.8.0 `-O3`	0.00056	0.1565	279×	139 · `sqrt`	$1.402 \cdot 10^6$
`gcc` 5.4.0 `-O3`	0.00056	0.1883	336×	167 · `sqrt`	$1.193 \cdot 10^6$

Figure 4. Performance using full resumptions. All measurements are on a 2016 Surface Book with an Intel Core i7-6600U at 2.6GHz with 8GB ram (LPDDR3-1866) using 64-bit Windows 10 & Ubuntu 16.04. The benchmark ran for 100000 iterations. The *Native* version is a plain C loop, while the *Effect* version uses effect handlers to implement state. *Operation cost* is the approximate cost of an effect operation relative to a double precision *sqrt* instruction. *Ops/s* are effect operations per second performed without doing any work.

```
noreturn void jumpto( cstack* cstack, jmp_buf* entry ) {
  // wrong!
  memcpy(c->base,c->frames,c->size);  // restore the stack
  longjmp(*entry,1); }                // restore the registers
```

In particular, the `memcpy` may well overwrite the current stack frame of `jumpto`, including the `entry` variable! Moreover, some platforms use a `longjmp` implementation that aborts if we try to jump up the stack [14].

The trick is to do `jumpto` in two parts: first we reserve in `jumpto` enough stack space to contain the stack we are going to restore and a bit more. Then we call a helper function `_jumpto` to actually restore the context. This function is now guaranteed to have a proper stack frame that will not be overwritten:

```
noreturn noinline
void _jumpto( byte* space, cstack* cstack, jmp_buf* entry ) {
  space[0] = 0;                       // make sure is live
  memcpy(c->base,c->frames,c->size);  // restore the stack
  longjmp(*entry,1);                  // restore the registers
}
noreturn void jumpto(cstack* cstack, jmp_buf* entry ) {
  void*     top   = get_stack_top();
  ptrdiff_t extra = top - cstack_top(cstack);
  extra += 128;                 // safety margin
  byte* space = alloca(extra);  // reserve enough stack space
  _jumpto(space,cstack,entry); }
```

As before, for clarity we left out error checking and assume the stack grows up and `extra` is always positive. By using `alloca` we reserve enough stack space to restore the `cstack` safely. We pass the `space` parameter and write to it to prevent optimizing compilers to optimize it away as an unused variable.

4.4. Performance

To measure the performance of operations in isolation, we use a simple loop that calls a `work` function. The native C version is:

```
int counter_native(int i) {
  int sum = 0;
  while (i > 0) { sum += work(i); i--; }
  return sum; }
```

The effectful version mirrors this but uses a *state* effect to implement the counter, performing two effect operations per loop iteration:

```
int counter() {
  int i; int sum = 0;
  while ((i = state_get()) > 0) {
    sum += work(i);
    state_put(i - 1);  }
  return sum; }
```

The `work` function is there to measure the relative performance; the native C loop is almost "free" on a modern processors as it does almost nothing with a loop variable in a register. The work function performs a double precision square root:

```
noinline int work(int i) {  return (int)(sqrt((double)i)); }
```

This gives us a baseline to compare how expensive effect operations are compared to the cost of a square root instruction.

Figure 4 shows the results of running 100,000 iteration on a 64-bit platform. The effectful version is around 300× times slower, and we can execute about 1.3 million of effect operations per second.

The reason for the somewhat slow execution is that we capture many resumptions and fragments, moving a lot of memory and putting pressure on the allocator. There are various ways to optimize this. First of all, we almost never need a *first-class* resumption that can escape the scope of the operation. For example, if we use a `OP_NORESUME` operation that never resumes, we need to capture no state and the operation can be as cheap as a `longjmp`.

Another really important optimization opportunity is *tail resumptions*: these are resumes in a tail-call position in the operation handler. In the benchmark, each `resume` remembers its continuation in a fragment so it can return execution there – just to return directly without doing any more work! This leads to an ever growing handler stack with fragment frames on it. It turns out that in practice, almost *all* operation implementations use `resume` in a tail-call position. And fortunately, we can optimize this case very nicely giving orders of magnitude improvement as discussed in the next section.

5. Optimized Tail Resumptions

In this section we expand on the earlier observation that tail resumptions can be implemented more efficiently. We consider in particular a operation handler

Evaluation contexts:

$$\begin{array}{lll} \mathsf{F} & ::= & [] \mid \mathsf{F}(e) \mid v(\mathsf{F}) \mid op(\mathsf{F}) \mid \mathsf{val}\ x = \mathsf{F};\ e \mid \mathsf{handle}_h(\mathsf{F}) \mid \mathsf{yield}_{op}(\mathsf{F}) \\ & & \\ \mathsf{Y}_{op} & ::= & [] \mid \mathsf{Y}_{op}(e) \mid v(\mathsf{Y}_{op}) \mid op(\mathsf{Y}_{op}) \mid \mathsf{val}\ x = \mathsf{Y}_{op};\ e \\ & \mid & \mathsf{handle}_h(\mathsf{Y}_{op}) \qquad\qquad\qquad\qquad \text{if } op \notin h \\ & \mid & \mathsf{handle}_h(\mathsf{Y}_{op'}[\mathsf{yield}_{op'}(\mathsf{Y}_{op})]) \qquad\quad \text{if } op' \in h \end{array}$$

