Functional programming with algebraic effects

(Programowanie funkcyjne
z efektami algebraicznymi)

Szymon Kiczak

Praca inżynierska

Promotor: dr Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

29.01.2024

Abstract

Algebraic effects and their handlers are a relatively new concept in functional programming. They provide generalization over some common programming concepts. One of functional programming languages that implement algebraic effects is Helium language developed by team from University of Wrocław — Institute of Computer Science. Since it is a research language, not many long and complicated programs were written in it. Goal of this thesis is to create parser combinators library using Helium language with application of algebraic effects and describe a process of doing it. Feedback from my experience, as a person who previously never wrote a program in Helium, may help in further development of that language.

Efekty algebraiczne i ich handlery to dosyć nowy koncept w programowaniu funkcyjnym. Pozwalają one uogólnić niektóre z występujących już w wielu językach programowania pojęć. Jednym z funkcyjnych języków programowania który umożliwia korzystanie z efektów algebraicznych jest jezyk Helium rozwijany przez zespół z Instytutu Informatyki Uniwersytetu Wrocławskiego. Jak dotąd, jako że jest to język służący głównie do badań, powstało w nim bardzo niewiele dłuższych programów. Celem niniejszej pracy jest stworzenie programu realizującego kombinatory parserów korzystając z Helium z wykorzystaniem efektów algebraicznych i opisanie procesu tworzenia tego programu. Moja perspektywa jako osoby wcześniej niezaznajomionej z tym językiem może pomóc w dalszym jego rozwoju.

Contents

Chapter 1

Introduction

1.1 My perspective

I am an computer science student with some experience in writing programs in functional programming languages, for example in Racket and Ocaml, and using functional programming features in multi-paradigm languages such as Python and Rust. All of these are mature langauges used by sizeable communities of programmers with thorough documentation. However, goal of this thesis is to develop a program in Helium, which is a new experimental language used by few researchers with nearly nonexistent documentation and support. By trying to develop a non-trivial program in it, I offer a fresh look at the problems faced by newcomers attempting to understand algebraic effects and this language.

1.2 Algebraic effects

Despite my experience with functional programming and different programming languages, before starting this project I have never heard about algebraic effects. This is not surprising as they are a relatively new concept known mainly to programming language researchers. For me the most intuitive way of understanding algebraic effects was that they can be perceived as a generalization over common mechanism of error throwing and catching. I was introduced to first examples and usage of them (in Koka language) by the Daan Leijen's "Type directed compilation of row-typed algebraic effects" paper recommended by my advisor. This article emphasize that algebraic effects come with various advantages: "they can be freely composed, and there is a natural separation between their interface (as a set of operations) and their semantics (as a handler)" [1].

1.3 Parser Combinators

During the initial stage of research for this thesis I had to choose what type of program I want to implement in Helium. Parsing and especially parser combinators, as they are a functional approach to parsing, caught my attention. I did not want my program to use algebraic effects in a forced, redundant way. Choosing parser combinators as a matter of this thesis allowed me to use algebraic effects and their instances in my program reasonably.

Parser combinator is a function that combines multiple parsers (also functions) to create new, more complicated parser. Two most basic combinators of parsers are the sequential combinator and the alternative combinator [2]. A sequential combinator executes one parser after another. An alternative combinator chooses from two parsers one that is not failing (if it is possible). More advanced combinators can be build using these two functions.

1.4 Helium language

Helium is a research programming language designed and developed by the team from Institute of Computer Science of University of Wrocław and it is currently not used outside of academic purposes. As described on its own website it is "a very experimental programming language that boasts advanced algebraic effects with effect instances, sophisticated polymorphism and abstraction for types and effects through a module system in the tradition of ML [3]". Important feature of Helium that is not present in Koka is a possibility of using effect instances. They are easy to use and clearly explained in documentation [5]. At the moment of writing this thesis (January 2024), not a lot of complicated programs were written in Helium. This project aims to contribute to the codebase of programs made in that language.

As a result of being an experimental programming language, Helium's official documentation [4] is very incomplete, which has proven to be a major obstacle during development of my program. Most recent version of documentation consists of only four pages explaining most important concepts introduced in this language. In particular there is no documentation for built in functions from `Prelude, List, String`, etc. To discover whether certain basic function exists and what it does, user has to manually search through programs in `lib` directory. Helium's standard library also lacks certain basic functions which are usually included in functional languages. During development I noticed that there is no included function to transform `String` to `List Char` and no way to explicitly extract the first element of a list (`head` in most languages).

