Adaptive Effect Handling in Frank

Lukas Convent
University of Lübeck, Germany
convent@isp.uni-luebeck.de

Keywords Algebraic Effects, Effect Handlers, Modularity

Introduction Frank [12] is a functional programming language with support for algebraic effects that are statically tracked via an effect type system. Algebraic effects were introduced by Plotkin and Power [15–17] as a means to specify an interface between an effectful program and an effect handler. In order to interact with the environment, an effectful program may issue an effect such as "Give me a number!". Such effects are abstract and thus need to be interpreted. Plotkin and Pretnar's notion of an effect handler [18] fulfills this role: It may e.g. provide the continuation of the effectful program with some number, but may also drop the continuation and continue differently.

Effect handlers offer a flexible way of combining effectful components, but in most implementations including Frank there arises the problem that upon composing two effectful components, intermediate effects cannot be properly encapsulated. More generally, the need to decouple the *point of access* to an effect from the *point of introduction* of an effect demonstrates a lack of expressiveness. After motivating the problem in detail, we present the solution of adaptors as a new language construct in Frank.

Effect Handling in Frank Let us take a look at an example of the interaction between an effectful program and an effect handler in Frank. Consider the effectful program `ask! + ask!`, where `!` denotes that `ask` is applied to zero arguments. As in every standard programming language, the semantics of the `+` operator is clear, however the semantics of the `ask` operator is undefined. Such an abstract operator is called a *command*. Multiple commands, together with their signatures, form an *effect interface*, e.g.

```
interface Reader S = ask: S
interface Abort    = abort X: X
```

An effect interface may be polymorphic (e.g. `Reader S`), and a command may be polymorphic (e.g. `abort X`), too, where the latter allows instantiation *on every command request*. An instantiated effect interface like `Reader Int` is called an *effect instance*. Now consider the effect handler `const21` which can interpret the `Reader Int` effect instance by constantly responding to `ask` requests with 21. In Frank, an effect handler and an effectful program are brought together by application, e.g. `const21 (ask! + ask!)`. Note that the dynamic semantics of this term is fully defined, and one way of observing the dynamics in Frank is via a term reduction relation. Some intermediate reductions are omitted in the following example.

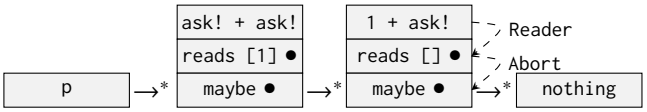

Figure 1. For `p := maybe (reads [1] (ask! + ask!))`,
(a) $\longrightarrow^*$ denotes the closure of the transition relation
(b) $\dashrightarrow$ denotes an effect flow from effect generator to handler

```
const21 (ask! + ask!)  →*  const21 (21 + ask!)  →*  42
```

In Frank, effectful programs, effect handlers and ordinary functions are generalised to the notion of an operator. In particular, an operator can both generate and handle effects. Consider the `reads` operator which is parameterised by a list of type `X` and handles `Reader X` by serving the next element of the list on every `ask` command. If the list is empty, it triggers the `Abort` effect. Consider furthermore the `maybe` operator which handles `Abort` by returning `nothing` and which returns `just v` if the argument computes to a value `v`. Here are two example reductions with both operators involved.

```
maybe (reads [1,2] (ask! + ask!))  →*  just 3
maybe (reads [1]   (ask! + ask!))  →*  nothing
```

In the following, a second perspective on the dynamic semantics of Frank is given, namely via transitions on a stack machine as shown in Fig. 1 (a). The correspondence to term reductions is as follows: A stack can be seen as the linearisation of the abstract syntax tree (AST) of a term where the subterm that gets reduced next is on top of the stack. A stack frame is a term that may have a hole • as a placeholder for the subterm currently being computed further up the stack.

A handler introduces a local scope over the computations it can handle. In the stack, a handler's scope reaches over the stack frames on top of the handler's frame. At every stack frame there is a collection of effects that can be handled further down the stack, which is called the *ambient (ability)*. Notice that there may be multiple instances of the same interface available, inducing an order on the instances in the ambient according to their position in the stack. The effect and type system statically tracks the ambient and guarantees that no command request ever misses its target.

The Problem: Leaking Effects Because stack diagrams explicitly show the reduction order, they are well-suited to point out for each sub-computation and effect their corresponding handler. Consider the dashed arrows in the third

POPL SRC'19, January 2018, Cascais/Lisbon, Portugal
2019.

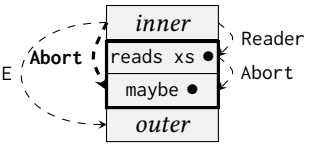

Figure 2. Stack of term *outer* `(maybe (reads xs` *inner*`))` where `E` $\notin$ { `Abort`, `Reader` } and the thick arrow marks the unwanted interception

intermediate stack in Fig. 1 (b), where an arrow connects an effect-generating frame to the one that is responsible for handling that effect in the current ambient.

Let us consider now the general scenario of applying the composition `maybe ∘ reads` (for a fixed list `xs`) to some *inner* computation within some *outer* context, as shown in Fig. 2. The intention is that any effect emitted by *inner*, except `Reader`, gets handled by *outer*. This is not what happens though. The composition `maybe ∘ reads` not only handles the `Reader` effect from *inner*, but also intercepts any `Abort` effect. This would interfere with any external handling mechanism between *inner* and *outer* if it uses its own `Abort` instance.

The general problem is that while building up the stack, the ambient grows monotonically while sometimes there is the need to remove effects. In the example, the `Abort` instance needs to be removed from the ambient at *inner*.