New Reduction rules:

$$\begin{array}{llll} (\mathit{handle}) & \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot op(v) & \overset{-}{\longrightarrow} & e[x \mapsto v,\ \mathit{resume} \mapsto \lambda y.\, \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot y] \\ & & & \text{with } \ (op(x) \to e) \in h \\ & & & \\ (\mathit{thandle}) & \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot op(v) & \overset{-}{\longrightarrow} & \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot \mathsf{yield}_{op} \cdot \mathit{resume}(e)[x \mapsto v] \\ & & & \text{with } \ (op(x) \to \mathit{resume}(e)) \in h \\ & & & \qquad\ \ \mathit{resume} \notin \mathsf{fv}(e) \\ & & & \\ (\mathit{tail}) & \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot \mathsf{yield}_{op} \cdot \mathit{resume}(v) & \overset{-}{\longrightarrow} & \mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot v \quad \text{with } (op \in h) \end{array}$$

Figure 5. Optimized reduction rules with yield frames. Rules (δ), (β), (*let*), and (*return*) are the same as in Figure 2.

of the form $(op(x) \to \mathit{resume}(e)) \in h$ where $\mathit{resume} \notin \mathsf{fv}(e)$. In that case:

$$\begin{array}{l} \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot op(v) \longrightarrow \\ \mathit{resume}(e)[x \mapsto v,\ \mathit{resume} \mapsto \lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y] \longrightarrow \{\, \mathit{resume} \notin e \,\} \\ (\lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y)(e[x \mapsto v]) \longrightarrow^* \{\, e[x \mapsto v] \longrightarrow^* v' \,\} \\ (\lambda y.\, \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot y)(v') \longrightarrow \\ \mathsf{handle}_h \cdot \mathsf{X}_{op} \cdot v' \end{array}$$

Since we end up with the same stack, $\mathsf{handle}_h \cdot \mathsf{X}_{op}$, we do not need to capture and restore the context $\mathsf{handle}_{op} \cdot \mathsf{X}_{op}$ at all but can directly evaluate the operation expression e as if it was a regular function call! However, if we leave the stack in place, we need to take special precautions to ensure that any operations yielded in the evaluation of $e[x \mapsto v]$ are not handled by any handler in $\mathsf{handle}_h \cdot \mathsf{X}_{op}$.

5.1. A Tail Optimized Semantics

In order to evaluate such tail resumptive expressions under the stack $\mathsf{handle}_{op} \cdot \mathsf{X}_{op}$, but prevent yielded operations from being handled by handlers in that stack, we introduce a new *yield* frame $\mathsf{yield}_{op}(e)$. Intuitively, a piece of stack of the from $\mathsf{handle}_{op} \cdot \mathsf{X}_{op} \cdot \mathsf{yield}_{op}$ can be ignored – the yield_{op} specifies that any handlers up to h (where $op \in h$) should be skipped when looking for an operation handler.

This is made formal in Figure 5. We have a new evaluation context F that evaluates under the new yield expression, and we define a new handler context Y_{op} that is like X_{op} but now also skips over parts of the handler stack that are skipped by yield frames, i.e. it finds the innermost handler that is not skipped.

The reduction rules in Figure 5 use a new reduction arrow $\overset{-}{\longrightarrow}$ to signify that this reduction can contain yield_{op} frames. The first five rules are equivalent to the

Compiler	Native (s)	Effects (s)	Slowdown	Operation Cost	Ops/s
`msvc` 2015 `/O2`	0.059	0.197	3.3×	1.15 · `sqrt`	$134 \cdot 10^6$
`clang` 3.8.0 `-O3`	0.059	0.153	2.6×	0.79 · `sqrt`	$150 \cdot 10^6$
`gcc` 5.4.0 `-O3`	0.059	0.167	2.8×	0.90 · `sqrt`	$141 \cdot 10^6$

Figure 6. Performance using tail optimized resumptions. Same benchmark as in Figure 4 but with $10 \cdot 10^6$ iterations.

usual rules except that the the (*handle*) rule uses the Y_{op} context now instead of X_{op} to correctly select the innermost handler for *op* skipping any handlers that are part of a $\mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot \mathsf{yield}_{op}$ (with $op \in h$) sequence.

The essence of our optimization is in the (*thandle*) rule which applies when the *resume* operation is only used in the tail-position. In this case we can (1) keep the stack *as is*, just pushing a yield_{op} frame, and (2) we can skip capturing a resumption and binding *resume* since $resume \notin \mathsf{fv}(e)$. The “unbound” tail *resume* is now handled explicitly in the (*tail*) rule: it can just pop the yield_{op} frame and continue evaluation under the original stack.

