MINISTRY OF EDUCATION AND SCIENCE OF THE RUSSIAN FEDERATION

Federal state autonomous higher educational institution
SOUTHERN FEDERAL UNIVERSITY

I. I. Vorovich Institute of Mathematics, Physics & Computer Science

Georgy Lukyanov

STRUCTURING EFFECTFUL COMPUTATIONS

Thesis submitted for the degree of
MSc in Computer Science

Supervisor:
Assistant professor Artem Pelenitsyn

Rostov-on-Don
2017

CONTENTS

INTRODUCTION

Software systems are required to be reliable. For example, biomedical implants must be able to operate autonomously within patients, adapting to short term and long term changes, with required lifetimes in the order of decades, and one-minute downtime failure of banking software may lead to significant expenses. Safety and solidity of software may be achieved through testing and post-design quality assurance, but there is always a possibility to leave some branches of an operations scenario untested, leading to potential disaster.

In contrast with post-design testing, that doesn't provide full correctness guarantees, formal methods provide a systematic approach for developing complex systems in a correct-by-construction manner. One reason of advancement of formal methods in comparison to testing is a maximal possible elimination of human errors. For instance, techniques known as model checking perform exhaustive search through entire space of possible states of the system and evaluating its status in every single case, completely excluding any possibility of non-specified behaviour (presuming the correctness of a specification).

One class of formal verification techniques involve the advanced type systems of modern programming languages. Powerful yet lightweight formal verification provided by these languages are based on famous Curry-Howard correspondence — a direct relationship between computer programs and mathematical proofs — that was discovered by the American mathematician Haskell Curry and logician William Alvin Howard in late

1960s. The famous Philip Wadler's paper "Propositions as Types" [1] gives a historical perspective alongside with points of view on Curry-Howard correspondence from different fields of mathematics and computer science. Curry-Howard correspondence is of great value for software developers because it provides a possibility to formulate desired properties of programs in terms of types and automatically check the correctness of those programs through type checking.

Advances in type theory and theory of programming languages led to the development and spreading of effects systems — a particular kind of type-guided verification providing a possibility to separate pure computations from ones flavoured with a computational side effect, like, for instance, file system IO. These techniques was first given an account by Moggi [2] who used monads to provide a denotational semantics to the λ-calculus with effects, and then Wadler [3] gave monads a practical instantiation in Haskell programming language.

Computational effects control techniques got plenty of attention both in academic and software engineering communities. As applications, such as, for example, parser combinators [4], were explored, a lot more requirements and demands were introduced: combine effects in a modular way and provide finer-grained control for effects. This led to development of both monadic [5] and alternative approaches [6] [7].

This work aims to application of programming languages features supporting explicit and precise control of computational effects. It addresses the problem of construction of parser combinators libraries and full-scale web applications using three approaches to effectful programming:

- Monad transformers [5] in Haskell.
- Algebraic effects and effects handlers [7] in a form of extensible effects [8] in Haskell

- The Frank [9] programming language featuring first-class support of algebraic effects and effects handlers.

Main results of this work were presented in international and local conferences. The work on constructing parsers with extensible effects was presented in 4th international conference “Tools and Methods of Program Analysis 2017” [10] in Moscow. The material on construction of parsers in Frank was presented in the conference “Programming languages and compilers 2017” in Rostov-on-Don [11].

This thesis is structured as follows: the first chapter gives a historical retrospective of research on computational effects; the second and third chapters describe the approaches to structuring of effects on programming languages and present announced parser combinators libraries; the fourth chapter demonstrates the usage of functional programming and computational effects control on a case-study — the development of full-fledged Haskell web application.

GOAL STATEMENT

The goal of this work is to study the applications of approaches to structuring of computational effects to real-world software development. The thesis explores the monadic and algebraic effects and handlers frameworks by solving a standard model problem: building a parser combinators library. The work include a comparison of these approaches in terms of expressiveness and, for some of them, performance, and makes observations about their applicability.

This thesis addresses the stated goals by breaking by solving the following tasks:

- Develop a monadic parser combinators library to illustrate the usage of monad transformers in Haskell.
- Develop a parser combinators library on top of extensible effects — an embedding of algebraic effects into Haskell.
- Highlight the differences between monad transformers and extensible effects.
- Use Frank programming language featuring first-class support of algebraic effects and effects handlers to analyse the approaches to parsing libraries' construction and report the experience.
- Benchmark the performance of the developed libraries and compare it to a one of existing solutions.

- Demonstrate the usage of the typed functional programming language with an effect system at scale: develop a full-featured effect-structured web application with Haskell.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor Prof. Artem Pelenitsyn for his support, motivation, knowledge and qualification. Prof. Pelenitsyn was the first person who exposed me to real research activities. I thank him also for being a person who cares. I doubt I would have started to do research on functional programming if I wouldn't have his support at one critical moment in the past.

Besides my advisor, I would like to thank Prof. Vitaly Bragilevsky for being the best teacher I've met. His courses on the theory of computation, functional programming and certified software development have advanced my knowledge and skills crucially. Besides being an excellent computer science teacher, Prof. Bragilevsky is also an inspiring, motivating and concerned, person. Thanks to his support, I've been able to go on an academic exchange programme and get the experience which made me rethink many aspects of my life.

I thank the other teachers and my fellow students for all the assistance I've received and for a perfect study environment I've had though all these years.

Last but not the least, I would like to thank my family for always giving me a support if I was in need.

CHAPTER 1

A HISTORICAL RETROSPECTIVE ON COMPUTATIONAL EFFECTS

Recent advances in the theory of programming languages led to the development and spreading of functional programming languages with advanced type systems. This provide software engineers with a possibility to encode system specification in type-level, enforcing statically checked guarantees of correctness. A large cluster of errors is introduced into programs by uncontrolled side effects such as file system IO, network communication and mutable state. While many mainstream programming languages such as C++, Java and C# follow static type discipline, they do not track side-effects, thus making it harder to reason about program correctness. Being an impure language, Java has a feature of *checked exceptions* that might be considered a very simple effect system specialised for exception handling. In contrast, pure languages like Haskell and Idris oblige all kinds of effectful programs to have special type markers.

```
int plus(int x, int y) {
  print("I'm mutating the World without you noticing!");
  return x + y;
}
```

Listing 1.1 — Uncontrollable side-effect in an impure language

The development of approaches to the structuring of computations with side effects and implementations of these approaches in program-

```
plus :: Int -> Int -> Int
plut x y = x + y
```

Listing 1.2 — Pure Haskell function

```
plusIO :: Int -> Int -> IO Int
plutIO x y = do
  print "I'm mutating the World, but you know it"
  return (x + y)
```

Listing 1.3 — Impure Haskell function with an effect annotation

ming languages has an exciting history. This chapter gives a historical excursion.

1.1. Origins of side effects control

From the very beginning, programming languages have been created to be the lingua franca for humans and machines. And the compilers must have been the translators. Therefore, the evolution of programming languages and compilers is massively affected by the dichotomy between the readability for humans and efficiency of machine code generation. This dichotomy was one of the primary motivations for programming language research: a field of theoretical computer science which studies syntax and semantics of programming languages.

Software, be it a number cruncher or a compiler, or an accounting system, is not likely to be developed and maintained by the efforts of one person. Therefore, the syntax of a programming language must be readable, and the semantics must be clear and adequate. Moreover, in some fields errors in software are impossible to tolerate. The need to build correct programmes leads to the development of pre and post-development software verification techniques. The first category includes various kinds of testing: unit testing, test-driven development, functional and integration testing. But the bad thing about testing is that it's, in fact, impossi-

ble to *prove* that the system is correct. Therefore, programming languages must have been enriched with a fundamental way to construct only correct programmes, to make the incorrect ones unrepresentable. This leads to the second category of software verification: the one based on *formal methods* and, more specifically, type systems of programming languages. It would be great to design the language which would reject incorrect programmes. And that is exactly what an adequate type system can do.

So, every programming language has a set of built-in constructions and syntactic rules. Programmers could take the constructions and combine them with the rules. But the programming language also have a semantics — a way to assign a *meaning* to every *term* of the language. There are several approaches to describe programming language semantics, including *operational*, *denotational*, and *axiomatic*. And the task for the programming language designers is to construct the language in a way when the terms with inadequate semantics would be rejected by the type system.

A classical example of how type system could help to introduce correctness control into a programming language is the difference between the untyped λ-calculus and the simply typed λ-calculus. If we consider a property of *strong normalisation* of β-reduction as a criterion, then it is shown that for the untyped λ-calculus β-reduction as a rewrite rule is not strongly normalising — i.e. there are such things as non-terminating computations or "infinite loops". Whereas for the simply typed λ-calculus β-reduction is strongly normalising; hence evaluation of any term would terminate in a finite number of steps. That is so because constructions which lead to non-termination, such as **Y**-combinator, are not *well-typed* in the simply typed λ-calculus.

