Categorisation and implementation of build systems using algebraic effects

(Kwalifikacja i implementacja
systemów kompilacji z użyciem
efektów algebraicznych)

Jakub Mendyk

Bachelor thesis

Supervisor: dr Filip Sieczkowski

University of Wrocław
Faculty of Mathematics and Computer Science
Institute of Computer Science

Wrocław 2020

Abstract

Algebraic effects and handlers are a new way to deal with side effects. Build systems, despite being advanced computer programs that use sophisticated algorithms, are rarely the object of study. Recently, this has changed due to Mokhov et al. who in „Build systems à la carte" approach the subject in an abstract way, thus introducing categorisation and implementation of build system using programming language Haskell. The reader is introduced to the algebraic effects and handlers in an accessible way by presenting the subject both in theory and practice – the latter by using the programming language Helium. The crowning part of the paper is implementation of concepts introduced by Mokhov et al. in Helium. As the result, it is possible to compare those two implementations, and to see how programming with algebraic effects and handlers looks like.

Efekty algebraiczne i uchwyty to nowe podejście do ujarzmienia efektów ubocznych. Systemy kompilacji, choć są rozbudowanymi i wykorzystującymi skomplikowane algorytmy programami, nie cieszą się zainteresowaniem badaczy. Zmienia się to jednak za sprawą „Build systems à la carte" autorstwa Mokhov'a i innych, którzy podchodzą do tematu systemów kompilacji w sposób abstrakcyjny oraz przedstawiają ich kwalifikację i implementację z użyciem języka programowania Haskell. Praca w przystępny sposób wprowadza czytelnika do zagadnienia efektów algebraicznych i uchwytów. Zostają one opisane w teoretyczny, a dzięki wykorzystaniu języka programowania Helium, także w praktyczny sposób. Zwieńczeniem pracy jest powtórzenie implementacyjnych wyników Mokhov'a i innych korzystając z języka Helium. W konsekwencji możliwe jest porównanie obu implementacji oraz zaobserwowanie jak wygląda programowanie z efektami algebraicznymi i uchwytami.

Contents

Chapter 1

Introduction

1.1 Troubles with computational effects

Computer programs – thanks to their ability to interact with external resources such as storage, computer networks or human users – can do way more than just perform predefined computations. However, this causes their runtime behaviour to depend on the external world in which these resources live and makes the programs' not just sequences of pure computations but also accompanying side effects.

Moreover, computational effects make it substantially harder to understand and reason about programs' behaviour and their correctness which limits their modularity and leads to more frequent introduction of mistakes and bugs by the authors. To avoid such issues, it is often attempted to split the programs into pure and impure parts while minimising the size of the latter. Despite these efforts, it is not trivial to tell if certain module of a program performs only pure computations and we often have to resort to trusting the author that it is true indeed.

1.2 Dealing with computational effects

One of the methods of solving that problem is inserting information about side effects into the type system. We can then use mechanisms of type inference and verification for automatic identification of functions that are not pure – making it easy for a programmer to tell, from the function's signature, which effects might appear during function's runtime. Well known example of such solution are monads in Haskell programming language. Unfortunately, concurrent use of two independent resources represented by different monads is non-trivial and requires additional structures, such as monad transformers, which bring in additional challenges – the program of modularity has only been shifted into other space.

On the other hand, there is a new attempt to deal with side effects with the

help of type systems called algebraic effects and handlers. From the surface, they seem similar to exceptions known in many programming languages or system calls in operating systems. However, due to the split between definitions of effectful operations and their semantics, and an interesting use of continuations, they give the programmer ease of thinking and reasoning about the programs using them. And in contrast to monads, multiple algebraic effects can be freely used at once.

1.3 Build systems

To find a fine example of computer programs, whose main task is interaction with external resources, you can look no further than at build systems. There the user provides a set of mutually-depended tasks with information how to execute each of them by using results of other tasks, and the systems is responsible for their correct ordering and execution. Furthermore, we expect the build system to track changes in inputs and – when asked to update the results – rerun only the tasks whose output values will change. An examples of such systems are Make and – which might seem surprising – office applications for editing spreadsheets (such as popular Excel).

In recent publications titled „Build systems à la carte" [1, 2], authors introduce a method of categorising build systems by taking into account how they determine order of task execution and in which way tasks are identified as requiring a rebuild. Presented categorisation leads the authors to introduction of a framework for creating build systems with expected properties which happens to be easily implementable in Haskell. What is more, it turns out that Applicative and Monad type classes correspond to the possible level of complexity of dependencies between tasks.