5.1.1. Soundness We would like to preserve the original semantics with our new optimized rules: if we reduce using our new $\overset{-}{\longrightarrow}$ reduction, we should get the same result if we reduce using the original reduction rule $\longrightarrow$. To state this formally, we define a *ignore* function on expression, $\overline{e}$, and contexts $\overline{\mathsf{F}}$ and $\overline{\mathsf{Y}}$. This function removes any $\mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot \mathsf{yield}_{op}$ sub expressions where $op \in h$, effectively turning any of our extended expressions into an original one, and taking F to E, and Y_{op} to X_{op}. Using this function, we can define soundness as:

Theorem 1. (*Soundness*)
If $\mathsf{F} \cdot e \overset{-}{\longrightarrow} \mathsf{F} \cdot e'$ then $\overline{\mathsf{F}} \cdot \overline{e} \longrightarrow \overline{\mathsf{F}} \cdot \overline{e'}$.

The proof is given in Appendix A.3.

5.2. Implementing Tail Optimized Operations

The implementation of tail resumptions only requires a modification to yielding operations:

```
value yield(const optag* optag, value arg) {
  const operation* op;
  handler* h = hstack_find(optag,&op);
  if (op->opkind==OP_NORESUME)  yield_to_handler(h,op,arg,NULL);
  else if (op->opkind==OP_TAIL) {
    hstack_push_yield(h);               // push a yield frame
    value res = op->opfun(NULL,op,arg); // call operation directly
    hstack_pop_yield();                 // pop the yield again
    return res;
  }
  else return capture_resume_yield(h,op,arg); }
```

We simply add a new operation kind `OP_TAIL` that signifies that the operation is a tail resumption, i.e. the operation promises to end in a tail call to `tail_resume`. We then push a yield frame, and directly call the operation. It will return with the final result (as a `tail_resume`) and we can pop the yield frame and continue. We completely avoid capturing the stack and allocating memory. The `tail_resume` is now just an identity function:

```
value tail_resume(const resume* r, value arg) { return arg; }
```

In the real implementation, we do more error checking and also allow `OP_TAIL` operations to not resume at all (and behave like and `OP_NORESUME`). We also need to adjust the `hstack_find` and `hstack_pop_upto` functions to skip over handlers as designated by the `yield` frames.

5.3. Performance, again

With our new optimized implementation of tail-call resumptions, let's repeat our earlier *counter* benchmark of Section 4.4. Figure 6 shows the new results where we see three orders of magnitude improvements and we can perform up to 150 million (!) tail resuming operations per second with the `clang` compiler. That is quite good as that is only about 18 instruction cycles on our processor running at 2.6GHz.

6. What doesn't work?

Libraries for co-routines and threading in C are notorious for breaking common C idioms. We believe that the structured and scoped form of algebraic effects prevents many potential issues. Nevertheless, with stacks being copied, we make certain assumptions about the runtime:

- We assume that the C stack is contiguous and does not move. This is the case for all major platforms. For platforms that support "linked" stacks, we could even optimize our library more since we can then capture a piece of stack by reference instead of copying! The "not moving" assumption though means we cannot resume a resumption on another thread than where it was captured. Otherwise any C idioms work as expected and arguments can be passed by stack reference. Except..
- When calling `yield` and `(tail_)resume`, we cannot pass parameters by stack reference but must allocate them in the heap instead. We feel this is a reasonable restriction since it only applies to new code specifically written with algebraic effects. When running in debug mode the library checks for this.
- For resumes in the scope of a handler, we always restore the stack and fragments at the exact same location as the handler stack base. This way the stack is always valid and can be unwound by other tools like debuggers. This is not always the case for a first-class resumption that escapes the handler scope – in that case a resumption stack may restore into an arbitrary C stack, and the new C stack is (temporarily) only valid above the resume base. We

have not seen any problems with this though in practice with either `gdb` or Microsoft's debugger and profiler. Of course, in practice almost all effects use either tail resumptions or resumptions that stay in the scope of the handler. The only exception is really the `async` effect but that in that case we happen to still resume at the right spot since we always resume from the same event loop.

7. Related Work

This is the first library to implement algebraic effects and handlers for the C language, but many similar techniques have been used to implement co-routines [22,1.4.2] and cooperative threading [1, 4, 6, 7, 11]. In particular, stack copying/switching, and judicious use of `longjmp` and `setjmp` [14]. Many of these libraries have various drawbacks though and restrict various C idioms. For example, most co-routines libraries require a fixed C stack [9, 15, 25, 41], move stack locations on resumptions [31], or restrict to one-shot continuations [8].

We believe that is mostly a reflection that general co-routines and first-class continuations (`call/cc`) are *too* general – the simple typing and added structure of algebraic effects make them more "safe" by construction. As Andrej Bauer, co-creator of the Eff [3] language puts it as: *effects+handlers* are to *delimited continuations* as what *while* is to *goto* [21].