Apart from lacking documentation, some parts of a language are not implemented and error messages that appear in that cases do not suggest how to circumvent or solve these problems. Numerous times when importing my program into interpreter, I encountered fatal error with enigmatic message:

```
Fatal error: exception Failure("Not␣implemented:␣implicit␣instance"),
```

which had to be explained to me by my advisor. Additionally, I also came across a bug, where in type of expression `<unnamed>` was displayed instead of a name of an effect instance.

1.5 State effect

Program presented in this thesis extensively uses algebraic effects with state. Storing internal state allows all handler operations to share information between them, for example in the case of parser effect, state allows operators to know what has already been parsed and what is not yet parsed.

My first introduction into algebraic effects with state was through the previously mentioned “Type directed compilation of row-typed algebraic effects” paper [1], which uses Koka language in all examples. In that article state effect handler is defined in the following way:

```
val state = handler(s) {
    return x -> (x,s)
    get() -> resume(s,s)
    put(s') -> resume(s',())
}
```

However, implementation of effects with state in this project is based on Helium wiki page [5] and Handlers Tutorial [6]. The following example of a `State` effect and handler implementation in Helium language is taken from the Helium wiki page.

```
signature State (S : Type) =
| put : S    => Unit
| get : Unit => S

let hState init =
  handler
  | put s      => fn _ => resume () s
  | get _      => fn s => resume s s
  | return x   => fn _ => x
  | finally f => f init
  end
```

State effect has two operations: put and get, former for setting internal state to desired value, latter for reading current value of state. Put takes a new state as an argument and returns unit value, while get takes unit as an argument and returns state. Above handler evaluates to a lambda expression which is calculated with provided input when handler exits using `finally` keyword.

Koka's syntax allows simpler declaration of the state effect handler than Helium, as it does not require wrapping everything in a lambda expression (at least not explicitly). Difference between these two ways of declaring handlers made me confused. The way in which state handler works is unfortunately not explained in Helium wiki, so I had to spend time deciphering how it compares to the one I have seen before.

Chapter 2

Development process

First partially working version of my program contained only one algebraic effect with state, which was used to parse a list passed as an initial state. State consisted of two lists: one with the already parsed characters, one with the characters left to parse. It was intended to be used through the `parse` function which does not require user to know much about effect and handler:

    parse expr list,

where `expr` is an expression to use during parsing (for example `isInt`) and `list` is a list of characters to parse. Early implementation of the `Parse` effect and `parser` handler looked like that:

```
signature Parse (A: Type) =
| satisfy: (Char -> Option A) => Unit
| get: Unit => Pair (List (Option A)) (List Char)
| put: Pair (List (Option A)) (List Char) => Unit

let parser (init: List Char) =
    handler
    | satisfy f => fn (p, cont) =>
        let (result, rest) =
            match cont with
            | x::xs => ((f x), xs)
            | [] => (None, [])
            end
        in resume () (result::p, rest)
    | get _     => fn s => resume s s
    | put s     => fn _ => resume () s
    | return x  => fn _ => x
    | finally f => f ([], init)
    end
```

```
let parse expr (str: List Char) = handle ‘a in expr ‘a () with
   parser str .
```

Usage example of this program:

```
[IO,RE]> let expr () = isInt (); isInt (); get ();;
[IO,RE]> parse expr [’1’, ’2’, ’3’];;
(,) ((::) (Some ’2’) ((::) (Some ’1’) [])) ((::) ’3’ [])
: Pair (List (Option Char)) (List Char).
```

As can be seen in this example, sequencing multiple parsers is very easy with this implementation. At this point I realized that unfortunately it is impossible to express alternative parser combinator with such definition of the effect and handler. Adjusting my program to be able to compute alternative parser combinator (`orElse`) correctly turned out to be a biggest problem I had during the whole process of writing this thesis. New effect to handle branching of computation had to be interchangeable with parse effect, so it would be possible to write parsing instruction in one expression, for example:

```
isInt (); orElse (isInt ()) (isLowerCaseLetter ()) (); get ();
```

and handle it once (with one or multiple effect handlers).

My intuition, stemming from previous programming experience in other languages, was to compute result of the first expression and if it results in success, return that expression, otherwise return second expression. Following example is just one of the many versions of effect operators I created, but it is quite representative of my thought process:

```
| tryelse expr1 expr2 =>
    let (success1, parsed1, cont1) = get expr1
    in match success1 with
    | True => resume expr1
    | False => resume expr2
    end.
```

Main problem with that approach is that the state is shared across both possible computation paths. This means that when first expression does not succeed, result of combining two parsers with alternative operator will be the same as result of sequencing two parsers.