1.4 About this paper

This work aims to introduce the reader – who has experience with Haskell and basics of functional programming languages – to the innovative solution which are algebraic effects and handlers, and to demonstrate – by following Mokhov et al – an implementation of build systems using algebraic effects and handlers in a Helium programming language. As the result, it is possible to compare those two implementations, and to see how programming with algebraic effects and handlers looks like.

In the second chapter, we introduce simple and informal model of calculations that uses algebraic effects and handlers. Some examples of representing standard computational effects in our model are described.

Chapter three serves as an introduction to „Build systems à la carte", discussion of observations made be the authors and description of abstraction of build systems and their consequences. Contents of the source publication is presented on a level sufficient to understand implementation of build systems with algebraic effects and

handlers in chapter five. At the same time, the reader is encouraged to get acquainted with full contents of the work by Mokhov et al because it is an interesting and easy to read publication.

The fourth chapter begins with enumeration of existing programming languages and libraries that enable programming with algebraic effects and handlers. Next, the reader is introduced to the Helium programming language and presented with sample problems and their solutions that use effects and handlers. Moreover, use of multiple effects at the same time is demonstrated – which is significantly easier than with monads in Haskell.

Finally, the crowning fifth chapter contains implementations of schedulers, rebuilders and build systems inspired by the results of „Build systems à la carte". However, this time in a programming language with algebraic effects and handlers. Presented are the differences between abstract types from which the implementations are derived, and how use of effects and handlers influence the final form. Showcased result is missing one of the schedulers, thus and explanation is provided why it is so.

Chapter 2

Algebraic effects in theory

We will introduce notation serving as a description of simple computation, which will help us – with diving deep into it's mathematical origins – understand how simple and fascinating at the same time are algebraic effects and handlers. Presented notation is purposefully informal, since it should present abstract description of computations with effects in an accessible way without introducing concrete programming language.

Next, we will see how popular examples of side effects can be expressed using our notation. Finally, reader will be presented with resources for further reading, which extend the information presented in this chapter.

2.1 Notation

We will consider computations on values of following types:

- booleans B – with values T, F, and standard logical operators,
- integers $\mathbb{Z}$ – with equality checking and basic arithmetic operations,
- unit type U – inhabited by single value u,
- and pairs of these types.

Our model will consist of expressions:

- **return** v – where v is an expression with booleans or integers,
- **if** $v_1 = v_2$ **then** e_t **else** e_f – conditional expression, where $v_1 = v_2$ is a check for equality of two integer expressions,
- abstract operations $\{op_i\}_{i \in I}$ – causing side effects – whose behaviour is not specified, and their signatures are $op_i : A_i \to (B_i \to C_i) \to D_i$, where A_i, B_i, C_i and D_i are some types. Expression $op_i(n, \kappa)$ represents and operation with

argument n and further part of the computation κ parameterised by the result of the operation, which *may* be executed by the operation,

- handlers, that is expressions of the form **handle** e **with** $\{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}_{i \in I}$, where e is some other expression; handler defines the behaviour of (until now abstract) operations.

For instance, these are examples of computations in our notation:

$$\begin{gathered} \textbf{return}\ 0, \quad \textbf{return}\ 2+2, \quad op_1(2, \lambda x.\ \textbf{return}\ x+1) \\ \textbf{handle}\ op_1(2, \lambda x.\ \textbf{return}\ x+1)\ \textbf{with}\ \{\ op_1\ n\ \kappa\ \Rightarrow\ \kappa\ (2 \cdot n)\ \} \end{gathered} \tag{2.1}$$

For increased readability, if set in a handler doesn't include all operations, we assume that the handler doesn't define behaviour of the operation; equivalently, the set is extended with $op_i\ n\ \kappa\ \Rightarrow\ op_i(n, \kappa)$.

We won't explicitly define evaluation of expressions, instead we will describe some equalities that will hold:

- $(\lambda x.\ e_1)\ e_2 \equiv e_1\,[{}^{x}/_{e_2}]$ – application of an argument to function,
- **if** $v_1 = v_2$ **then** e_t **else** $e_f \equiv \begin{cases} e_t & \text{when } v_1 \equiv v_2 \\ e_f & \text{otherwise} \end{cases}$
- **handle return** v **with** $H \equiv$ **return** v – handler doesn't modify behaviour of pure expressions,
- **handle** $op_i(a, \lambda x.\ e)$ **with** $H \equiv h_i\,[{}^{n}/_{a},\ {}^{\kappa}/_{\lambda x.\ \textbf{handle}\ e\ \textbf{with}\ H}]$, where $H\ =\ \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}$.