Recently, there are various implementations of algebraic effects, either embedded in Haskell [21, 44], or built into a language, like Eff [3], Links [18], Frank [30], and Koka [26]. Most closely related to this article is Multi-core OCaml [12, 13] which implements algebraic effect natively in the OCaml runtime system. The prevent copying the stack, it uses linked stacks in combination with explicit copying when resuming more than once.
Multi-core OCaml supports *default* handlers [12]: these are handlers defined at the outermost level that have an implicit `resume` over their result. These are very efficient and implemented just as a function call. Indeed, these are a special case of the tail-resumptive optimization shown in Section 5.1: the implicit `resume` guarantees that the resumption is in a tail-call position, while the outermost level ensures that the handler stack is always empty and thus does not need a yield_{op} frame specifically but can use a simple flag to prevent handling of other operations.

8. Conclusion

We are excited by this library to provide powerful new control abstractions in C. For the near future we plan in integrate this into a compiler backend for the P language [10], and to create a nice wrapper for `libuv`. As part of the P language backend, we are also working on a C++ interface to our library which requires special care to run destructors correctly.

References

[1] Martín Abadi, and Gordon D. Plotkin. "A Model of Cooperative Threads." *Logical Methods in Computer Science* 6 (4:2): 1–39. 2010. doi:10.2168/LMCS-6(4:2)2010.

[2] Steve Awodey. *Category Theory.* Oxford Logic Guides 46. Oxford university press. 2006.

[3] Andrej Bauer, and Matija Pretnar. "Programming with Algebraic Effects and Handlers." *J. Log. Algebr. Meth. Program.* 84 (1): 108–123. 2015. doi:10.1016/j.jlamp.2014.02.001.

[4] Dave Berry, Robin Milner, and David N. Turner. "A Semantics for ML Concurrency Primitives." In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 119–129. POPL'92. Albuquerque, New Mexico, USA. 1992. doi:10.1145/143165.143191.

[5] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. "Pause 'n' Play: Formalizing Asynchronous C#." In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 233–257. ECOOP'12. Beijing, China. 2012. doi:10.1007/978-3-642-31057-7_12.

[6] Gérard Boudol. "Fair Cooperative Multithreading." In *Concurrency Theory: 18th International Conference*, edited by Luís Caires and Vasco T. Vasconcelos, 272–286. CONCUR'07. Lisbon, Portugal. Sep. 2007. doi:10.1007/978-3-540-74407-8_19.

[7] Frédéric Boussinot. "FairThreads: Mixing Cooperative and Preemptive Threads in C." *Concurrent Computation: Practice and Experience* 18 (5): 445–469. Apr. 2006. doi:10.1002/cpe.v18:5.

[8] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. "Representing Control in the Presence of One-Shot Continuations." In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 99–107. PLDI'96. Philadelphia, Pennsylvania, USA. 1996. doi:10.1145/231379.231395.

[9] Russ Cox. "Libtask." 2005. `https://swtch.com/libtask`.

[10] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. "P: Safe Asynchronous Event-Driven Programming." In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 321–332. PLDI '13. Seattle, Washington, USA. 2013. doi:10.1145/2491956.2462184.

[11] Edsger W. Dijkstra. "The Origin of Concurrent Programming." In , edited by Per Brinch Hansen, 65–138, chapter Cooperating Sequential Processes. 2002.

[12] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. "Concurrent System Programming with Effect Handlers." In *Proceedings of the Symposium on Trends in Functional Programming.* TFP'17. May 2017.

[13] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop.* Sep. 2015.

[14] Ralf S. Engelschall. "Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation." In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 20–31. ATEC'00. San Diego, California. 2000.

[15] Tony Finch. "Coroutines in Less than 20 Lines of Standard C." `http://fanf.livejournal.com/105413.html`. Blog post.

[16] Agner Fog. "Calling Conventions for Different C++ Compilers and Operating Systems." Feb. 2010. `http://agner.org/optimize/calling_conventions.pdf`.

[17] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control.” In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*. ICFP’17. 2017. arXiv:1610.09161.

[18] Daniel Hillerström, and Sam Lindley. “Liberating Effects with Rows and Handlers.” In *Proceedings of the 1st International Workshop on Type-Driven Development*, 15–27. TyDe 2016. Nara, Japan. 2016. doi:10.1145/2976022.2976033.

[19] Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. “Continuation Passing Style for Effect Handlers.” In *Proceedings of the Second International Conference on Formal Structures for Computation and Deduction*. FSCD’17. Sep. 2017.

[20] Intel. “Itanium C++ ABI.” 1999. `https://itanium-cxx-abi.github.io/cxx-abi`.

[21] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in Action.” In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 145–158. ICFP ’13. ACM, New York, NY, USA. 2013. doi:10.1145/2500365.2500590.

[22] Donald Knuth. *The Art of Computer Programming*. Volume 1. Addison-Wesley.

[23] Peter J. Landin. *A Generalization of Jumps and Labels*. UNIVAC systems programming research. 1965.

[24] Peter J. Landin. “A Generalization of Jumps and Labels.” *Higher-Order and Symbolic Computation* 11 (2): 125–143. 1998. doi:10.1023/A:1010068630801. Reprint from [23].

[25] Mark Lehmann. “Libcoro.” 2006. `http://software.schmorp.de/pkg/libcoro.html`.

