An implementation of lexically scoped effect handler

CONG (MC) MA, University of Waterloo

Algebraic effect handlers are powerful language features that allow programmer to define interesting control flows, including exception, generators and async-await. Recently, researchers have found that traditional interpretation of algebraic effect handlers violates the principle of modular reasoning, and proposes a new semantic called lexically scoped effect handler. This paper offers an implementation of the lexically scoped effect handler, and shows that due to the absense of stack-walking, this implementation achieves better performance than its counterpart.

ACM Reference Format:
Cong (MC) Ma. 2022. An implementation of lexically scoped effect handler. 1, 1 (December 2022), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

- In section 2, we illustrate the problem of accidentally handled effects, which violates the principle of modular reasoning.
- In section 3, we give static semantics and operational semantics of the lexically scoped handler.
- In Section 4, we show how we implement lexically scoped handler as a library in C
- In section 5, we evaluate our implementation against a baseline, and show that due to the absense of stack walking, the lexically scoped handler achieves better performance.

2 ALGEBRAIC EFFECTS AND ACCIDENTAL HANDLING

Traditional algebraic effects suffer from accidental handling. We illustrate what that means using the example below.

```
map[X,Y,E](l : List[X], f : X  Y throws E) : List[Y] throws E
```

The map function takes in a list and a lambda, and it runs the lambda on every element of the list. The map function is polymorphic over the input type X, output type Y and the effect E of the lambda. Since the effect E is abstract for map, the implementation of map should not behave differently according to what E is instantiated with at the runtime. However, consider the case where "map" is implemented using a "while" loop and it detects the end of the list by catching a "NoSuchElement" exception. If the effect parameter E is also instantiated with such exception, meaning that f can throw this exception inside map, according to the semantics of algebraic effect, this exception will be caught by the handler in map. This violates the abstraction of E, and makes it difficult to reason about the program due to the break of modularity.

Author's address: Cong (MC) Ma, University of Waterloo.

XXXX-XXXX/2022/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The core of the problem lies behind the dynamic nature of algebraic effect. The dynamic semantics of algebraic effect specifies that, when an effect is raised, the closest enclosing handler in the evaluation context is used to handle the effect. That's why the handler in map is used to handle the effect raised in f: the handler is the closest in the context of the effect raise site.

To solve the problem of accidental handling, previous work introduces "lexically scoped effect handler"[Zhang and Myers 2019]. In this semantics, references to the handlers are passed into functions as term variables, and "raise" statement require an explicit handler variable. In this setting, function argument to a higher-order function can be partially applied to a handler reference before entering to the high-order function, thus ensuring that the effect it raises are handled by the appropriate handler.

3 SYNTAX, STATIC SEMANTICS AND DYNAMIC SEMANTICS

In this section, we formally present the syntax, selected static semantics and dynamic semantics of lexically scoped effect handler.

Syntax

effects	e	$::=$	$\alpha \mid h$
handlers	h	$::=$	$\ell \mid \mathbb{L}$
types	τ, σ, θ	$::=$	$\mathbb{1} \mid \forall\overline{\alpha}.\forall\overline{\ell}:\overline{\mathbb{F}}.\,\overline{\tau} \longrightarrow [\sigma]_{\overline{e}} \mid$
			$\mathbf{cont}\ [\tau] \rightsquigarrow [\sigma]_{\overline{e}}$
values	v	$::=$	$() \mid \mathsf{x} \mid \mathbf{cont}\ K \mid$
			$\Lambda\overline{\alpha}.\,\Lambda\overline{\ell}.\,\lambda\overline{\mathsf{x}}.\,t$
terms	t, s	$::=$	$v \mid \mathbf{throw}\ (\mathbf{cont}\ K)\ v \mid$
			$\mathbf{raise}\, h\ [\overline{e}]\ [\overline{h}]\ \overline{s} \mid$
			$t\ [\overline{e}]\ [\overline{h}]\ \overline{s} \mid$
			$\mathbf{handle}_{\mathbb{L}}\ s\ \mathbf{with}_{\mathbb{F}}\ \Lambda\overline{\alpha}.\,\Lambda\overline{\ell}.\,\lambda\overline{\mathsf{x}}.\,\lambda\mathsf{k}.\,t$

effect names	$\mathbb{F}$	*effect variables*	α
handler names	$\mathbb{L}$	*handler variables*	ℓ
term variables	$\mathsf{x}, \mathsf{y}, \mathsf{k}, \ldots$		