Through the past years, a wide and deep research on typed lambda calculi was carried out, and then it found applications in the theory of programming languages and in functional programming. Both untyped (or *uni-typed*) and typed functional programming languages have a strong theoretical background and have always been a platform for researchers to explore applications of new ideas. In the late 1980s, typed functional

programming started to receive a lot of attention and plenty of ideas got implemented in experimental languages. One of the primary and most promising tracks was the introduction of purely functional programming — a style of building the structure and elements of computer programmes, that treats all computation as the evaluation of mathematical functions. This approach extensively uses notions of *immutability* and *referential transparency*.

And again, the development of programming languages was mostly driven by a dichotomy, but this time between abstract beauty of purely functional programmes and necessity of impure ones. That is, to be valuable, programmes must have *side-effects*: output their result to a file, communicate with a user, mutate state, etc. But naive introduction of side-effects into purely functional programmes was reported [12] to make programmes denotational semantics unstable and non-extensible. Therefore, high demand for solid mathematical foundations for controlling side-effects emerged.

1.2. Emergence of effect systems

Initially, purely-functional programming languages with immutable data structures were required to explicitly track all changes to the data in operation. This was making programmes cumbersome and wordy. But then it was noted by Moggi [2] that most programmes dealing with state and other side-effects follow the same pattern. He proposed to use *monads* — a notion from category theory — to structure side-effecting computations and factor the semantics via a new, generic, computational metalanguage that saves repeated work, modularises, explains, cleans up reasoning by moving side-conditions into the type system.

- Separate values A from computations TA, which may have observable behaviour other than producing a value of type A.

- Given a category C, T is an *endofunctor* $T : C \to C$, so can lift $f : A \to B$ to $Tf : TA \to TB$, and this preserves identity and composition.
- There's a *natural transformation* with components $\eta_A : A \to TA$ which expresses how values may be (uniformly) viewed as trivial computations.
- There's a natural transformation $\mu_A : T(TA) \to TA$ that lets (uniformly) combine effectful behaviours, so computation of a computation could be treated as a computation
- Satisfying monadic laws.

Later, Wadler grasped [3] the Moggi's idea and implemented it in the purely-functional language Haskell. That was the first well-known incorporations of monads into a programming language.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b)   -> m b
  (>>)   :: m a ->  m b         -> m b
  return ::   a                 -> m a
  fail   :: String -> m a
```

Listing 1.4 — Monad type class in Haskell

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Listing 1.5 — Monad laws

Monads were widely adopted by functional programming community and were used to refactor and unify such techniques as list comprehensions, purely-functional exceptions and state-threading. Many more monad instances were later discovered, including functional parsers [4], ways to control concurrency, logging and a lot of others. Moreover, it was

discovered that monads are a convenient way to build domain-specific languages (DSLs).

Essentially, every monad represents one computational effect: state, non-determinism, exceptions, etc. But complex programmes is likely to require an ability to produce multiple side effects. Therefore, a way to describe computations with several computational effects in a modular manner were demanded.

The first attempt to formally approach the effects combination problem was made by Wadler and King [13] who investigated a possibility to combine monads with each other to yield a *combined monad* — a one possessing properties of two or more monads. Then, Liang at al. [5] came up with the notion of *monad transformers* — building blocks for construction of modular programming language interpreters. The major achievement of the paper is a way to specify effects interactions through *lifting* and changing semantics by reordering monads in *monad stacks*. Monad transformers were a magnificent breakthrough. They strongly fortified in the Haskell community and were considered a state-of-the-art technique for a long time.

However, alternative approaches to the problem of the unstable denotational semantics of pure functional languages with effects were investigated. Cartwright and Felleisen proposed a formalism named *extended direct semantics* [12] almost at the same time with Moggi's monads. That formalism represents effects as a generalisation of exceptions: a computation may "rise" an effect, e.g. perform IO or mutate some kind of state, that must be later handled by a centralised *authority* (similar to exception handler). The idea of treating effects as exceptions was later studied more formally and the notions of *algebraic effects* [7] by Bauer and Pretnar and *effects handlers* [14] by Plotkin and Pretnar were introduced. These abstractions received much attention lately and several implementations were introduced.

Currently, the problem of construction of effectful computations have several main competing approaches: it is yet to analyse their simi-

larities and differences, to determine which one is better, or to find out if they are somehow the same.

1.3. Current state of the field

This section gives a short description of current state of computational effects: primary research directions and support of computational effects in programming languages.

The monadic approach is widely accepted in the functional programming community, especially in Haskell. Haskell even have a special syntactic sugar, *do-notation*, to make monadic code look more "imperative". This notation then gets de-sugared into a sequence of monadic *binds* and λ-abstractions. For computations with multiple side-effects, monad transformers are considered the standard technique: a lot of major Haskell libraries and applications are implemented on top of a monad-transformers stacks, for instance, monadic parser combinators library Parsec [15].

Despite the fact that monads are that popular in Haskell, there are implementations of alternative approaches. Kiselyov at al. presented *Extensible Effects* — an approach alternative to monad transformers for construction computations with several effects. Extensible effects are a continuation of Cartwright and Felleisen work [12] but based on the mathematical concept of a *free monad* — a method to derive a monad from an arbitrary functor. Extensible effects also may be considered a Haskell embedding of algebraic effects and effects handlers. This thesis explores applications of extensible effects to the construction of parser combinators libraries, see chapter 3.2 for details.

Brady's Idris [16] programming language has an effect system based on algebraic effects and effects handers. Idris is a strict dependently-typed purely-functional programming language, and its effect system heavily employs dependent types.

Bauer and Pretnar created and effect system for their ML-like language Eff [17]. The Eff itself is build around the concepts of algebraic effects and effects handlers.

One of the most recent implementations of algebraic effects and effects handlers is Frank programming language by Lindley at. al [9]. Frank is an experimental language with native support for algebraic effects, it has special syntactic constructions to define algebraic effects and their handlers, and is designed to make programming with algebraic effects natural. Frank is discussed in more detail in chapter 3.1 of this thesis, and its usage to implement parser combinators is explored.

Some studies were delivered to compare the expressive power of these approaches [18], which carried out the verdict that monad transformers may be converted into introduced *modular algebraic effects*: every *modular algebraic effect* gives rise to a monad transformer and vice-versa. However, the relative convenience of programming with these abstractions is yet to be explored.

In conclusion, it could be said that nowadays both monadic approach and algebraic effects with their handlers are popular to construct computations with multiple side-effects. Both abstractions have a wide range of implementations in programming languages and the problem of unification and finding an ultimate approach to rule all the effects has not yet been solved.

CHAPTER 2

MONADIC APPROACH TO COMPUTATIONAL EFFECTS

Monads were initially injected into programming languages context by Moggi as a tool to assign a denotational semantics to computational effects [2]. Later, Wadler adopted monads as a programming paradigm and introduced them into functional programming languages [3]. Monads are sometimes referred to as a "programmable semicolon" — a powerful way to construct sequences of computation with possible side effect. Afterwards, even more notions from category theory were given first-class support in modern programming languages, providing programmers with highly abstract, powerfully expressive and mathematically structured ways to build software.

2.1. Monads in Haskell

In Haskell programming language, monads are types that have an instance of `Monad` type class and satisfy three laws. They are used to distinguish pure computations from ones having some kind of side effect: mutable state, exceptions, non-determinism, etc.

The `>>=` operation, also known as *monadic bind*, represents a mentioned earlier "programmable semicolon". It takes a value in a monadic context as it first argument, an action that transforms that value as a second argument and returns a transformed value in the same monadic context.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b)   -> m b
  (>>)   :: m a ->  m b         -> m b
  return ::   a                 -> m a
  fail   :: String -> m a
```

Listing 2.1 — Monad type class

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Listing 2.2 — Monad laws

Monads have broad usage in functional programming. They are first-class citizens in purely-functional languages like Haskell and a wide range of Haskell-libraries have monadic interface. Mainstream programming languages also employ specific monads in a form of build-in language constructions, i.e. LINQ in C#or optionals in Swift.

As it was previously said, monads are used to characterise types of computations with a particular side effect. But what if a computation may potentially produce two or more effects? Then, means to combine several computational effects are needed. Monadic approach provide notion of *monad transformer* [5] — a type that may add properties of a given monad to any other. Monad transformers are widely used in Haskell to build computations carrying multiple side effects.

2.1.1. Monad transformers

Paper [5] describes a concept of a monad transformer — a building block for types describing computations with multiple side effects. Every transformer lets to add a particular effect, e.g. mutable state, configuration, exceptions, to a given monad. Transformers are put on top of a base monad to form a *monad stack* — a type characterising a computation with

multiple side effects. Consider an example of a function in a monad combining effects of mutable state and configuration (listing 2.3).

```
adder :: StateT String (Reader Int) Int
adder = do
  str <- get
  num <- ask
  return $ num + read str

adder' :: (MonadState String m, MonadReader Int m) => m Int
adder' = ...
```

Listing 2.3 — Effectful adder based on monad transformers

Here `adder` and `adder'` describe the same computation, but the first function is bounded to specific monad stack, while second just restricts effects that the stack ought to provide.