It has taken me over a month and several iterations of programs and I still did not have fully working solution. Finally I gave up and had to rely on help of my advisor to show me how to implement alternative parser combinator. His idea with list of resumptions generally turned out to be very unintuitive for me at first. I did not know that resumptions (denoted with `resume` keyword) could be put in a deferred state in a list, which was the main trick of working solution. That fact was not explained or hinted in documentation. During learning about Helium I assumed that `resume` is a special keyword that returns some value and does not stop computation when handled.

Chapter 3

The final program

3.1 Computation model

In order to create a simple parser combinator library I define two following algebraic effects with their corresponding handlers.

The `Branch` algebraic effect is needed for representing different paths of execution and it also contains a hidden state. This state is a stack (list) of resumptions in the following format:

```
data rec StateElem (E: Effect) (T: Type) =
{ run: Bool -> [E] List (StateElem E T) -> T } .
```

Each element of state in `Branch`: `StateElem` is delayed resumption of flip operator. Flip can resume twice: first time returning `True` and with second resumption added to state, second time when delayed resumption is called during handling of the `fail` operator that returned `False`. Return value of flip operator indicates whether current path of execution is a result of successfull or failed `"if"` condition.

```
signature Branch =
| fail: Unit => Unit
| commit: Unit => Unit
| flip: Unit => Bool
```

As defined above, these three effect operations are meant to do the following:

1. `fail` — ends path of execution and calls first resumption from the stack
2. `commit` — ends one path of execution and return result
3. `flip` — returns `True` and follow with current path of execution and adds second path of execution on stack.

```
let branchhandler initstate =
    handler
    | fail ()    => fn state =>
        match state with
        | []     => resume () []
        | f::fs => f.run False fs
        end
    | commit res => fn state =>
        match state with
        | []     => None
        | f::fs => resume res fs
        end
    | flip ()    => fn state => resume True ({run=resume}::state)
    | return x   => fn _ => Some x
    | finally f => f initstate
    end
```

Similarly to usual `State` effect, above handler evaluates to a lambda expression which takes initial state, which in case of this program it will always be empty list. Currently `flip` and `commit` effect operators are used only in the `orElse` function explained later. Generally `commit` operator should only be used when the handler's state is not empty, after `flip` operator was called. When called on an empty state, `commit` operator causes whole handled calculation to return `None` value indicating that an error occurred during parsing. Otherwise handler should return `Some result`.

Algebraic effect named `Parse` is responsible for storing functions which will be evaulated later (satisfy) with hidden state of computation. It is parametrized by type `A` which characterizes the type of data returned by a parser. State of this effect has the following type:

```
Pair (Pair Bool (List (Option A))) (List Char)
```

and contains information about:

1. whether whole parsing ended without errors — `True`, or not — `False`,
2. list of type `Option A`, where `Some x` tells that character was parsed succesfully and `None` means that character did not meet condition applied to it,
3. list of characters not yet parsed.

```
signature Parse (A: Type) =
| satisfy: (Char -> Option A) => Unit
| get: Unit => Pair (Pair Bool (List (Option A))) (List Char)
| put: Pair (Pair Bool (List (Option A))) (List Char) => Unit
```

As defined above, these three effect operations are meant to do the following:

1. `satisfy f` — execute function `f` on first character from list of elements left to parse (last part of the state),
2. `get` — returns hidden state,
3. `put s` — inserts value `s` into state.

Handler of `Parse` effect is parametrized with `init` — a list of characters to parse. This handler holds state in the same way as State from Helium examples. `Parse` returns function which takes an initial state and returns changed state after performing all operations in expression passed to handler. Initial state is `(True, [], init)`, at beginning there are no errors, nothing was parsed and list of characters remaining to parse is full.

```
let parser (init: List Char) =
    handler
    | satisfy f => fn (prevsuccess, parsed, cont) =>
        let (result, rest) =
            match cont with
            | x::xs => ((f x), xs)
            | [] => (None, [])
            end
        in match result with
            | Some x => resume () (prevsuccess, result::parsed, rest)
            | None => resume (fail ()) (False, result::parsed, rest)
            end
    | get _     => fn s => resume s s
    | put s     => fn _ => resume () s
    | return x  => fn _ => x
    | finally f => f (True, [], init)
    end
```