[26] Daan Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects.” In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

[27] Daan Leijen. *Structured Asynchrony Using Algebraic Effects*. MSR-TR-2017-21. Microsoft Research. May 2017.

[28] “Libhandler.” 2017. `https://github.com/koka-lang/libhandler`.

[29] “Libuv.” `https://github.com/libuv/libuv`.

[30] Sam Lindley, Connor McBride, and Craig McLaughlin. “Do Be Do Be Do.” In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*, 500–514. Paris, France. Jan. 2017. doi:10.1145/3009837.3009897.

[31] Sandro Magi. “Libconcurrency.” 2008. `https://code.google.com/archive/p/libconcurrency`.

[32] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. “System V Application Binary Interface: AMD64 Architecture Processor Supplement.” Apr. 2017. `http://chamilo2.grenet.fr/inp/courses/ENSIMAG3MM1LDB/document/doc_abi_ia64.pdf`.

[33] MSDN. “Using Setjmp and Longjmp.” 2017. `https://docs.microsoft.com/en-us/cpp/cpp/using-setjmp-longjmp`

[34] Matt Pietrek. “A Crash Course on the Depths of Win32 Structured Exception Handling.” *Microsoft Systems Journal (MSJ)* 12 (1). Jan. 1997.

[35] Gordon D. Plotkin, and John Power. “Algebraic Operations and Generic Effects.” *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

[36] Gordon D. Plotkin, and Matija Pretnar. “Handlers of Algebraic Effects.” In *18th European Symposium on Programming Languages and Systems*, 80–94. ESOP’09. York, UK. Mar. 2009. doi:10.1007/978-3-642-00590-9_7.

[37] Gordon D. Plotkin, and Matija Pretnar. “Handling Algebraic Effects.” In *Logical Methods in Computer Science*, volume 9. 4. 2013. doi:10.2168/LMCS-9(4:23)2013.

[38] Matija Pretnar. “Inferring Algebraic Effects.” *Logical Methods in Computer Science* 10 (3). 2014. doi:10.2168/LMCS-10(3:21)2014.

[39] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley. 1994.

[40] Wouter Swierstra. "Data Types à La Carte." *Journal of Functional Programming* 18 (4): 423–436. Jul. 2008. doi:10.1017/S0956796808006758.

[41] Simon Tatham. "Coroutines in C." 2000. `https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`. Blog post.

[42] Hayo Thielecke. "Using a Continuation Twice and Its Implications for the Expressive Power of Call/CC." *Higher Order Symbolic Computation* 12 (1): 47–73. Apr. 1999. doi:10.1023/A:1010068800499.

[43] Stefan Tilkov, and Steve Vinoski. "NodeJS: Using JavaScript to Build High-Performance Network Programs." *IEEE Internet Computing*. 2010.

[44] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope." In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Göthenburg, Sweden. 2014. doi:10.1145/2633357.2633358.

A. Appendix

A.1. Implementing Setjmp and Longjmp

Here is some example of the assembly code for `setjmp` and `longjmp` for 32-bit x86 with the `cdecl` calling convention is [16]. The `setjmp` function just moves the registers into the `jmp_buf` structure:

```
; called with:
; [esp + 4]: _jmp_buf address (cleaned up by caller)
; [esp]    : return address
mov     ecx, [esp+4]     ; _jmp_buf to ecx
mov     [ecx+ 0], ebp    ; save registers
mov     [ecx+ 4], ebx
mov     [ecx+ 8], edi
mov     [ecx+12], esi
lea     eax, [esp+4]     ; save esp (minus return address)
mov     [ecx+16], eax
mov     eax, [esp]       ; save the return address (eip)
mov     [ecx+20], eax
stmxcsr [ecx+24]         ; save sse control word
fnstcw  [ecx+28]         ; save fpu control word
xor     eax, eax         ; return zero
ret
```

Note that we only need to save the *callee save* registers; all other temporary registers will have already been spilled by compiler when calling the `setjmp` function.

The `longjmp` function reloads the saved registers and in the end jumps directly to the stored instruction pointer (which was the return address of `setjmp`):

```
; called with:
; [esp+8]: argument
; [esp+4]: jmp_buf adress
; [esp]  : return address (unused!)
_lh_longjmp PROC
  mov     eax, [esp+8]        ; set eax to the return value (arg)
  mov     ecx, [esp+4]        ; ecx to jmp_buf
  mov     ebp, [ecx+ 0]       ; restore registers
  mov     ebx, [ecx+ 4]
  mov     edi, [ecx+ 8]
  mov     esi, [ecx+12]
  ldmxcsr [ecx+24]            ; load sse control word
  fnclex                      ; clear fpu exception flags
  fldcw   [ecx+28]            ; restore fpu control word
  test    eax, eax            ; longjmp should never return 0
  jnz     ok
  inc     eax
ok:
  mov     esp, [ecx+16]       ; restore esp
  jmp     dword ptr [ecx+20]  ; and jump to the eip
```