One characteristic of monad stacks is that ordering of monads is statically encoded in the type, so there is no runtime control of effect interaction. The Second problem of monad transformers is a need to write a lot of boilerplate type class instances, that is, to add a new effect, every possible combination of newly added effect with existing ones must be covered with instances to provide automatic lifting, thus $\mathcal{O}(n^2)$ instances must be written, where n is a number of monad transformers provided by the developing library.

And, finally, monad transformers do not provide a way to express computations that produce several homogeneous effects, e.g. two `State` effects, without sacrificing automated lifting.

Monad transformers are well established abstraction for building modular effectful computations. It has been widely accepted by Haskell community and a lot of useful libraries have been implemented on top of it. Nevertheless, it has some flaws regarding its convenience of use. Lately, algebraic effects and effects handlers — an alternative approach to structuring of effectful computations has emerged. It has received wide account in recent publications [7] [8] and has several implementations in

```haskell
class Monad m => MonadNew a m where
  action1 :: m a
  action2 :: m ()

instance MonadNew m => MonadNew (ExceptT e m) where
  action1 = lift action1
  action2 = lift action2

instance MonadNew m => MonadNew (IdentityT m) where
  action1 = lift action1
  action2 = lift action2

  ...
```

Listing 2.4 — Monad classes and instances for lifting

programming languages [8] [9]. The third chapter of this thesis gives a brief overview of two implementations: extensible effects library for Haskell and Frank programming language.

2.2. Monadic parsers

Consider a simple type to represent a parser.

```haskell
type Parser a = String -> Maybe (a,String)
```

Listing 2.5 — Basic parser type

In this representation, parser is a function from input stream to possibly non-present accepted result paired with the input stream remains. Types similar to `Parser` may be treated as effectful computation. One way to represent computations with effects in Haskell programming language is to use a concept of `Monad`. This particular type could be made an instance of `Monad` type class. Comprehensive information about properties of parsers like one presented above may be found in paper [4].

To make parsers more modular, extend capabilities and improve convenience of syntactic analysers, the `Parser` effect could be factorised into

a set of primitive effects. Moreover, the set of effect may be extended: it is handy to run parsers in a configurable environment or introduce logging. This section gives an overview to monadic monad transformers: a monadic framework for to build modular effectful computations. As an example, we give a sketch of monadic parser combinators library. We do not give the complete implementation here, because it's API is fairly similar to one of library implemented in Frank programming language and discussed in chapter 3.

2.2.1. Parser as a monad transformer stack

Monad transformer is a concept which lets to enrich a given monad with a property of other monad. Multiple monad transformers may be combined together to form monad stack, that is, a monad possessing all properties of it's components.

Monadic parser combinators library developed in this work also uses two-layer monad stack: state and either, where the last one provides effect of possibility of error. Thus, type for parser takes a following form.

```
newtype Parser t a = Parser (
    StateT (ParserState t) (Either (ErrorReport t)) a
  ) deriving ( Functor, Applicative, Monad
             , MonadState (ParserState t)
             , MonadError (ErrorReport t)
             )
```

Listing 2.6 — Parser monad stack

This representation of a parsers also is parametrised with type of input stream. Types `ParserState` and `ErrorReport` are algebraic data types for representing parser's state and possible analysis errors respectively.

The most low-level primitive which serves as a basis for all parser combinators is a parser that consumes a single item from input stream.

```
item :: TM.TextualMonoid t => Parser t Char
item = do
  state  <- get
  let s = TM.splitCharacterPrefix . remainder $ state
  case s of
    Nothing -> throwError (EmptyRemainder "item",state)
    Just (c,rest) -> do
      let (c,rest) = fromJust s
      put (ParserState {position = updatePos (position state) c
                        , remainder = rest})
      return c
```

Listing 2.7 — Unconditional single item consumer

More advanced parsers from developed library: conditional consumer and given string consumer.

```
sat :: TM.TextualMonoid t => (Char -> Bool) -> Parser t Char
sat p = do
  state <- get
  x <- item `overrideError` (EmptyRemainder "sat")
  if p x then return x else
    throwError (UnsatisfiedPredicate "general",state)
```

Listing 2.8 — Conditional consumer

To parse terminals it's convenient to introduce the parser that accepts a specific string.

```
string :: TM.TextualMonoid t => String -> Parser t String
string s = do
  state <- get
  (mapM char s) `overrideError`
    (UnsatisfiedPredicate ("string " ++ s))
```

Listing 2.9 — Parser for a static string

To actually perform parsing, it's necessary to implement a function that runs a computation. It's need to be pointed out, that order of effect handling is statically encoded in type of monad stack.

```
parse :: TM.TextualMonoid t =>
  Parser t a -> t -> Either (ErrorReport t) (a,ParserState t)
parse (Parser p) s =
  runStateT p (ParserState {remainder = s, position = initPos})
    where initPos = (1,1)
```

Listing 2.10 — Running the parser

2.3. Conclusion

Overall, a concept of monad transformers has a considerable convenience in programming due to its maturity and popularity. However, this approach lacks flexibility, doesn't permit stacks with several homogeneous effects (for instance, multiple `StateT` transformers) without losing automatic lifting (`lift`) and requires boilerplate type class instance declaration.

The next chapter considers different method of constructing parser combinators: one based on algebraic effects and effects handlers — an alternative framework of construction of effectful computation. We are going to present two prototype parser combinators libraries, one embedded into Haskell by means of extensible effects, and another implemented in Frank — an experimental programming language with native support foe algebraic effects.

CHAPTER 3

ALGEBRAIC EFFECTS AND EFFECTS HANDLERS

3.1. Algebraic effects in Frank

Algebraic effects and effects handlers provide an alternative to monads and monad transformers way to express effectful computations. Building parser combinators in term of algebraic effects and the process of parsing as their handlers is a solid model problem to find out strengths and weaknesses of this approach.

This section describes a prototype implementation of parser combinators library in experimental programming language Frank [9] which has first-class support for algebraic effects and effects handlers. We explore how parsers may be represented either by combination of multiple effects, for instance, mutable state and possible failure, or a monolithic effect signature. In conclusion, we make a note on expressive power of Frank's implementation of algebraic effects and handlers.

3.1.1. Building parsers as a combination of effects

This section employs Frank to build a prototype of parser combinators library.

Defining parser combinators

As noted in [4] simple parser could be expressed as a computation with two effects: state of input stream and a possibility of failure. Thus, handling parsing means handling a combination of those two effects, that is done by composing handlers for failure and state (see listing 3.1).

```
parse : {[Error, State (List Char)] X} -> (List Char) -> Maybe X
parse p str = catch (state str p!)
```

Listing 3.1 — Handling combination of state and failure

First parser that serves as a most basic building block in construction of more advanced ones is the *unconditional consumer*. It must take the first item of the input stream and yield it as a result, updating the state of the input stream with it's remains. In case of exhausted input, parser must fail. That is exactly the behaviour described by `item` function of listing 3.2.

```
item : [Error, State (List Char)] Char
item! = on get! { nil -> fail
                | (x :: xs) -> put xs; x}
```

Listing 3.2 — Parser consuming single item

Of course, unconditional consumption of the input stream without any actions doesn't make much sense. Actually, we would prefer consuming some items to others. Thus, *conditional consumer* that checks if an item satisfies a given predicate prior to consuming and fails otherwise, must be implemented (3.3).

```
sat : {Char -> [Error, State (List Char)] Bool} ->
      [Error, State (List Char)] Char
sat p = on item! {c -> if (p c) {c} {fail}}
```

Listing 3.3 — Conditional consumer

Having these basic building blocks, we are already able to construct practical parsers. A useful application of `sat` is the `char` parser that accepts a given character from the input stream (listing 3.4).

```
char : Char -> [Error, State (List Char)] Char
char c = sat {x -> eqChar x c}
```

Listing 3.4 — Parser for a given character

Besides accepting specific characters, `sat` parser could be used to implement other basic parsers. For instance, if predicate `isLetter` determining weather of not given character is Latin letter is defined, we could supply it to `sat` and acquire a parser for letters (listing 3.5). The same could be done for decimal digits.

```
letter : [Error, State (List Char)]Char
letter! = sat isLetter

digit : [Error, State (List Char)]Char
digit! = sat isDigit
```

Listing 3.5 — Parsers letters and digits

Now, being able to parse singular characters, we can make ourselves a task to accept sequences. That could be useful, for example, to parse terminals of some grammar. Here the main power of Frank's effect support comes in handy. As far as a string is essentially a list of characters, we can use the standard `map` function in presence of `Error` and `State (List Char)` *abilities*.

```
string : (List Char) ->
         [Error, State (List Char)] (List Char)
string str = map char str
```

Listing 3.6 — Parser for a given string

But individual chars and known-in-advance strings are not that interesting. Therefoerm we implement the combinators for alternatives and repetition.