Selected Dynamic Semantics

$$[\textsc{raise}] \quad \mathbf{handle}_{\mathbb{L}}\ K[\mathbf{raise}\ \mathbb{L}\ [\overline{e}]\ [\overline{h}]\ \overline{s}]\ \mathbf{with}_{\mathbb{F}}\ t \longrightarrow t\ [\overline{e}]\ [\overline{h}]\ \overline{s}\,(\mathbf{cont}\ K)$$

$$[\textsc{throw}] \quad \mathbf{throw}\ (\mathbf{cont}\ K)\ v \longrightarrow K[v]$$

Selected Static Semantics

$$[\textsc{tm-h}]\quad \frac{\begin{array}{c}\Delta \mid \mathrm{P},\ \mathbb{L}:\mathbb{F} \mid \Gamma \ \vdash s:[\sigma]_{\overline{e_2},\,\mathbb{L}} \\ \Delta,\ \overline{\alpha} \mid \mathrm{P},\ \overline{\ell}:\overline{\mathbb{F}} \mid \Gamma,\ \overline{\mathsf{x}}:\overline{\tau_2},\ \mathsf{k}:\textbf{cont}\ [\tau]_{\overline{e_1}} \rightsquigarrow [\sigma]_{\overline{e_2}} \ \vdash t:[\sigma]_{\overline{e_2}}\end{array}}{\Delta \mid \mathrm{P} \mid \Gamma \ \vdash \textbf{handle}_{\mathbb{L}}\ s\ \textbf{with}_{\mathbb{F}}\ \Lambda\overline{\alpha}.\,\Lambda\overline{\ell}.\,\lambda\overline{\mathsf{x}}.\,\lambda\mathsf{k}.\,t:[\sigma]_{\overline{e_2}}}$$

$$[\textsc{tm-raise}]\quad \frac{\begin{array}{c}\Delta \mid \mathrm{P} \vdash \overline{e} \qquad \mathrm{P} \vdash \overline{h_2} \qquad \mathrm{P}(h) = \mathbb{F} \\ op(\mathbb{F}) = \forall\overline{\alpha}.\forall\overline{\ell}:\overline{\mathbb{F}}.\,\overline{\tau} \longrightarrow [\sigma]_{\overline{e_2}} \\ \Delta \mid \mathrm{P} \mid \Gamma \ \vdash \overline{s_i}:[\overline{\tau_i}]_{\overline{e}_2}\end{array}}{\Delta \mid \mathrm{P} \mid \Gamma \ \vdash \textbf{raise}\, h\ [\overline{e}]\ [\overline{h_2}]\ \overline{s}:[\sigma]_{h,\overline{e}_2}}$$

$$[\textsc{tm-throw}]\quad \frac{\begin{array}{c}\Delta \mid \mathrm{P} \mid \Gamma \ \vdash t:\big[\textbf{cont}\ [\tau]_{\overline{e_1}} \rightsquigarrow [\sigma]_{\overline{e_2}}\big]_{\overline{e_2}} \\ \Delta \mid \mathrm{P} \mid \Gamma \ \vdash s:[\tau]_{\overline{e_1}}\end{array}}{\Delta \mid \mathrm{P} \mid \Gamma \ \vdash \textbf{throw}\ t\ s:[\sigma]_{\overline{e_2}}}$$

In this language, every handler is assigned a unique name $\mathbb{L}$, and is introduced into the scope by the "handle" term. The handlers can be passed down to the functions that raise it. When a handler is eventually raised(see rule "RAISE"), the evaluation context between the raise site and the corresponding handle term is reified into a continuation "K", and the handler is invoked with the arguments and "K". The continuation K can later be thrown(see rule "THROW").

Notice that if a handler is raised when the corresponding handle term is not in the context, the evaluation is stuck. This can happen when the handler escapes the function through a closure. To prevent this from happening, the language treats handler lifetime as computational effects. A computation that raises a handler is typed with the lifetime effect of the handler. These effects are syntactically marked as subscripts to types. For example, a function type is annotated with the effect that it can raise, and a continuation is similarly annotated with the effects that the continuation can raise.