Using above equalities, let's simplify expression from the last example:

$$\begin{aligned} &\textbf{handle}\ op_1(2, \lambda x.\ \textbf{return}\ x+1)\ \textbf{with}\ \{\ op_1\ n\ \kappa\ \Rightarrow\ \kappa\ (2 \cdot n)\ \} \equiv \\ &\textbf{handle}\ (\lambda x.\ \textbf{return}\ x+1)(2 \cdot 2)\ \textbf{with}\ \{\ op_1\ n\ \kappa\ \Rightarrow\ \kappa\ (2 \cdot n)\ \} \equiv \\ &\textbf{handle return}\ 4+1\ \textbf{with}\ \{\ op_1\ n\ \kappa\ \Rightarrow\ \kappa\ (2 \cdot n)\ \} \equiv \\ &\textbf{return}\ 5 \end{aligned} \tag{2.2}$$

2.2 Equations, failure effect and mutable state

Up until now we didn't assume anything about behaviour of effectful operations. Handlers could work in any possible way. Let's limit ourselves and define conditions for handlers of selected operations. For example, let's say that for operation op_r, all handlers must satisfy following equality:

$$\forall n\ \forall e.\ \textbf{handle}\ op_r(n, \lambda x.\ e)\ \textbf{with}\ H \equiv n \tag{2.3}$$

We notice that there exists only one intuitive handler satisfying this equality, this handler is: $H = \{\ op_r\ n\ \kappa\ \Rightarrow\ n\ \}$. What is more, it's behaviour is suspiciously similar to the construct of exceptions in popular programming languages:

```
try {
  raise 5;
  // ...
} catch (int n) {
  return n;
}
```

The similarity is fully intended. As it turns out, our language with single operation and an equation has enough power to express the construct that in other languages is impossible to create by a programmer and has to be included by the author of a language.

Let's consider next example. For readability sake, In the next example, for readability sake instead of naming operations op_i we will use more meaningful names: *get* and *put*. These operations have signatures $get : U \to (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$, $put : \mathbb{Z} \to (U \to \mathbb{Z}) \to \mathbb{Z}$. Let's define equalities that will make *get* and *put* behave like operations on mutable memory cell:

- $\forall e.\ get(u, \lambda\_.\ get(u, \lambda x.\ e)) \equiv get(u, \lambda x.\ e)$

  subsequent reads from the call without modifying it give same results,

- $\forall e.\ get(u, \lambda n.\ put(n, \lambda u.\ e)) \equiv e$

  putting the value that is already there doesn't influence the result,

- $\forall n.\ \forall f.\ put(n, \lambda u.\ get(u, \lambda x.\ f\ x)) \equiv f\ n$

  computation that reads from the cell gives the same result as if cell's value was provided directly as an argument,

- $\forall n_1.\ \forall n_2.\ \forall e.\ put(n_1, \lambda u.\ put(n_2, \lambda u.\ e)) \equiv put(n_2, \lambda u.\ e)$

  the cell behaves as if it remembered only the most recent value.

Let's notice that while we limit possible external behaviour of operations *get* and *put*, handler's authors have freedom in implementing how the operations will behave internally.

2.3 Looking for success

The next example of side effect, that we will consider in this chapter, is nondeterminism. We would like to be able to express computations in which some parameters

can have many values and their choice should be made in a way to satisfy some given conditions. For example, we have three variables x, y and z and we would like to check if logical formula $\phi(x, y, z)$ is satisfiable. For this purpose we will define operation $amb : U \to (B \to B) \to B$ that causes the side effect of nondeterminism. Let's write and expression that will solve our problem:

$$\begin{aligned} &\textbf{handle } amb(u, \lambda x.\ amb(u, \lambda y.\ amb(u, \lambda z.\ \phi(x, y, z)))) \\ &\textbf{with } \{\ amb\ u\ \kappa\ \Rightarrow\ \kappa\ (T)\ \textbf{or}\ \kappa\ (F)\ \} \end{aligned} \tag{2.4}$$

When we defined failure effect, the computation wasn't continued. In the case of nondeterminism we continue twice, substituting for the nondeterministic variable different values. This way, we can legibly check all possible valuations, and as the result determine if the formula is satisfiable.

One can notice, that if we wanted to check is formula is a tautology, it would suffice to replace only a single word – **or** with **and**, thus getting a new handler:

$$\begin{aligned} &\textbf{handle } amb(u, \lambda x.\ amb(u, \lambda y.\ amb(u, \lambda z.\ \phi(x, y, z)))) \\ &\textbf{with } \{\ amb\ u\ \kappa\ \Rightarrow\ \kappa\ (T)\ \textbf{and}\ \kappa\ (F)\ \} \end{aligned} \tag{2.5}$$

Introduced construction of effects, operations and handlers creates dual mechanism, in which operations are producers of effects, and handlers are their consumers. By taking away from operations their semantic meaning, or only requiring satisfaction of simple equations, we got an incredibly powerful tool that let us – in a simple, declarative way and crucially (in contrast to popular programming languages) single-handedly – define own advanced side effects.