Consider the case we have to parsers p_1 and p_2, and we would like to construct a parser that accepts all inputs that are recognisable by both

p_1 and p_2. In our setting we could implement this by deterministic choice: we apply p_1 and yield the result if it succeeds, otherwise we apply p_2 and initialise an error if it has failed, returning its result in case of success.

```
choose : {[Error, State (List Char)] X} ->
         {[Error, State (List Char)] X} ->
          [Error, State (List Char)] X
choose p1 p2 =
  on (parse p1 get!) { (right _)  -> p1!
                     | (left  _)  ->  on (parse p2 get!)
                       { (right _)   -> p2!
                       | (left err)  -> throw err
                       }
                     }
```

Listing 3.7 — Alternative combinator

For the simplest instance for `choose` usage, reconsider `letter` and `digit` parsers. We could combine those with the alternative combinator to accept alphanumeric characters (listing 3.8).

```
alphanum : [Error, State (List Char)]Char
alphanum! = choose digit letter
```

Listing 3.8 — Parser for alphanumerics

The motivation for repetition combinators is the need to apply an already defined parser multiple times with no certainty about the amount of required applications. Repetition combinators are useful for problems like parsing a sequence of statements of a programming language. In parser combinators approach, repetition combinators are usually defined as two mutually recursive functions: `many` accepts the result of zero or more applications of its argument-parser p and `some` succeeds if p is applicable at least once.

As an example of repetition combinator usage, consider parser for words — a sequence of letters (listing 3.10).

Besides plain sequences, it is very common to have sequences separated by some kind of marker, for instance in CSV files. Therefore, a repeti-

```
many : {[Error, State (List Char)]X} ->
       [Error, State (List Char)](List X)
many p = choose {some p} {nil}

some : {[Error, State (List Char)] X} ->
       [Error, State (List Char)](List X)
some p = p! :: many p
```

Listing 3.9 — Repetition combinators

```
word : [Error, State (List Char)] (List Char)
word! = some letter
```

Listing 3.10 — Parser for words

tion with separation combinator could be useful. It could be implemented on top of `many` combinator, see listing 3.11. Again, `sepby` accepts any, including empty, sequence, while `sepby1` requires at least one complete period.

```
sepby : {[Exception ParseError, State String]X} ->
        {[Exception ParseError, State String]Y} ->
        [Exception ParseError, State String](List X)
sepby p sep = choose {sepby1 p sep} {[]}

sepby1 : {[Exception ParseError, State String]X} ->
         {[Exception ParseError, State String]Y} ->
         [Exception ParseError, State String](List X)
sepby1 p sep = p! :: (many {sep!; p!})
```

Listing 3.11 — Repetition with separation combinators

The discussed combinators

Case-study: parsing simplified Markdown

As a usage example for the developed library, consider the parser of simplified Markdown-like language. We take into account only basic constructions to keep the example concise: the language is limited to headers,

plain paragraphs of text and unordered lists. The abstract syntax tree of the language is captured in Frank by the algebraic data type (listing 3.12).

```
data Document = Document (List Block)

data Block = Blank
           | Header (Pair Int Line)
           | Paragraph (List Line)
           | UnorderedList (List Line)

data Line = Empty | NonEmpty (List String)
```

Listing 3.12 — Simplified Markdown AST

Again, to keep this example simple and concise, we do not consider any in-line formatting like bold or italic cases. Therefore in-line elements are represented as plain strings of characters. The line of text is either an empty line or non-empty one, and we use `choose` combinator to represent this alternative (listing 3.13). An empty line is a sequence of spaces terminated with a newline character — exactly this is captured by `emptyLine` parser: we use `many` combinator supplying parser for a whitespace, then the parser for a newline character and, finally, the data constructor `Empty` of `Line` data type. To build a parser for a non-empty line, we need to refine the previous parser to be able to accept the meaningful contents. We use repetition with separation combinator `sepby` to parse a sequence of words separated by spaces and then return an accepted sequence wrapped up in the `NonEmpty` data constructor. The interesting thing here is the `let`-binding. We need to accept and than throw away the newline character that terminates the line, but at the same time we need to save the meaningful contents. Hence we parse it and bind to a name for later use in data constructor that gets returned. In Frank we work with effectful functions in the same way like we do with the pure ones; therefore syntactic sugar similar to `do`-notation becomes redundant.

The abstract syntax described in listing 3.12 has a data type to represent a block of a Markdown document. To be able to implement a parser for

```
line : [Exception ParseError, State String] Line
line! = choose emptyLine nonEmptyLine

emptyLine : [Exception ParseError, State String] Line
emptyLine! = many {char ' '};
             char '\n';
             Empty

nonEmptyLine : [Exception ParseError, State String] Line
nonEmptyLine! = many {char ' '};
                let ws = sepby word {char ' '}
                in char '\n'; NonEmpty ws
```

Listing 3.13 — Parsers for lines

blocks we must implement a parser for every possible block type: header, paragraph or unordered list.

```
block : [Exception ParseError, State String] Block
block! = choose header {choose paragraph unorderedList}
```

Listing 3.14 — Parser for a block of Markdown document

A Markdown header is a sequence of hash (#) symbols followed by a non-empty line of text. The number of hashes represents the level of importance of the header, that is, one hash stands for HTML tag <h1>, two hashes for <h2>, etc. Therefore, the parser needs to count the hashes and supply the number as an argument to the `Header` data constructor. Exactly that is done by `header` parser and, thanks to Franks applicative syntax for effects, we could supply the calls to inner parsers directly as the arguments to the resulting data constructor.

```
header : [Exception ParseError, State String] Block
header! = Header (length (some {char '#'})) line!
```

Listing 3.15 — Direct applicative style of defining parsers

Parsers for paragraphs and unordered lists are similar. We use repetition combinator `some` to accept a non-empty sequence of lines for a para-

graph and a non-empty sequence of lines prefixed with a star for unordered list.

```
paragraph : [Exception ParseError, State String] Block
paragraph! = Paragraph (some nonEmptyLine)

unorderedList : [Exception ParseError, State String] Block
unorderedList! = UnorderedList (some (char '*'; line))
```

Listing 3.16 — Direct applicative style of defining parsers

Finally, we are ready to implement a parser for the complete document as a non-empty sequence of blocks.

```
document : [Exception ParseError, State String] Document
document! = Document (some block)
```

Listing 3.17 — Parser for the document

To parse the document, we use the `parse` function that was defined earlier in this section, supplying the parser and the input string. If everything is correct, the resulting AST will be produced.

```
> parse document "# ToDo list\nThings to do today\n* Drink coffee
   \n* Write thesis\n"

right (Document ([Header 1 (NonEmpty (["ToDo", "list"])),
                  Paragraph ([NonEmpty (["Things", "to", "do", "
                     today"])]),
                  UnorderedList ([NonEmpty (["Drink", "coffee"]),
                                  NonEmpty (["Write", "thesis"])])
                                     ]))
```

Listing 3.18 — Using the parser

The implemented parser is a simple example of usage of parser combinators library implemented on top of algebraic effects and handlers abstraction. It gives a good example of usage of the facilities that Frank provides to build effectful computations.

3.1.2. Parsers as a standalone effect

The minimal parser combinators interface could be described with a monolithic algebraic effect. It could bring parsers the usual benefit of algebraic effects and effects handlers approach: independence of effect's commands and their interpretation. This was proved to be useful by Lindley [19] for parsers: a single set of parsing commands could be given different interpretations with idioms, arrows and monads, yielding parsers of different expressiveness.

As far as Frank has built-in support for algebraic effects and effects handlers, implementation of a monolithic effect for parsing could be a natural way to represent parsers in Frank.

```
interface Parser =
    fail : forall Y . Y
  | sat : {Char -> Bool} -> Char
  | choose : forall Y . {[Parser] Y} -> {[Parser] Y} -> Y
  | many : forall Y . {[Parser] Y} -> List Y
```

Listing 3.19 — Monolithic parsing effect

However, we came into a restriction of Frank's effects system, which makes it impossible to express the parser's signature as desired due to impossibility to construct effect interfaces incorporating commands returning values of different types. That is, every command of the interface must return a value of the same (maybe polymorphic) type, making an interface from listing 3.19 impossible to express.

3.2. Extensible Effects: algebraic effects embedded in Haskell

Paper [8] presents extensible effects — an alternative to monad transformers approach to typing computations with several interacting side effects.

The main idea of extensible effects is an analogy between effectful computations and client-server communication. An expression that is about to introduce some side effect: perform IO, throw an exception or something else like that must first make a *request* to some global authority which is in charge of system resources to handle this side effect. The request describes an effectful action that needed to be done and a continuation that must be executed after an action is performed.