When handling expression `satisfy f` handler has to evaluate function `f` on the first element of not yet parsed input contained in state. If that function accepts character `x` from input, state of handler is updated with `Some x` added to the parsed characters list, otherwise state is updated with `False` value in field denoting whether whole parsing ended without errors and `None` is added to parsed characters list. In latter case expression returns `fail ()`, which causes outer `branchhandler` handler to consider this path of computation as failed if `branchhandler`'s resumption stack is not empty. When `branchhandler`'s resumption stack is empty, `fail` evaluates to `() : Prelude.Unit`. This means that this handler can only be used together with `branchhandler`.

Two most basic parser combinating operators: sequence and alternative are implemented as `(;)` and `orElse`. These two functions can be used to construct more complicated parsers. Sequencing two parsers could be realized as a separate function — `andThen f1 f2 ()` but I decided to use already existing code. In result, sequencing two parsers is done by using basic function — `(;)` from Helium Prelude, as all parsers are deferred functions taking `() : Prelude.Unit` as last argument:

```
    let ( ; ) () x = x .
```

Alternative of two parsers - `orElse` has to use `Branch` effect operators to represent two possible paths of computation:

```
let orElse f1 f2 () =
    if ( flip () ) then (
        f1 ();
        commit ()
    )
    else (
        f2 ()
    ).
```

When handling `orElse` expression, first resumption of `flip ()` operator will return `True` value and if the path with `f1 ()` function returns without `fail ()`, `commit ()` will throw away a second `flip ()` resumption and resume further computation. Otherwise `flip ()` operator will resume again, this time with `False` value and second path with `f2 ()` will be executed.

All simple parsers that accept just one character are defined using `satisfy` operator from `Parse` effect. Names of these simple parsers are generally self-explanatory: `isInt, isSpace, isLowerCaseLetter,` etc. Example of such parser is the following one which checks whether a character is an integer or not:

```
let isInt () =
    satisfy (fn c => if Char.isDigit c then Some c else None).
```

When composed by parser combinators, lambda functions inside `satisfy` operator will be executed during handling of whole expression containing multiple operators by `parser init`. Characters used as input for these lambdas will be provided from `init` list.

3.2 Usage

Parsing list of characters `input` with predefined expression `E` in most primitive way can be done using that expression with `parser` handler parametrized with `input`:

```
handle ‘a in
```

```
    handle ‘b in
        E ‘a ‘b ()
    with parser input
with branchhandler [].
```

In the following examples, `isInt` checks whether character from input is an integer and `repeat f n` is a function that sequences `n` functions `f`:

```
[IO,RE]> let expr3 (‘a: Branch) (‘b: Parse Char) () =
    repeat (isInt ‘b) 3 (); get ‘b ();;
[IO,RE]> handle ‘a in
            handle ‘b in
                expr3 ‘a ‘b ()
            with parser [’1’, ’2’, ’3’, ’4’, ’5’]
        with branchhandler [];;
Some
    ((,)
        ((,) True ((::) (Some ’3’) ((::) (Some ’2’) ((::) (Some ’1’)
    []))))
        ((::) ’4’ ((::) ’5’ [])))
: Option (Pair (Pair Bool (List (Option Char))) (List Char)).
```

As can be seen in the example above, list with result of parsing is reversed, because handler adds newly parsed elements to beginning of a list (faster than adding on end of list). Easiest way to use parser in common scenario, with predefined expression and input in string type (not in list of characters), is with a simple wrapper:

```
[IO,RE]> let expr4 (‘a: Branch) (‘b: Parse Char) () =
    repeat (isInt ‘b) 4 (); get ‘b ();;
[IO,RE]> parsewrappedstring expr4 "1234567" ();;
(::) (Some ’1’) ((::) (Some ’2’) ((::) (Some ’3’) ((::) (Some ’4’)
    [])))
: List (Option Char).
```

3.2.1 Examples

Example of `orElse` usage, this parser first tries to parse integer and if it fails, it tries to parse lowercase letter:

```
[IO,RE]> let letterorinteger (‘a: Branch) (‘b: Parse Char) () =
    orElse ‘a (isInt ‘b) (isLowerCaseLetter ‘b) (); get ‘b();;
[IO,RE]> doparse letterorinteger [’a’, ’b’, ’c’] ();;
Some ((,) ((,) True ((::) (Some ’a’) [])) ((::) ’b’ ((::) ’c’ [])))
: Option (Pair (Pair Bool (List (Option Char))) (List Char))
```

```
[IO,RE]> parsewrappedstring letterorinteger "abc" ();;
(::) (Some 'a') [] : List (Option Char).
```