With the lifetime effects, the static semantics ensures that handler is always raised within the lifetime of the corresponding handle term(see rule "TM-RAISE"). Previous literature has proved that such language is sound, meaning that a well-typed program will not get stuck. In addition, this semantics gives programmer finer control over the runtime behavior of effects handling, and it has been shown that programmer can use it to avoid the problem of accidental handling.

4 IMPLEMENTATION

Our implementation aims for efficiency, therefore we don't consider CPS based translation[Schuster et al. 2022] as that will create a lot of closure on the heap. Instead, we choose to extend C with lexically scoped handler. Intuitively, C's stack can be treated as continuation. Each activation record contains the data and the return address needed to resume the computation. To be able to treat continuation as first class construct, stack frames can be copied into heap and reinstate onto the stack later. This captures the reification of the continuation.

4.1 Challenge with first-class continuation in C

However, treating stack frames as continuation brings up a unique challenge in C. C allows program to have pointers pointing into stack frame, so to keep the pointer valid, when reinstating the stack frames, the stack frames need to be at its original locations. However, at the time when the continuation is resumed, the original place may be occupied by other stack frames, and care needs to be taken to avoid overwriting these frames.

$$\textbf{let}\ \mathsf{x}\ =\ \textbf{handle}_{\mathbb{L}}\ \textbf{raise}\ \mathbb{L};\ 42\ \textbf{with}_{\mathbb{F}}\ \lambda\mathsf{k}.\,\mathsf{k}\ \textbf{in}$$
$$(\lambda\mathsf{c}.\,\textbf{throw}\ \mathsf{c}\ ())x$$

Fig. 1. Extra stack frames

The example in Figure 1 illustrates the issue. An effect $\mathbb{L}$ is raised and the continuation is reified and stored into x. x is then passed down into a function, which resume the continuation. Notice that when the continuation is resumed, the stack now contains one more stack frame that is created by $\lambda c. ...$, which will be overwrite by the continuation if it's simply put into its original place.

4.2 Shadow stack and segmented stack

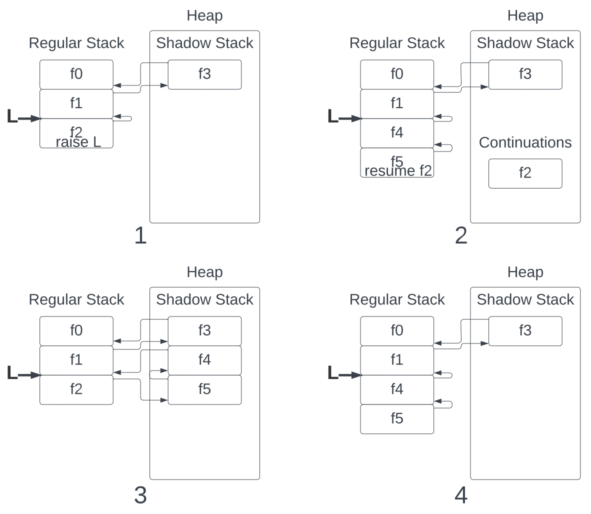

Fig. 2. Shadow stack

[Leijen 2017] tackles the problem by managing the extra stack frames on a "shadow stack". The idea is that, when reinstating the continuation, if there are extra stack frames that will be overwritten, they are copied onto the "shadow stack". When the continuation finishes, the fragment will be copied back to the stack and continue its execution.

Figure 2 illustrates the idea of shadow stack with a program that returns and later resumes a continuation. The figure shows four states of stack and heap during the execution. The thin arrows indicate the control flow. The thick arrow indicates the stack location where the handler is installed; this serves as the delimiter up to which point the continuation is reified. The first state shows what is right before when the effect is raised. The stack frame of f2 is reified into a continuation, and is stored on the heap, as is shown on the second state. In this example, the handler invokes a f4, which later invokes f5, and then resumes the continuation. Now, we need to put the stack frames of the continuation into its original place, so we relocate the stack frames for f4 and f5 onto the shadow stack, and this is shown in the third state. As the continuation finishes execution, the control goes back to the handler, so the fragment f4 and f5 are copied back to the stack, and the execution continues. This is shown in the forth state.

The advantage of this design is that the algeberaic effect is offered as a zero-cost abstraction, meaning that if the program doesn't use algeberaic effect, this design doesn't incur any runtime overhead. However, when algeberaic effect is heavily used, this design incur a lot of copying, which is very expensive.