In early variants of libraries similar to extensible effects, an authority that manages requests was a separate concept, like an operating system kernel, or IO-actions handler of GHC runtime. This manager possesses all the system resources (files, memory, etc.): it considers every request and makes a decision if it should be fulfilled or rejected. This external effect interpreter had a great power, but lacked flexibility.

More flexibility and modularity may be introduced with concept of algebraic effects and effects handlers [7], that inspired extensible effects. Thus, some major points of extensible effects:

- Effects handlers are parts of users program: somehow analogous to exception handlers. Every handler is authorised to manage effects of some part of program and produce effects by itself, which are going to be taken care of by some other handler.

- Effect typing system that tracks a type-level collection of effects active for every computation. This collection is an open union — a type-indexed coproduct of functors. An application of every handler affects the type: handled effect is excluded from collection. Therefore, it could be statically checked that all effects are handled.

- Extensible effects exploits a notion of free monad to build effectful DSLs. An instance of `Monad` type class provides programmer with set of familiar Haskell techniques such as `do`-notation and applicative programming.

One of the advantages of extensible effects in comparison to monad transformers is the absence of need in boilerplate type class instance declaration to perform lifting. And there is more: extensible effects permit computations with several similar effects without losing a possibility of automatic lifting. Consider an example of function with two readable environmental constants:

```
adder :: ( Member (Reader Int) r
         , Member (Reader String) r) => Eff r Int
adder = do
  num <- ask
  str <- ask
  return $ num + read str
```

Listing 3.20 — Effectful addition

Besides, extensible effects don't enforce an order of effects combination statically as monad transformers stacks do, thus giving a precise control of effects interactions in runtime. Next listing contains a computation and two handlers: first one doesn't preserve state in case of failure and returns `Nothing`, but second one does and returns `(0,Nothing)`.

```
countdown :: ( Member Fail r
             , Member (State Int) r) => Eff r ()
countdown = do
  state <- get
  if state == (0 :: Int) then die
  else put (state - 1) >> countdown

runCountdown1 n = run $ runFail $ runState (n :: Int) $ countdown

runCountdown2 n = run $ runState (n :: Int) $ runFail $ countdown
```

Listing 3.21 — Stateful computation

The rest of the section is dedicated to the canonical example of effectful computation: parser combinators. We use extensible effects to build a prototype of a parser combinators library and then give a small account to performance benchmarking. We do not give the complete implementa-

tion here, because it's API is fairly similar to one of library implemented in Frank programming language and discussed earlier in this chapter.

3.2.1. Building parsers with extensible effects

Extensible effects are essentially an embedding of algebraic effects and effects handlers into Haskell, therefore, the library has a lot in common with the Frank one, described in the previous section. Nevertheless, Haskell is a much more mature language, hence we are equipped with convenient features, that make our live easier.

We are going to represent parsers as computations with two side effects: possible failure and mutable state of an input string. Haskell's `ConstraintKinds` language extension allows us to a constraint alias that will help us to keep the type signatures of combinators concise:

```
type Parsable r = (Member Fail r, Member (State String) r)

type Parser r a = Parsable r => Eff r a
```

Listing 3.22 — The effects of parser

The `Parsable` constraint declares the effects performed by parsers: fallible computation and presence of state. Constraint `Member Fail r` points out that set of effects `r` must contain effect `Fail`, whereas type of return value `Eff r Char` tells that function `item` yields value of type `Char` and may perform effects from the list `r`.

Consider the basic primitive of the library — function that consumes a single item of and input string.

Generally, from syntactic point of view, declaration of the combinators based on extensible effects is similar to regular monadic code. This is achieved by type `Eff r a` having an instance of `Monad` type class. `Eff r a` is a free monad constructed on top of the functor `r` which is an open union of effects. As long as `Eff r a` is a monad, regular monadic donotation and applicative style become available.

```
item :: Parser r Char
item = do
  s <- get
  case s of
    [] -> put s >> throwError ()
    (x:xs) -> put xs >> pure x
```

Listing 3.23 — Single item consumer

Of course, unconditional consumption of the input stream without any actions doesn't make much sense. Actually, we would prefer consuming some items to others. Thus, *conditional consumer* that checks if an item satisfies a given predicate prior to consuming and fails otherwise, must be implemented (3.24).

```
sat :: (Char -> Bool) -> Parser r Char
sat p = do
  s <- get
  x <- item
  if p x then pure x else (put s >> throwError ())
```

Listing 3.24 — Conditional consumer

Consider the implementations of alternative (listing 3.25) and repetition (listing 3.26) combinators. We do not go in much details here because those combinators are quite similar to ones of the Frank library. The only difference is a need to write code in monad-flavoured style instead of direct applicative.

```
choose :: Parser r a -> Parser r a -> Parser r a
choose ma mb = do
  s <- get
  catchError ma $ \(ea :: ()) -> do
    put s
    catchError mb $ \(eb :: ()) -> throwError ()
```

Listing 3.25 — Alternative combinator

Extensible effects, in contrast to monad transformers, allow to set an order of effect handling just before running the computation. Thus, same

```
many :: Parser r a -> Parser r [a]
many v = many_v
 where
   many_v = some_v `choose` (pure [])
   some_v = (fmap (:) v) <*> many_v

some :: Parser r a -> Parser r [a]
some v = some_v
 where
   many_v = some_v `choose` pure []
   some_v = (fmap (:) v) <*> many_v
```

Listing 3.26 — Repetition combinators

computation may produce different behaviour, controlled by order of application of handlers. For instance, in next listing types of handlers `parse` and `parse'` are different because `parse` handles `Fail` after `State` and yields pair of last occurred state and possibly missing result of parsing, i.e. saves last state with no respect to success of parsing. Conversely, `parse'` handles `State` first and doesn't return any state in case of unsuccessful parsing.

```
parse :: Eff (Fail :> (State s :> Void)) a -> s -> (s, Maybe a)
parse p inp = run . runState inp . runFail $ p

parse' :: Eff (State s :> (Fail :> Void)) w -> s -> Maybe (s, w)
parse' p inp = run . runFail . runState inp $ p
```

Listing 3.27 — The parsing effects handler

3.3. Conclusion

The Frank programming language provides convenient and expressive features for programming with algebraic effects and handlers. Frank's native support for computations with multiple effects doesn't require programmes to deal with much of boilerplate. When the collection of effects of the computation is determined, there is nothing more holding the pro-

grammer from doing his job: no need to wrap one's head around complex types like monad transformers stacks and static effects ordering. In addition, direct applicative style of defining effectful computations often leaves it unnecessary to give names to intermediate results, thus making code easier to follow.

Representing parser as a computation combining several side effects is a triumph of modularity. Nevertheless, it somehow abuses one of the most important points about algebraic effects and effect handlers: the independence of syntax and semantics. Of course, parsing is perfectly representable by the combination of several algebraic effects, but it might also be interesting and useful to represent it as a separate effect signature to be able to assign different interpretations to the same parser's syntax.

Unfortunately, the current state of Frank's effect system doesn't provide a way to express effect interfaces required for building full-featured parser combinators as a monolithic effect. The system could be refined by introducing a possibility to incorporate GADT-like syntax into effect signatures.

Because Frank is in its infancy, it lacks some useful language features that are habitual from more mature functional languages. For example, there is no way to declare type synonyms; hence the effect list must be included in type signature of every parser combinator, making source code a bit more verbose than in could be.

Nevertheless, Frank is powerful is a very interesting research project, a proof-of-concept language that allows prototyping type-safe DSLs with precise control of side-effects.

Extensible effects are an embedding of algebraic effects and handlers in Haskell, thus the sketched parser combinators library has a lot in common with the one implemented in Frank (see section 3.1). However, as far as Extensible Effects are implemented on top of free monads, programmer must adopt the monadic style for definition of effectful computations and parsers implementation become a bit more wordy than in Frank. But from the other side, free monads let to Haskell programmers to keep the useful

syntactic sugar of do-notation and thus reuse a lot of the monadic code with minimal corrections.

Algebraic effects and handlers provide and alternative to monads approach to structuring of effectful computations. It still is not mature enough but the has received a lot of attention lately and was proven to have the same expressive power [18] while requiring less boilerplate in some cases.

CHAPTER 4

FUNCTIONAL PROGRAMMING AND COMPUTATIONAL EFFECTS AT SCALE

This chapter presents and discusses “Student’s Big Brother” — a distributed system to support study workflow in a programming class. It was designed to help teachers to distribute their attention to all students uniformly. The system consists of client daemons, watching students activity and sending their source code to a server in real-time (over HTTP), for storage, processing and displaying in teacher web-interface. The source code is freely available on GitHub [20].

The system is mostly implemented using Haskell programming language and the implementation extensively uses monad transformers and extensible effects — the concepts discussed previously in this thesis — to structure the necessary side-effects and separate the effectful functionality from the pure code.