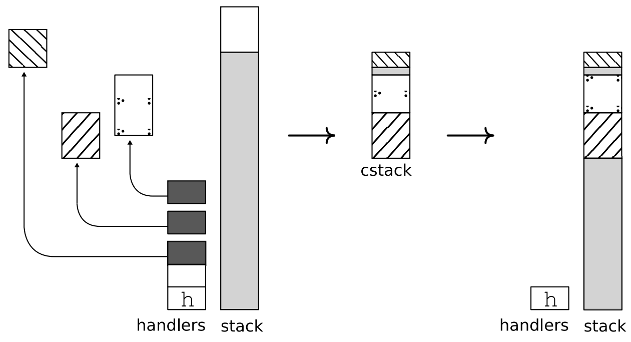

Figure 7. Unwinding a stack with fragments

Actually, it turns out that on 32-bit Windows we also need to extend the assembly to save and restore the exception handler frames that are linked on the stack (pointed to by the first field of the thread local storage block at `fs:[0]`). See Section A.4 for more information.

A.2. Unwinding through fragments

This section shows in more detail how to unwind the handler stack in the presence of fragments (as discussed in Section 4.3.1). The `hstack_pop_upto` function without fragments can just pop handlers, skipping over yield segments, until it reaches the handler `h`. In the presence of fragment handlers though, it also needs to restore the C stack as it was when the handler `h` was on top of the stack.

Figure 7 illustrates this. When unwinding to handler `h`, there are three fragment handlers on the stack. In the figure, the top of the gray part of the stack shows the `stackbase` for `h`. When unwinding the handler stack, a new composite fragment is created that applies each fragment on the handler stack in order. After unwinding, the C stack can be properly restored to its original state.

Note that when unwinding with fragments, you cannot skip over yield segments – all fragment frames up to the final handler must be taken into account to restore the C stack as it was when the handler was originally installed.

A.3. Proof of Soundness

Here we prove soundness of the tail resumptive optimization discussed in Section 5. Useful properties of the *ignore* function are $\overline{v} = v$, and $\overline{\mathsf{F}} \cdot \overline{\mathsf{handle}_h \cdot \mathsf{Y}_{op} \cdot \mathsf{yield}_{op}}$ (with $op \in h$) equals $\overline{\mathsf{F}}$.

Proof. (*Of Theorem 1, Section 5.1.1*) We show this by case analysis on reduction rules. The first five rules are equivalent to the original rules. For (*yhandle*) we have:

$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ op(v)}$$
$$\quad=$$
$$\overline{\mathsf{F}\cdot\ \mathsf{handle}_h\cdot\ \mathsf{Y}_{op}}\cdot\ op(v)$$
$$\quad=$$
$$\overline{\mathsf{F}}\cdot\ \mathsf{handle}_{\overline{h}}\cdot\ \overline{\mathsf{Y}}_{op}\cdot\ op(v)$$
$$\quad\longrightarrow$$
$$\overline{\mathsf{F}}\cdot\ \overline{e[x\mapsto v,\ resume\mapsto\lambda y.\,\mathsf{handle}_{\overline{h}}\cdot\ \overline{\mathsf{Y}}_{op}\cdot\ y]}$$
$$\quad=\ \{\ resume\notin\mathsf{fv}(e)\ \}$$
$$\overline{\mathsf{F}}\cdot\ \overline{e[x\mapsto v]}$$
$$\quad=\ \{\ op\in h\,\}$$
$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ \mathsf{yield}_{op}}\cdot\ \overline{e[x\mapsto v]}$$
$$\quad=$$
$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ \mathsf{yield}_{op}\cdot\ e[x\mapsto v]}$$

In the (*tail*) rule, we know by construction that in the original reduction rules, *resume* would have been bound as a regular $resume\mapsto\lambda y\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ y}$:

$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ \mathsf{yield}_{op}}\cdot\ resume(v)$$
$$\quad=$$
$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ \mathsf{yield}_{op}}\cdot\ (\lambda y\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ y})(v)$$
$$\quad=\ \{\ op\in h\,\}$$
$$\overline{\mathsf{F}}\cdot\ (\lambda y\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ y})(v)$$
$$\quad\longrightarrow$$
$$\overline{\mathsf{F}}\cdot\ \overline{\mathsf{handle}_h\cdot\ \mathsf{Y}_{op}\cdot\ v}$$
$$\quad=$$
$$\overline{\mathsf{F}\cdot\ \mathsf{handle}_h\cdot\ \mathsf{Y}_{op}}\cdot\ v$$

□

A.4. C++ and Exception Handling

Making the library work with C++ is not entirely straightforward. In particular, we need to adapt the library to deal with exceptions and destructors. Fortunately, there are just a few places that need to be adapted:

1. When pushing a handler frame, we ensure that the handler gets popped even in the presence of exceptions.
2. When an exception reaches a fragment frame, we need to restore the C stack and continue throwing the exception.
3. When a resumption is never resumed, we should ensure that all destructors captured by that resumption are still executed.