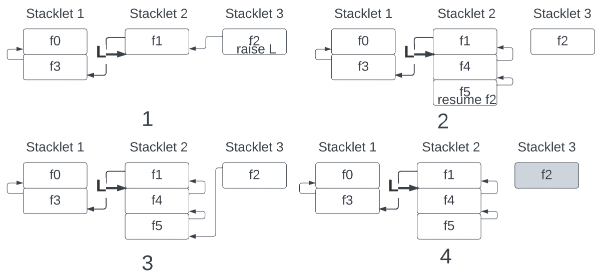

Fig. 3. Segmented stack

An alternative design, adopted by [Ghica et al. 2022; Sivaramakrishnan et al. 2021], uses segmented stacks and achieve better performance by avoid excessive copying. In this setting, when entering the scope of a handler, a new stack is created and is linked with the previous one. Now, returning a continuation is achieved by returning the handle of the stack, which is a constant operation. And resuming a continuation is achieved by linking the stack.

This design also prevents the problem in Figure 1, where the parent context grows the stack and collides with the continuation that will be resumed. Since each handler has its own stack, the parent handler can grows its own stack without impacting the inner context.

Figure 3 illustrates the behavior of segmented stacks using a program that's the same as in Figure 2. The state begins with three stacklets, and the function f2 lives in the stacklet 3. When an effect is raised, the entire stacklet is reified into a continuation, and the handler's frames run in the stacklet 2, as is shown in the State 2. When the handler resumes the continuation, a simple pointer assignment is needed as is shown in State 3. Finally, when the f2 finishes execution, the entire stacklet is garbage collected, as is shown in State 4.

4.3 Absence of stack walking in lexically scoped handler

The previous section discuss the implementaiton of delimited continuation used in algebraic effect. To implement algebraic effect fully, the compiled program should be able to find the closest enclosing handler of the appropriate type(this behavior is specified in the operational semantics of traditional algebraic effect, which is not discussed in this paper). Most of the previous work implement this behavior by using a linked list to chain the handlers, and walks the list at the runtime to find the appropriate handler. This obviously incurs linear cost.

In contrast, the lexically scoped handler, which is the main concern of this paper, does not need stack walking. At compile time, the definition of the handlers are passed down the call chain, so the raise statement has a direct reference to the handler, and invocation has constant cost.

4.4 Implementation details

We implement lexically scoped handler on top of a C library libmprompt[Leijen 2021]. The library provides the capability of doing delimited continuation, and it has been previously used to implement traditional algebraic effect in C. In the next section, we provide an evaluation by running the benchmark programs written in lexically scoped handler and traditional algebraic effect.

Our implementation adopt a simple optimization: when a handler is tail resumptive or abortive, we do not allocate a new stacklet. This optimization uses an observation that tail resumptive or abortive handlers do not need to use the continuation, so we do not need to prepare a separate stacklet in anticipation of creating a continuation. Admittedly this optimization can also be adopted in the baseline, but we have not implement it.

The structure of the repository is explain below

- *include/mprompt.h* header file of libmprompt
- *src/mprompt/** implementation of libmprompt
- *include/mpeff.h* header file of traditional algebraic effect
- *src/mpeff/** implementation of traditional algebraic effect
- *include/mplexeff.h* header file of lexically scoped effect handler
- *src/mplexeff/** implementation of lexically scoped effect handler
- *test/** test files for traditional algebraic effect
- *lextest/** test files for lexically scoped effect handler

4.5 Instruction to run

Run the following command to build the libraries and tests, and run the tests. The time to run will be printed out. The repository is at https://github.com/hflsmax/mplexeff

- mkdir build; cd build
- cmake .. -B . -DCMAKE_BUILD_TYPE:STRING=Release
- make -j
- ./test_mpe_main
- ./lextest_mpe_main
- ./lexopttest_mpe_main

5 EVALUATION

We use a suite of benchmark to compare the performance of traditional algebraic effect and lexically scoped handler. The tests are written slightly differently for each libraries because to be able to use lexically scoped handler, the handler needs to be passed down the call chain. Otherwise, tests used in the comparison are the same.