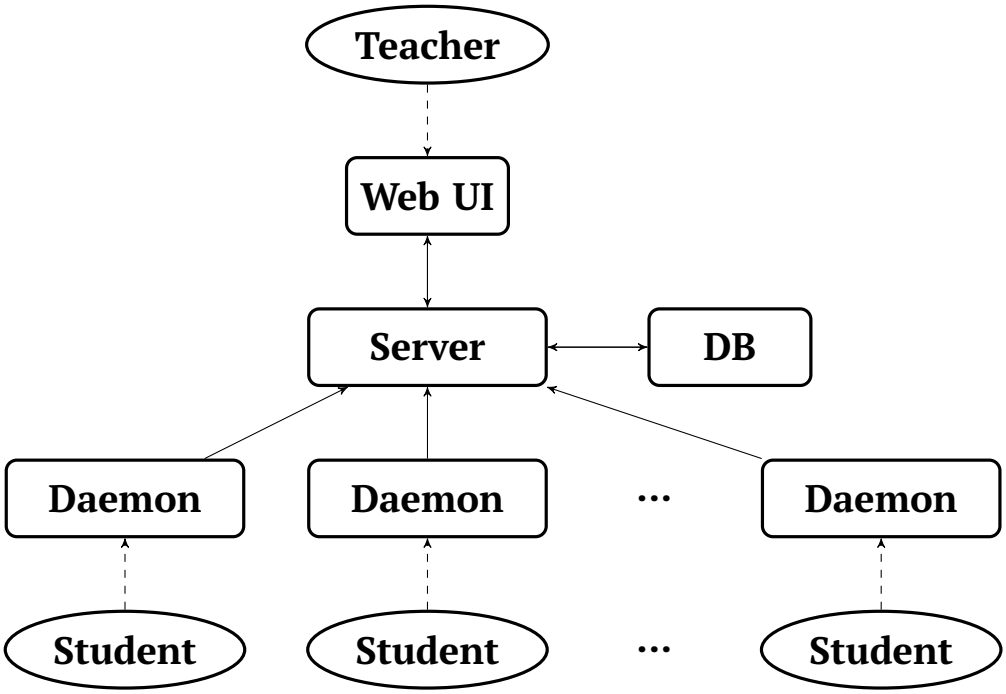
The rest of the chapter is structured as follows: the first section gives a short note on motivation for the system’s creation; the second section abstractly describes the system’s architecture; the third section focuses on implementation details: configuration files, integration and deployment and benefits of the use of functional programming; the last section reports the system usage in real-life teaching.

4.1. Motivation

It is unfortunate, but most undergraduate students hesitate to ask questions during programming practice sessions. Usually, the instructing teacher is fighting students' shyness by walking around the room, observing students and trying to determine whether the particular student is succeeding or not. But due to the limited time of the sessions and the fact that there is only one instructor per 10-15 students, the described procedure is quite inefficient. Besides, teacher's status checks may interrupt the string students who don't need the support. To approach the problem, we propose to build a real-time monitoring system of students activities during programming classes. This system should have features of basic version control and of displaying the students' source code in a lightweight web-interface for the instructor to be able to observe the activity of every student in real-time.

The primary purpose of the system is to distribute teachers attention between all the active students uniformly and to help the teacher to determine which one needs an assistance but hesitates to ask for it.

4.2. System's architecture



The system consists of three software components: the server, the client daemon and the teacher's web user interface (UI). The communication between all participants is performed over HTTP using JSON-formatted messages.

The rest of this section gives a more detailed account of the role of every component.

4.2.1. Daemon: data collector

The client daemon is a data collection agent. It is designed to run silently on a student's computer, observe student's activity and report it to a server.

The daemon's operation protocol is simple: it runs on a student's computer, watches the working directory for changes in source code files and regularly sends these changes to the application server for storage and distribution.

4.2.2. Server: data keeper and distributor

The server's role is to be the central authority for the daemons: collect the files the send, store them in a database and give them to the web UI on demand.

The server's code is essentially a CRUD-service [21] — it implements creation, removing, updating and deletion procedures of the considered entities — source code files — and maps these operations onto database procedures.

4.2.3. Web UI: data presenter

The web UI conveys the information about current state of students works to the teacher. It keeps up with the present state of server's database and provides a minimalistic interface to browse the files of every student separately.

4.3. Implementation details

The server and the client daemon are implemented with Haskell programming language and employ Haskell's type system to enforce safety and correctness of operation and communication. The teacher's web application is built with standard web stack: HTML, CSS and JavaScript.

4.3.1. Servant: type safety as a service with typelevel APIs and monad transformers

Advanced Type systems provide the facilities for *lightweight verification* of the programs: static type checking may help to find a lot of errors at compile time and hold them from leaking out into run time.

Servant [22] is a Haskell web-framework built around a type-level DSL for HTTP APIs description. The main idea is to represent APIs by

Haskell types thus making APIs *first-class* entities. This approach gives a huge boost to ensuring the correctness of API endpoints handlers [1] by statically checking their conformance to the desired API and consistency with each other. Besides ensuring the correctness of handlers, first-class API types provide means to automatically generate both documentation and client libraries (in Haskell, JS, etc.) from the API type. These features are implemented in Servant by means of *datatype-generic* programming [23].

The server-side part of "Student's Big Brother" is built on top of Servant. Consider a simplified version of the API type served for the student daemons (listing 4.1).

```
type StudentsAPI =
  "files" :> Capture "student_id" StudentId
          :> ReqBody '[JSON] [SourceFile]
          :> Post '[JSON] ()
  :<|>
  "register-student" :> ReqBody '[JSON] Student
                     :> Post '[JSON] Student
```

Listing 4.1 — Daemon's API

The type could seem a bit complicated, but it, in fact, is not. Let us take a closer look at it.

The daemon's API consists of two endpoints: `/files/:stundet_id` and `/register-student`. Note that `:<|>` type operator represents API "combination".

The first endpoint awaits a list of files of a particular student — a `POST` request with the body enclosing a `JSON`-encoded list of objects of type `SourceFile` (see listing 4.2).

The second endpoint is dedicated to the initialisation of the daemon and is similar to the first one.

Servant's type-level DSL for API specification makes use of modern GHC Haskell-extensions such as `DataKinds` (for type-level constants),

[1] Note that the notion of API handler is orthogonal to effects handlers from the previous chapter

```
data SourceFile =
  SourceFile { path              :: FilePath
             , contents          :: Text
             , modification_time :: ModificationTime
             } deriving (Eq, Show, GHC.Generic)
```

Listing 4.2 — Type to represent source files

`TypeOperators` and others to make the API types seem very similar to handlers that implement them.

Consider the handler for the discussed API (listing 4.5). The `(:<|>)` now appear on term-level and serve to combine handlers instead of APIs. The handlers are regular Haskell monadic computations making use of *monad transformers* to structure their *side-effects*: database IO, access to static configuration, and network communication. The HTTP request parameters are abstracted-out and represented as normal input parameters.

```
server :: ServerT API (ReaderT ServerConfig IO)
server =  updateFiles :<|> registerStudent

updateFiles :: StudentId -> [SourceFile] -> ReaderT ServerConfig
   IO ()
updateFiles uid files = do
  cfg <- ask
  dbConnection <- liftIO $ dbConnect $ db cfg
  liftIO $ dbUpdateFiles dbConnection uid files
  liftIO $ dbDisconnect dbConnection

registerStudent :: Student -> ReaderT ServerConfig IO Student
registerStudent student = ...
```

Listing 4.3 — Handler of daemon's API

Servant makes the web service implementation closely tied to its API specification by the ropes of Haskell's static type system. This makes impossible for API endpoints handlers to diverge from the specification and provides a no-overhead way to keep both documentation and client libraries up-to-date with the API.

4.3.2. Sharing types between client and server

Servant's control for convergence of API specification, server implementation and client code inspires for an introduction of even tighter type-channelled connection: the domain types may be factored out into separate Haskell module and reused for both client and server thus making it impossible for them to diverge.

In the implementation of the developed system, this is done to mentioned earlier `SourceFile` type thus making it's path be like the following. Firstly, the source code files read from the student's computer, packed into the `SourceFile` type, encoded to JSON and sent to the daemon over the network. Then the files got received by the server, and deserialized from JSON into the same `SourceFile` type. Moreover, the JSON serialisation and deserialization procedures are generated mechanically for the `SourceFile` using the GHC's support for datatype-generic programming thus no errors could occur due to a human mistake in the implementation of these procedures.

4.3.3. Extensible effects at the daemon's service

Servant is built with extensive usage of monad transformers thus making it unnatural to use a different side-effect control techniques with it. Nevertheless, the client code is not doomed to the usage of monad transformers; thus it was decided to use extensible effects to structures the side effects of daemon's implementation.

The client daemon is a computation with three effects: file system IO to read source code files; static configuration information about the server's domain name and port, student's credentials, etc.; and the mutable state effect to maintain the list of current student's files.