A.4.1. Popping Handler Frames To ensure that a pushed handler frame is always popped when exiting the scope, even in the presence of exceptions, we use the C++ RAII idiom [39, 16.5].

```
class raii_hstack_pop {
public:
  raii_hstack_pop()   { }
  ~raii_hstack_pop()  {  hstack_pop(); }
};
```

When pushing a handler frame, we also create a corresponding *pop* object in the scope whose destructor will automatically pop the handler frame when either exiting the scope normally, or through an exception. For example, we adjust the code for tail resumptions in Section 5.2 to pop the yield frame on exit:

```
value yield(const optag* optag, value arg) {
  const operation* op;
  handler* h = hstack_find(optag,&op);
  if (op->opkind==OP_NORESUME)  yield_to_handler(h,op,arg,NULL);
  else if (op->opkind==OP_TAIL) {
    hstack_push_yield(h);               // push a yield frame
    raii_hstack_pop pop_yield();
    return op->opfun(NULL,op,arg);      // call operation directly
  }
  else return capture_resume_yield(h,op,arg); }
```

A.4.2. Raising through Fragment Frames Things become more complex when an exception reaches a fragment frame as described in Section 4.3.1. In this case, the original C stack stored in the fragment first needs to be restored, and then the exception needs to be re-thrown given the new execution context. First, we extend the `fragment` structure to also hold a first-class exception:

```
typedef struct _fragment {
  count              ref_count;
  jmp_buf            entry;  // point where the fragment was captured
  struct _cstack     cstack; // the C stack to restore
  value              arg;    // when jumped to, the result is passed here
  std::exception_ptr eptr;   // non-null if an exception happened
} fragment;
```

When a fragment returns with a non-null `eptr` the exception is re-thrown in the new context. We adapt the code for resuming from Section 4.3.1 as:

```
value resume(resume* r, value arg) {
  fragment* f = (fragment*)malloc(sizeof(fragment));
  f->refcount = 1;
  f->res = value_null;
  f->eptr = NULL;
  if (setjmp(f->entry) == 0) {
    // (R1) we saved our register context
    ...
  }
  else {
    // (R2) we jumped back to our fragment from (H3).
    value res = f->res;       // save the resume result to a local
    std:exception_ptr eptr;   // swap the eptr to a local
    std::swap(eptr,f->eptr);
    hstack_pop(hs);           // pop our fragment frame
    if (eptr) {
      std::rethrow_exception(eptr); // rethrow if propagating an exception
    }
    return res;               // and return the resume result
  }
}
```

We use `std::swap` here to save the exception pointer locally in order to ensure we use it linearly such that it gets properly released when re-throwing the exception.

The code for handlers now needs to catch any exceptions in order to propagate them through fragments. The code of Section 4.1 is adapted to do this. First we call the helper `handle_uptox` instead of `handle_upto`:

```
value handle( const handlerdef* hdef,
              value (*action)(value), value arg ) {
  void* base = NULL;
  return handle_uptox( hdef, &base, action, arg );
}
```

where `handle_uptox` takes care of handling fragments even in the case of exceptions:

```
noinline value handle_uptox( const handlerdef* hdef, base, action arg ) {
  fragment* f;
  value res;
  try {
    res = handle_upto(hdef, base, action, arg);
    f = hstack_try_pop_fragment();
  }
  catch (...) {
    f = hstack_try_pop_fragment();
    if (fragment==NULL) throw;           // re-throw immediately if there is no fragment
    f->eptr = std::current_exception();  // remember to re-throw!
    res = lh_value_null;
  }
  // (H4): returning from the handler
  if (f != NULL) {
    f->res = res;                 // pass the result
    jumpto(f->cstack,f->entry);  // and restore the fragment (to (R2))
  }
  // otherwise just return normally
  return res;
}
```

We also moved the original code for fragments in the `handle_upto` function (Section 4.3.1, `(H3)`) to `handle_uptox` (as `(H4)`) so it is outside the exception handler. The `handle_upto` function now just returns the result directly:

```
noinline value handle_upto( hdef, base, action, arg ) {
  value res;
  ...
  // (H3): returning from the handler
  return res;
}
```

A.4.3. Calling Destructors Finally, we need to ensure that when an `OP_NORESUME` operation never resumes, that we still invoke all the destructors up to the handler. To properly unwind the stack we resort to throwing a special `unwind_exception` instead of long jumping directly. We adapt the implementation of `yield` (Section 5.2) to use exceptions:

```
value yield(const optag* optag, value arg) {
  const operation* op;
  handler* h = hstack_find(optag,&op);
  if (op->opkind==OP_NORESUMEX) {
    yield_to_handler(h,op,arg,NULL);
  }
  else if (op->opkind==OP_NORESUME {
    throw unwind_exception(h,op->opfun,arg);
  }
  ...
}
```

We added a new category `OP_NORESUMEX` which returns to a handler in the old way without calling destructors which can still be useful in specific cases. For the normal `OP_NORESUME` an unwind exception is thrown. The exception will propagate down the stack invoking destructors as usual. As shown in the previous section, unwinding also proceeds correctly through fragment handlers.