Figure 4 shows a table with three rows, each corresponding to traditional algebraic effect, lexically scoped handler and optimized

	reader	greader	counter	ucounter	ocounter	gcounter	ocounter1	counter10	ocounter10
traditional	0.000s	0.000s	0.056s	0.070s	0.554s	0.108s	0.578s	0.582s	1.869s
lexically scoped	0.000s	0.000s	0.045s	0.053s	0.485s	0.097s	0.482s	0.048s	0.527s
lexically scoped opt	0.000s	0.000s	0.043s	0.053s	0.484s	0.099s	0.482s	0.047s	0.486s
	mstate	amb	amb-state	state-amb	nqueens	triples	multi-install		
traditional cont.	0.000s	0.000s	0.000s	0.000s	0.804s	1.180s	0.511s		
lexically scoped cont.	0.000s	0.000s	0.000s	0.000s	0.784s	0.994s	0.507s		
lexically scoped opt cont.	0.000s	0.000s	0.000s	0.000s	0.778s	0.998s	0.127s		

Fig. 4. Runtime comparison

lexically scoped handler. The optimization is explained in Section 4.4; we split that into a separate case to observe the benefit of the optimization more closely. The experiment was carry out on a Linux machine with CPU *Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz*. Each entry is ran once.

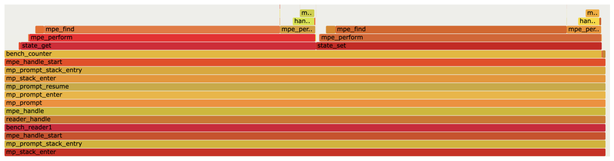

Fig. 5. counter10 flamegraph

The data shows that lexically scoped handler performs a lot better compared to traditional algebraic effect, and this can be contributed to the lack of stack-walking. The benchmark "ocounter10" shows the most dramatic improvement, where the runtime drops from 0.582s to 0.048s, a 91.7% decrease. This program is made of a deeply nested handlers, and effects are bubble up from deep down the handler chain to the top, therefore each effect invocation consists of expensive stack walking. We capture a flamegraph of the program in Figure 5, which shows that neraly 78% of the runtime is spent in the function "mpe_find", which handles the stack walking.

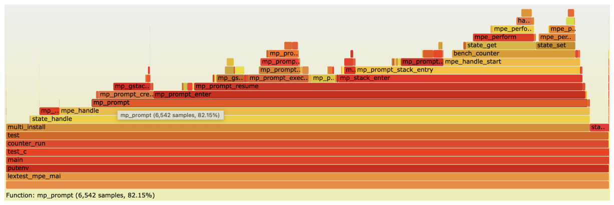

Fig. 6. multi-install flamegraph

The data also shows that the optimized lexically scoped handler performs very much the same as non-optimized one, except for the program "multi-install". The program consists of a loop that repeatly install a tail-resumptive handler that runs a short handled code. For the unoptimized implementation, this behavior results in repeatly creating stacklets and destroying them. In the optimized version, the library determines that the handler is tail-resumptive and therefore does not need to use continuation a first-class, so no stacklet is created. Again, we capture a flamegraph of the unoptimized program in Figure 6, which shows that 82% of the runtime is spent on managing the stacklet. The optimized version saves this unnecessary overhead.

Even though lexically scoped handler performs better in all benchmarks that we present, it can not encode some programs. For example, the program "rehandle"(not shown in the table) captures a continuation, and later resumes the continuation in a different context. When an effect is subsequently raised, the handler is search in the new context. This behavior can not be encoded in lexically scoped handler because the new handler is not within the lexical scope of the effect raise site.

6 CONCLUSION

In this paper, we study the implementation of delimited continuation, and implement the lexically scoped handler. We compare the performance of the lexically scoped handler and traditional algebraic effect, and found that due to the lack of stack walking, lexically scoped handler perform consistenly better. Therefore, there are more evidence to promote the use of lexically scope handler: not only is it easy to reason due to the preservation of modularity, it is also faster without losing much expressivity.

REFERENCES

Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-Level Effect Handlers in C+. *Proceedings of the ACM on Programming Languages* (2022).

Daan Leijen. 2017. Implementing algebraic effects in C. In *Asian Symposium on Programming Languages and Systems*. Springer, 339–363.

Daan Leijen. 2021. libmprompt. https://github.com/koka-lang/libmprompt.

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 566–579.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 206–221.

Yizhou Zhang and Andrew C Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.