2.4 Further reading

The intent of this chapter was to accessibly introduce ideas, definitions and constructions related to the subject of algebraic effect and handlers, which will be a foundation for understanding practical examples and implementation of build systems in the next chapters. Readers that are interested in diving deeper into the history and origin of algebraic effects and handlers are advised to get acquainted with following materials:

- „An Introduction to Algebraic Effects and Handlers" by Matija Pretnar [3],
- notes and series of lectures by Andrej Bauer title „What is algebraic about algebraic effects and handlers?" [4] available as both text and videos on YouTube,
- works of Plotkin with Power [5, 6], and with Pretnar [7] – if the reader wants to see one of the first results leading to algebraic effects and usage of handlers,

- community gathered around the subject of algebraic effects accumulates related resources in a repository [8] on GitHub.

Chapter 3

About build systems (and their categorisation)

Build systems – even though being used in almost all software project – are often neglected by their users, treated as a necessary evil, sometimes even are a source of anxiety and anger. Despite their popularity and – greater than one would expect – level of complexity, they have rarely been an object of study by scientists. However, they became a central object of interest in a recent publications by Andrey Mokhov, Neil Mitchell and Simon Peyton Jones titled „Build systems à la carte" [1] and „Build systems à la carte: Theory and practice" [2]. In this chapter we will follow their steps and discuss the results they got, in order to implement presented build systems using a language with algebraic effects and handlers later in this paper.

3.1 Examples of build systems

In order to understand deeper, non-trivial relations and similarities between build systems, let us look at a few examples of such systems used in the wild.

3.1.1 Make

Make is a very popular, widely available and relatively old build system. It is configured using files called *makefiles*, which define tasks, dependencies between them and how each task should be built. Let us consider example configuration for a simple program written in C.

Example configuration for Make

```
util.o: util.h util.c
    gcc -c util.c

main.o: util.h main.c
    gcc -c main.c

main.exe: util.o main.o
    gcc util.o main.o -o main.exe
```

Above configuration defines build rules for three tasks: *util.o*, *main.o* and *main.exe*. The line defining the tasks includes information about other tasks that the current one depends on, for instance *util.o* depends on tasks (here: files) *util.h* and *util.c*, and the task is built by executing *gcc -c util.c*. If some task doesn't have build rule defined, for example *util.h*, we will call it an input (task) in such configuration.

All information about dependencies between tasks are defined in single *makefile*. When the user wants to build task *main.exe*, they execute the program using command *make main.exe*. Then, the system will determine which tasks need to be updated and built in order to satisfy user's request. Since the procedure of build tasks is the same, independently from the results of subtasks, we will say that the build system has static dependencies. For such systems, a natural ordering of tasks in which they should be built is a topological order. This way each task will be built using „fresh" values of its dependencies. Otherwise, there could be a need to execute some tasks multiple times.

Let us notice, that during subsequent builds, system may not need to rebuild some tasks if values of their inputs didn't change. This observation leads us to the concept of minimality, which is defined by the authors as:

Definicja. (Minimality) A build system is minimal if it executes tasks at most once per build, and only if they transitively depend on inputs that changed since the previous build.

For determining which tasks should be rebuilt, Make uses file modification times. If the file corresponding to the task is older than file of one of its dependencies, the task should be rebuilt.

We should notice that for come configurations topological order of tasks might not exist, because there are circular dependencies between tasks – we will assume that is not a case from now on.

3.1.2 Excel

It may seem surprising, but we can think about spreadsheets (for example in Excel) as build systems. Cells, whose values are provided directly are input tasks, while formulas for the other cells are build instructions. With that interpretation,

spreadsheets become an interesting and useful example of build systems.

Let us consider an example of a spreadsheet document presented by the authors of the original paper:

A1: 10 B1: INDIRECT(“A" & C1) C1: 1
A2: 20

Function INDIRECT dynamically determines from which cell value will be read, and operator & is just string concatenation. When C1 = 1, value of the cell B1 will be the same as A1, and when C1 = 2, the value will be read from A2. As we can see, cells whose values are used to compute value of B1 depend on the value of C1. In such cases we say the task has (and more generally system supports) dynamic dependencies. Here we only have one level of indirection because dependencies of B1 are chosen by C1. However, in a more general case level of indirection can by arbitrarily large. As the result, solution based on topological sorting is not suitable because we can't determine the dependencies beforehand – without checking values of cells.