The following example shows that when `parser` can't parse a character it just leaves `None` in result list and changes result of whole parsing to `False`.

```
[IO,RE]> handle `a in
            handle `b in
                get `b (orElse `a
                        (notInt `b)
                        (isInt `b (); notInt `b (); isInt `b) ())
            with parser ['1','2','3']
        with branchhandler [];;
Some ((,) ((,) False ((::) (Some '3') ((::) None ((::) (Some '1')
    []))))) [])
: Option (Pair (Pair Bool (List (Option Char))) (List Char))
```

Example below uses `choice` function, which constructs a chain of `orElse` calls with each function from the list passed in argument:

```
[IO,RE]> let expr (`a: Branch) (`b: Parse Char) () =
            (choice `a [(repeat (isInt `b) 5),
                        (repeat (isLowerCaseLetter `b) 5),
                        (repeat (isUpperCaseLetter `b) 5)]) (); get `
   b ();;
[IO,RE]> doparse expr (strtolist "ABCDefgh") ();;
Some
    ((,)
        ((,)
        False
        ((::)
            None
            ((::)
                (Some 'D')
                ((::) (Some 'C') ((::) (Some 'B') ((::) (Some 'A')
    [])))))))
        ((::) 'f' ((::) 'g' ((::) 'h' []))))
: Option (Pair (Pair Bool (List (Option Char))) (List Char)).
```

In this example `commit` effect operator is used incorrectly, without calling `flip` operator before. Consequence of such error is `None` being returned as a result of parsing.

```
[IO,RE]> handle `a in
            handle `b in
                isInt `b (); commit `a (); get `b ()
            with parser ['1', '2']
        with branchhandler [];;
None : Option (Pair (Pair Bool (List (Option Char))) (List Char))
```

Chapter 4

Conclusions

4.1 Parsing library

In this thesis I created a model for parser combinator library using algebraic effects. Due to time constraints I was not able to implement more functions, nevertheless other functions combinating parsers can be created using basis established there.

4.1.1 Problems and limitations

In current state all parsers created by combining functions from this project return a list of parsed characters, not tree-like data structure which would be more universal. Only a basic subset of possible parsers is defined which results in very limited practical usability.

4.1.2 Further development

Most importantly more advanced parser combinators should be added to extend this project's use cases. Additionally better modularisation and encapsulation would benefit reusability of this code.

4.2 Helium language

During the development of new programming languages, creating new programs helps to detect bugs and build new features. Development of this project resulted in detecting one bug in Helium and allowed me to suggest adding some basic functions into standard library, which may help in further improvement of Helium. In addition, as a new user, I was able to take a fresh look at the documentation problems. In its current state, the documentation (Helium wiki) makes learning Helium's syntax hard for new users. Majority of the standard library is not featured in documentation at all. In result I had to often rely on trial and error method to figure out how this language works.

Naturally as an experimental programming language, Helium contains some bugs and unfinished features. Unfortunately some of the error messages are not descriptive enough. Because of that, Helium user can be confused whether error returned by compiler is an error of language itself or problem caused by badly written program. For example error message:

```
Fatal error: exception Failure("Not␣implemented:␣implicit␣instance")
```

could be refactored as:

```
Fatal error: exception Failure("Not␣implemented:␣implicit␣instance.
You␣could␣try␣applying␣effect␣instance␣explicitly.").
```

Another solution that could help new users would be to include in documentation set of commonly encountered errors and solutions to them.

Bibliography

[1] Type directed compilation of row-typed algebraic effects. POPL 2017: 486-499

[2] S. Doaitse Swierstra, Combinator Parsing: A Short Tutorial. LerNet ALFA Summer School 2008: 252-300 `https://www.cs.tufts.edu/~nr/cs257/archive/doaitse-swierstra/combinator-parsing-tutorial.pdf`

[3] Helium language readme file `https://bitbucket.org/pl-uwr/helium/src/master/`

[4] Helium wiki homepage `https://bitbucket.org/pl-uwr/helium/wiki/popl20/Home.md`

[5] Helium language tutorial `https://bitbucket.org/pl-uwr/helium/wiki/popl20/Tutorial_Instances`

[6] Matija Pretnar, An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. MFPS 2015: 19-35