Consider the type signature of the main loop of the daemon and it's type signature (listing 4.4).

```
loop ::  ( Member (Reader ClientConfig) r
         , Member (State ClientState)   r
         , SetMember Lift (Lift IO)     r
         ) r => Eff r ()
loop = ...
```

Listing 4.4 — Daemon's effects

The `Member X r` constraint ensures the presence of the `X` effect in the *open union* `r`. The `SetMember` constraint is more restrictive and lets have only one effect of a particular kind (in this case, IO).

Extensible effects are a Haskell embedding of the algebraic effects and effects handlers approach to side-effects structuring. The `Eff r a` is essentialist a *free monad* over the functor `r` — an effect algebra (or signature, or interface). Extensible effects represent the effect signatures with *open unions* — a type-level collection on effect markers. The effect handler is a regular Haskell function of type similar to **`Monad`** `m =>Eff r a =>m a`. It gives a semantics for the effect algebra in some specific monad - it may be the IO monad or any other one (even a monad transformer stack).

For a daemon's main loop the handler may look like one presented in listing 4.5.

```
runApp :: Eff (Reader cfg :> State s :> IO :> Void) ->
          cfg -> s -> IO ClientState
runApp action initState cfg =
  runLift . runState initState . runReader action $ cfg
```

Listing 4.5 — Effects handler

Here, in the handler's type, we specify not only the constraints, but the exact list of effects arranged in particular order. Type operator `(:>)` here is used to compose effect algebras and the **`Void`** represents an *uninhabited* empty type with no constructors — an algebra with no operations. In fact, in most cases there is no need to specify such types explicitly — GHC may infer them.

We do not present the implementation of `loop` function here but it may be accessed and reviewed in the previously referenced GitHub repository [20].

4.4. Exploitation experience report

We have a successful experience of “Student’s Big Brother” integration into the teaching workflow. The system was used in an undergraduate programming course at I. I. Vorovich institute of mathematics, mechanics and computer science [24]. A teacher who has been leading the class reported the system to be reliable and the user experience to be satisfying. It was also reported that students had been showing more effort to complete the tasks and their hesitation for asking questions have been significantly reduced.

CONCLUSION

This thesis contributes to the development of approaches to effectful computation. Implementations of three parser combinators libraries based on three effectful computation frameworks are presented: one based on Haskell monad transformers [25]; another implemented with extensible effects [26] — an embedding of algebraic effects and effects handlers into Haskell; and a third one is a prototype implementation of a parser combinators library in Frank [27] — an experimental programming language with native support for programming with algebraic effects and effects handlers.

The developed libraries demonstrate advantages and disadvantages of the considered approaches. The thesis also gives an account to performance benchmarking of the monad transformers and extensible effects based libraries.

While building the prototype parsing library in Frank, we found out the limit of expressibility of algebraic effects in Frank: currently, it is impossible to declare effects which would have commands yielding results of different types; thus making impossible to implement full-fledged parser as a monolithic effect. A possible future work may include an extension of Frank's effect system to support existential quantification or GADT-like syntax for commands of effect interfaces.

The last chapter of the thesis describes an application of functional programming language with side-effects control to development of real-life software. We present the architecture and explain the implemen-

tation details of a distributed system for real-time student activity monitoring “Students Big Brother”’ [20]. We exploits the Haskell’s type and effect systems to make the implementation reliable and maintainable. The server-side share the domain types with the client code thus making it impossible fir them to diverge. The server-side effects is structured with monad transformers and the data collection daemon code uses extensible effects. We report our experience on how advanced type system features may improve the maintainability of a code base, make a development process more structured, and, as a result, lead to a reliable software.

BIBLIOGRAPHY

1. *Wadler P.* Propositions As Types // Commun. ACM. — New York, NY, USA, 2015. — Nov. — Vol. 58, no. 12. — Pp. 75–84. — ISSN 0001-0782. — URL: http://doi.acm.org/10.1145/2699407.
2. *Moggi E.* Notions of Computation and Monads // Inf. Comput. — Duluth, MN, USA, 1991. — July. — Vol. 93, no. 1. — Pp. 55–92. — ISSN 0890-5401. — URL: http://dx.doi.org/10.1016/0890-5401(91)90052-4.
3. *Wadler P.* The Essence of Functional Programming // Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Albuquerque, New Mexico, USA : ACM, 1992. — Pp. 1–14. — (POPL '92). — ISBN 0-89791-453-8. — URL: http://doi.acm.org/10.1145/143165.143169.
4. *Hutton G.*, *Meijer E.* Monadic Parser Combinators: Technical Report ; Department of Computer Science, University of Nottingham. — 1996. — NOTTCS-TR-96-4.
5. *Liang S.*, *Hudak P.*, *Jones M.* Monad Transformers and Modular Interpreters // Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — San Francisco, California, USA : ACM, 1995. — Pp. 333–343. — (POPL '95). — ISBN 0-89791-692-1. — URL: http://doi.acm.org/10.1145/199448.199528.

6. *Mcbride C.*, *Paterson R.* Applicative Programming with Effects // J. Funct. Program. — New York, NY, USA, 2008. — Jan. — Vol. 18, no. 1. — Pp. 1–13. — ISSN 0956-7968. — URL: http://dx.doi.org/10.1017/S0956796807006326.

7. *Bauer A.*, *Pretnar M.* Programming with algebraic effects and handlers // J. Log. Algebr. Meth. Program. — 2015. — Vol. 84, no. 1. — Pp. 108–123. — DOI: 10.1016/j.jlamp.2014.02.001. — URL: http://dx.doi.org/10.1016/j.jlamp.2014.02.001.

8. *Kiselyov O.*, *Sabry A.*, *Swords C.* Extensible Effects: An Alternative to Monad Transformers // SIGPLAN Not. — New York, NY, USA, 2013. — Sept. — Vol. 48, no. 12. — Pp. 59–70. — ISSN 0362-1340. — URL: http://doi.acm.org/10.1145/2578854.2503791.

9. *Lindley S.*, *McBride C.*, *McLaughlin C.* Do be do be do // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. — 2017. — Pp. 500–514. — URL: http://dl.acm.org/citation.cfm?id=3009897.

10. *Lukyanov G.*, *Pelenitsyn A.* Functional parser of Markdown language based on monad combining and monoidal source stream representation // Proceedings of the Tools and Methods of Program Analysis 2017 conference, TMPA'17, Moscow, Russia, March 4, 2017. — 2017. — URL: http://tmpaconf.org/program2017en#list-of-accepted-papers.

11. *Lukyanov G.*, *Pelenitsyn A.* Building parsers with algebraic effects // Proceedings of the Programming Languages and Compilers 2017 conference, PLC'17, Rostov-on-Don, Russia, April 4, 2017. — 2017. — Pp. 185–190. — URL: https://elibrary.ru/item.asp?id=29097517.

12. *Cartwright R.*, *Felleisen M.* Extensible denotational language specifications // Theoretical Aspects of Computer Software: International

Symposium TACS '94 Sendai, Japan, April 19–22, 1994 Proceedings / ed. by M. Hagiya, J. C. Mitchell. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. — Pp. 244–272. — ISBN 978-3-540-48383-0. — DOI: 10.1007/3-540-57887-0_99. — URL: http://dx.doi.org/10.1007/3-540-57887-0_99.

13. *King D. J.*, *Wadler P.* Combining Monads // Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992. — 1992. — Pp. 134–143.

14. *Plotkin G.*, *Pretnar M.* Handlers of Algebraic Effects // Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings / ed. by G. Castagna. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. — Pp. 80–94. — ISBN 978-3-642-00590-9. — DOI: 10.1007/978-3-642-00590-9_7. — URL: http://dx.doi.org/10.1007/978-3-642-00590-9_7.

15. *Leijen D.* Parsec, a fast combinator parser: tech. rep. ; Department of Computer Science, University of Utrecht (RUU). — Oct. 2001. — No. 35.

16. *Brady E.* Idris: Implementing a Dependently Typed Programming Language // Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP '14, Vienna, Austria, July 17, 2014. — 2014. — 2: 1. — DOI: 10.1145/2631172.2631174. — URL: http://doi.acm.org/10.1145/2631172.2631174.

17. *Bauer A.*, *Pretnar M.* An Effect System for Algebraic Effects and Handlers // Logical Methods in Computer Science. — 2014. — Vol. 10, no. 4. — DOI: 10.2168/LMCS-10(4:9)2014. — URL: https://doi.org/10.2168/LMCS-10(4:9)2014.

18. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control / Y. Forster [et al.] // CoRR. — 2016. — Vol. abs/1610.09161. — URL: http://arxiv.org/abs/1610.09161.

19. *Lindley S.* Algebraic Effects and Effect Handlers for Idioms and Arrows // Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming. — Gothenburg, Sweden : ACM, 2014. — Pp. 47–58. — (WGP '14). — ISBN 978-1-4503-3042-8. — URL: http://doi.acm.org/10.1145/2633628.2633636.

20. Student's Big Brother source code repository. — URL: https://github.com/geo2a/students-big-brother (visited on 06/10/2017).