Eventually, the unwind exception is caught by operation handler. This is done by adapting the `(H1)` block in Section 4.1.1 for handling returns:

```
value handle_upto(hdef,base,action,arg) {
  value     res;
  handler*  h  = hstack_push(hdef,base);
  count     id = h->id;  // save id of the handler locally
  h->hdef      = hdef;
  h->stackbase = base;
  if (setjmp(h->entry) == 0) {
    // (H1): we recorded our register context
    try {
      raii_hstack_pop pop_handler();
      res = action(arg);
    }
    catch( const unwind_exception& e ) {
      if (e.handler->id != id) throw; // re-throw if not for us
      res = e.opfun(e.arg);           // call the operation
    }
    res = hdef->retfun(res);  // invoke the return handler
  }
  else {
    // (H2): we come back from a long jump
    ...
  }
  return res;
}
```

Note that we need to give handlers unique id's to identify them. We cannot directly compare the handler pointers since they may be moved when a handler stack is re-allocated. The identifier is used to see if an unwind exception is meant for us or another handler further down the stack. If we catch an unwind exception for our handler, we invoke the passed operation (which is an OP_NORESUME operation).

Besides OP_NORESUME operations, there is one other possibility to not resume: a first-class resumption (OP_GENERAL) can never be resumed! To detect this situation, we add a resumptions counter to a resume structure. When a resumption is released, and the resumptions counter is still zero, we need to invoke any destructors captured by the resumption:

```
void resume_release( resume* r ) {
  if (r->refcount == 1 && r->resumptions == 0) {
    try {
      r->resumptions = -1;    // so resume will raise an exception
      resume(r, value_null);  // resume it, with resumption set to -1
      assert(false);          // we should never get here
    }
    catch (const resume_unwind_exception& e) {
      if (e.resume != r) throw;  // re-throw if it is not our resumption
    }
  }
  ...
}
```

By setting the resumptions field to -1, the resume call will first restore the execution context as normal, but then raise a resume_unwind_exception. This is propagated as usual and finally caught in the catch block of resume_release. At that point we know all the captured destructors have been invoked and we can safely free the resumption.

A.4.4. Maintaining Exception Handler Chains Most C++ implementations use "zero cost" exception handling as initially specified by the Itanium C++ ABI [20]. This uses compile time tables to find the correct exception handlers and requires no special exception frames in the C stack. On some platforms though, and in particular 32-bit Windows, C++ exception handling is implemented us-

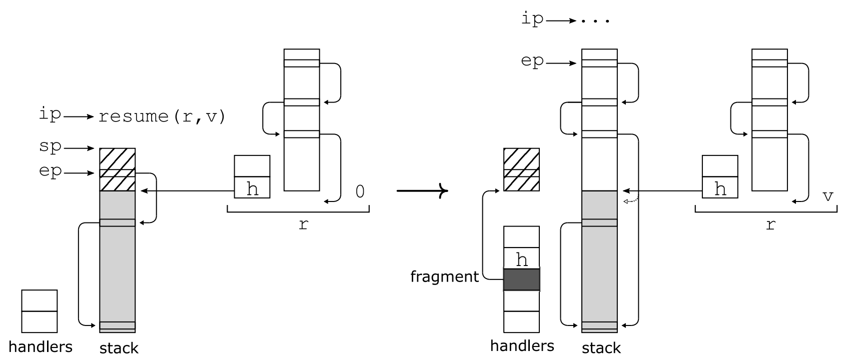

Figure 8. Resuming with linked exception handlers (adapted from Figure 3). The register `ep` points to the top of the exception handler list and is system dependent. On 32-bit Windows it is stored in the first field of the thread local storage block, i.e. `fs:[0]`.

ing a chain of exceptions handlers on the stack. On 32-bit Windows this is part of the operation system known as structured exception handling (SEH) [34].

When using resumptions in tail-call position, or when they do not escape the scope of the operation, any resumption is always restored at the same valid stack location. This ensures that any return addresses and also the exception handler frames always form a valid chain. However, this is not the case for a first-class resumption that is resumed in another arbitrary stack – since we always restore a stack at the same address, the resumed stack part is only valid up to that address.

This is in principle no problem: since we use fragment frames and a `try` block at any handler, we always restore the original stack before returning or re-throwing an exception at run-time. Nevertheless, this can be troublesome for tools like debuggers or profilers. For example, the Microsoft Visual Studio debugger always expects a valid chain of exception handlers when an exception is thrown and returns an error if the chain is broken across a resumption. Therefore, on platforms that require it (currently only 32-bit Windows), we also ensure we always have a valid exception handler chain when calling a resumption. This is shown in detail in Figure 8.

At initialization time we find the bottom exception handler frame which is usually put on the stack by the OS or the C runtime even before running `main`. Whenever a first-class resumption is entered, we find the bottom exception handler frame in the stack part of the resumption and patch it to point to the bottom-most exception frame. This way, there is always a valid exception handler chain on the stack. When an actual exception happens, we know that the bottom exception handler in the resumption stack is the `try` frame of the `handle` function which catches all exceptions – so at run-time the exception chain is never actually unwound beyond that point.

A.5. Resource Management and Handler Local State

Todo.