Recall that the ambient may contain multiple instances of the same interface, respecting their order. This allows one to formulate the problem even more generally: The way in which an effect is *accessed* in the ambient is exactly determined by where it is *introduced* in the syntax, which is sometimes undesirable. Besides the given example of the need to remove effects, there are use cases that require the reordering and or the copying of effects. Frank currently cannot express such changes to the ambient.

Adaptors as a Solution As a solution, *adaptors* allow to decouple the access to effects in the ambient from the syntactic positions of the handlers that introduced them. An adaptor specifies for an interface how its instances in the ambient are rearranged. This corresponds exactly to a renaming of De Bruijn indices. For example, the adaptor `<Reader(2, 0)>` applied to the ambient `[Reader Int, Reader Bool, Reader Int]` results in `[Reader Int, Reader Int]`. The instances for every interface in the ambient are indexed right-to-left, and so are the indexes specified in an adaptor. Thus the `0` index addresses the right and the `2` index the left `Reader Int` instance, while the `Reader Bool` instance is dropped from the ambient.

Returning to our example, one can insert an adaptor in order to solve the problem of the unintentional interception of the `Abort` effect.

outer `(maybe (<Reader()> (reads xs` *inner*`)))`

Statically, the `<Reader()>` adaptor removes all `Reader` instances from the ambient. Dynamically, each command is now attached with a *skip level* which is 0 by default. In our example, the adaptor `<Reader()>` increments the skip level of an arriving `ask` command, causing it to skip the next handler when going down the stack on the search for the right handler.

In Frank, an operator has to declare the effects that it handles in its signature, which is not only relevant for typechecking but also for reduction. It is natural to include adaptors in signatures, too. For example, the following operator that forwards `ask` commands and adds 1 to the result can have the following type.

```
inc : {<Reader(), Reader Int> X -> [Reader Int]X}
```

Here, the ambient's old `Reader` instance is first removed from the argument's ambient before adding a new `Reader` instance. Although adaptors are defined via renamings of indices, all adaptors can be represented without numbers through the use of syntactic sugar.

Related Work An early effect system that supports multiple static instances of the same effect includes Brady's library for Idris [5]. In contrast, Hillerström et al. row-based effect system for Links does not allow multiple instances [8]. In a different contrast, early versions of Bauer et al.'s Eff programming language allow the dynamic creation of effect instances, thereby offering an alternative way to encapsulate effects [1, 2]. Lindley et al. [12] propose negative adjustments which correspond to the special adaptors that remove the most-recent effect instance from the ambient. Leijen introduces an inject operation [9, 11] that corresponds to the same special adaptors, for his language Koka [10]. Biernacki et al. [3] introduce a lift operation that corresponds to the same special adaptors, for their $\lambda^{H/L}$ calculus. They furthermore extend $\lambda^{H/L}$ and add basic combinators to adjust the ambient [4] but are not quite as expressive as with adaptors. Piróg et al. [14] observe the unflexible nature of scope and semantics of a handler being tied together. The formal dynamic semantics in terms of a stack machine has a practical counterpart in McBride's language Shonky [13] that is used also as a backend of the Frank compiler.

Contributions and Outlook The main contributions are

1. Identifying the problem of effect leakage as a consequence of the inflexible nature of the ambient
2. Proposing the general solution of adaptors to adapt the ambient in a formal way
3. Implementation of adaptors in the Frank compiler [7]
4. Evaluation of adaptors on an actor concurrency model, thereby achieving effect encapsulation of the components in a larger practical example

A minor contribution is the new presentation of the (yet unpublished) contribution 1) by describing the link between an operational semantics and a stack-machine-based semantics.

Due to the rather recent proposal of effect handlers [18], effect handling is still a young area. Because it integrates naturally into type systems and provides a powerful abstraction mechanism, one can expect to find it in the toolboxes of tomorrow's programming languages. I am especially interested in further comparing the results of Piróg et al. [14] with respect to the separation of scope and semantics of a handler.

Remarks. Contributions 1), 2), 4) are also part of a paper written together with my co-authors Sam Lindley, Conor McBride and Craig McLaughlin which is currently under submission for ESOP'19. Upon identifying contribution 1) and working out the actor model (part of contribution 2) in my M.Sc. dissertation [6] (unpublished), I was substantially involved in developing the idea of adaptors (see contribution 2). Due to its simplicity, I reused the composition of `maybe` with `reads` in this presentation as a running example, originally proposed by Sam Lindley.

References

[1] Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *CALCO (Lecture Notes in Computer Science)*, Vol. 8089. Springer, 1–16.

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL* 2, POPL (2018).

[4] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *PACMPL* 3, POPL (2019).

[5] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 133–144.

[6] Lukas Convent. 2017. Enhancing a Modular Effectful Progamming Language. *M.Sc. dissertation, University of Edinburgh* (2017).

[7] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2018. Frank Adaptors Branch. https://github.com/frank-lang/frank/tree/adaptors

[8] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27.

[9] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP (EPTCS)*, Vol. 153. 100–126.

[10] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499.

[11] Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report. Microsoft Research.

[12] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.

[13] Conor McBride. 2016. Shonky. https://github.com/pigworker/shonky

[14] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *LICS*. ACM, 809–818.

[15] Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *FoSSaCS (Lecture Notes in Computer Science)*, Vol. 2030. Springer, 1–24.

[16] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FoSSaCS (Lecture Notes in Computer Science)*, Vol. 2303. Springer, 342–356.

[17] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

[18] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).