21. *Martin J.* Managing the Data Base Environment. — 1st. — Upper Saddle River, NJ, USA : Prentice Hall PTR, 1983. — ISBN 0135505828.

22. Type-level Web APIs with Servant: An Exercise in Domain-specific Generic Programming / A. Mestanogullari [et al.] // Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming. — Vancouver, BC, Canada : ACM, 2015. — Pp. 1–12. — (WGP 2015). — ISBN 978-1-4503-3810-3. — URL: http://doi.acm.org/10.1145/2808098.2808099.

23. Generic Views on Data Types / S. Holdermans [et al.] // Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings. — 2006. — Pp. 209–234. — DOI: 10.1007/11783596_14. — URL: https://doi.org/10.1007/11783596_14.

24. Сайт ФИИТ. — URL: http://it.mmcs.sfedu.ru/ (visited on 05/12/2017).

25. Parser combinators on top of monad transformers. — URL: https://github.com/geo2a/markdown_monparsing/tree/experimental (visited on 06/11/2017).

26. Parser combinators on top of extensible effects. — URL: https://github.com/geo2a/ext-effects-parsers (visited on 05/11/2017).

27. Parser combinators in Frank. — URL: https://github.com/geo2a/frankoparsec (visited on 05/12/2017).

28. *Kiselyov O.*, *Ishii H.* Freer Monads, More Extensible Effects // SIGPLAN Not. — New York, NY, USA, 2015. — Aug. — Vol. 50, no. 12. — Pp. 94–105. — ISSN 0362-1340. — URL: http://doi.acm.org/10.1145/2887747.2804319.

29. Markdown parsers benchmarks. — URL: https://github.com/geo2a/md-parsers-benchmarks (visited on 05/12/2017).

Appendices

APPENDIX A

PARSERS PERFORMANCE BENCHMARKS

We performed the performance evaluation of the restricted Markdown-like language parsers implemented using the designed libraries.

The benchmarks compare three parsers: the first one is implemented with our monad transformers parsing library; the second parser is one build with extensible effects library; and the third is the Markdonw parser of the highly-popular and industry-recognised Pandoc document converter. The results of the benchmarking show that the first parser is the fastest, even faster that Pandoc. This is probably because of it's high specialisation. The extensible effects parser is dramatically slower than the monad transformers one. The problem of performance of free monads based programmes is addressed by Kiselyov at al. [28]. The free monads need to have more efficient implementation in order to compete with monad transformers for performance.

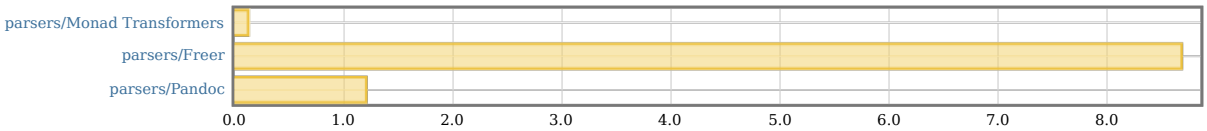
We do not benchmark the Frank parser combinators library because Frank is an experimental language and even doesn't have a compiler — only the type-checker and evaluator.

The rest of this appendix is a report generated by *criterion* — a Haskell benchmarking library. The report contains the mean values of the execution time, corresponding standard deviations and the execution time plots. The section "understanding this report" contains the detailed information on every entity in the report. The source code of the benchmarks is available on GitHub [29].

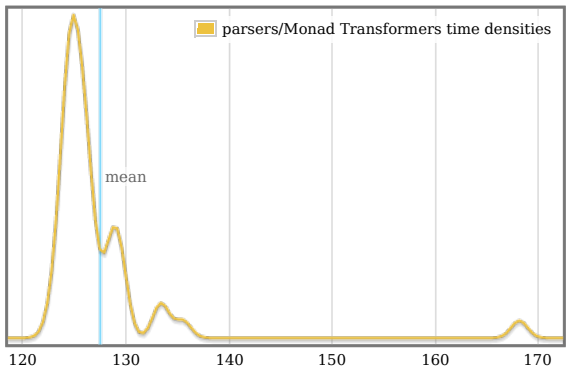
Markdown parsers performance measurements

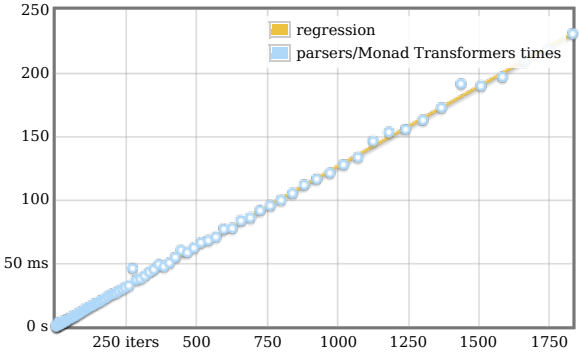
overview

The benchmarking was done on a Markdown file of size 355 bytes.



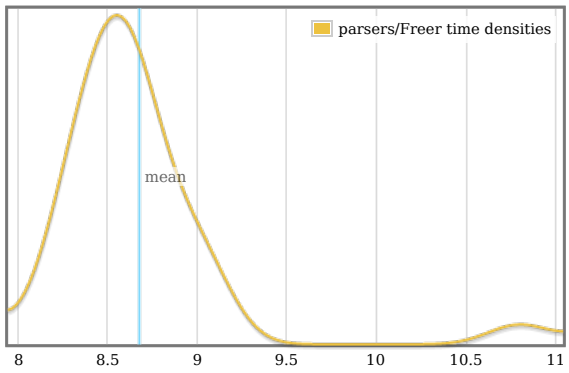
parsers/Monad Transformers

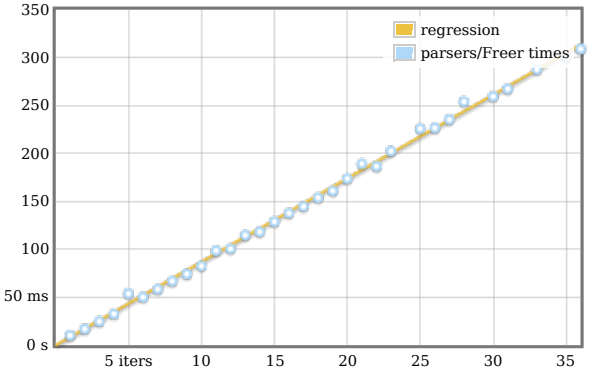




	lower bound	estimate	upper bound
OLS regression	125 µs	126 µs	128 µs
R² goodness-of-fit	0.998	0.999	1.000
Mean execution time	126 µs	128 µs	131 µs
Standard deviation	2.58 µs	6.95 µs	13.9 µs

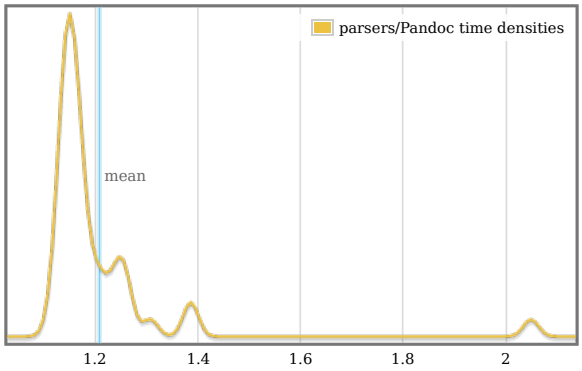
parsers/Freer

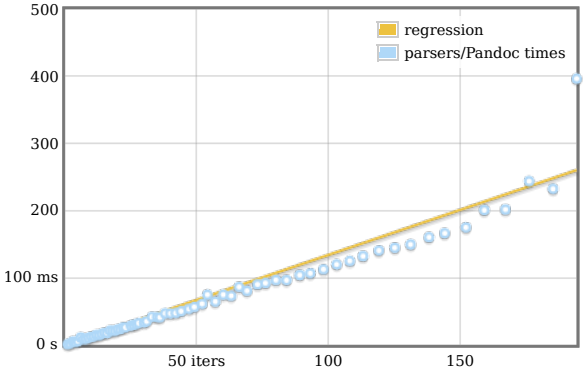




	lower bound	estimate	upper bound
OLS regression	8.58 ms	8.71 ms	8.90 ms
R² goodness-of-fit	0.996	0.998	0.999
Mean execution time	8.56 ms	8.68 ms	9.06 ms
Standard deviation	209 µs	466 µs	877 µs

parsers/Pandoc





	lower bound	estimate	upper bound
OLS regression	1.18 ms	1.34 ms	1.59 ms
R² goodness-of-fit	0.889	0.928	0.997
Mean execution time	1.18 ms	1.21 ms	1.30 ms
Standard deviation	57.1 µs	148 µs	291 µs

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x* axis indicates the number of loop iterations, while the *y* axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression prediction of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R² goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R² should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be. (Hover the mouse over the table headers to see the confidence levels.)

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.