Ordering of cells during build is a little more complicated in Excel. The systems maintains a sequence of cells (called chain). During the build process Excel build cells in the order determined by the chain. When cell A depends on other cell N, which has not been built yet, Excel performs a restart – stops execution of task A, shifts N in front of A in the chain and resume building starting from cell N. When the build is finished, resulting chain has an interesting property, that when the build process is executed again without changing inputs, the build will finish without restarts. The chain acts as an approximation of proper task ordering. In order to determine, which cells needs to be rebuilt, Excel maintains a dirty bit for each cell. Cell is dirty, when:

- it's an input whose value has changed,
- it's formula has changed,
- contains formula with a function that prevents static determining of dependencies – such as INDIRECT or IF.

It is easy to see, that Excel is not a minimal build system, because it over-approximates which cells need to be rebuilt. However, Excel tracks not only changes in inputs but also tasks definitions (that is formulas), which is a rare property in the world of build systems called self-tracking. Usually, change of task definition requires performing full build process by the user.

3.1.3 Shake

Shake is a build system, in which tasks are defined as tasks in a domain specific language embedded in Haskell. It supports dynamic dependencies and (in contrast to Excel) is minimal.

Instead of constructing a chain, Shake generates dependency graph at runtime. In the case of building task whose dependency is not yet ready, Shake suspends building of current tasks and begins build of the required tasks. When it is done, suspended task is resumed since it's dependency is now finished.

Other property that Shake has is ability to perform early cut-off – when some task has finished rebuilding but its value hasn't changed, there is no need to force rebuild of tasks that depend on this one. Make and Excel don't have such optimisation.

3.1.4 Bazel

The last example of a build system is Bazel, which was created as an answer to the growing need from large teams working on software projects of substantial size. In such project many people might build the same code fragments, which leads to waste of computational resources and time of programmers.

Bazel is a cloud build system – when the user wants to perform a build, the systems connects to the server and checks, inputs of which tasks hasn't change and if they have already been built by someone else. Bazel saves time of the programmer by downloading results of such tasks. Moreover, since single programmer usually performs modifications in only couple of modules, outputs of many tasks don't change and only a small part of them need rebuilding.

The systems tracks the changes by checking hashes of source files. When checksum of a file on computer and on the server differ, task is considered out-of-date and rebuilt. Next, the result and new checksum are stored on the server, functioning as a „cache" for build results.

Bazel currently doesn't support dynamic dependencies. During the build process it uses the restarting mechanism, and for determining which tasks need rebuilding in stores values and it's hashes of task along with history of executed build commands.

3.1.5 Conclusions

Four discussed build systems presented different degrees of freedom given to task authors. We learned about mechanisms used for building tasks and optimisations, which decrease the number of tasks that are unnecessarily built. Their usage enable some of the systems to achieve minimality.

3.2 Build systems abstractly

After discussing current state of affair, authors introduce us to the terminology and abstract representation of the space of build systems.

3.2.1 Terminology

An object on which build systems work is called a store, which assigns values to keys. In the case of Excel it is a spreadsheet document, and for Make it is a filesystem. The goal of the build system is modifying the store in order to bring the value associated with the key provided by the user up to date. System has a memory in the form of persistent information for the needs of later executions. The user provides description of tasks in the form of instructions explaining how each task should be built using outputs of other tasks.

Build system receives task definitions, a store and a key. After the build is finished, the value in store associated with the key should be up to date.

3.2.2 Store and tasks

Authors introduce following abstract representation of task and tasks (as a set of their definitions):

```
newtype Task c k v = Task (forall f. c f => (k → f v) → f v)
type Tasks c k v = k → Maybe (Task c k v)
```

The task is parameterised by a type of the key k and return value v. It computes its value using provided function used for getting values of other tasks. As we can see, the value is not provided directly, but instead in an unknown carrier f, belonging to class c. An examples of classes are *Applicative* and *Monad*.

A group of tasks is a function, which maybe associates a key with a definition of how to build a task identified by that key. Input tasks don't have definitions associated with their keys, and their values are accessed from the store. For instance, following example of spreadsheet document:

A1: 10	B1: A1 + A2
A2: 20	B2: 2 * B1

can be expressed using our abstraction as:

```
sprsh1 :: Tasks Applicative String Integer
sprsh1 "B1" = Just $ Task $ \fetch → ((+)  <$> fetch "A1" <*> fetch "A2")
sprsh1 "B2" = Just $ Task $ \fetch → ((*2) <$> fetch "B1")
sprsh1 _    = Nothing
```

The store is an abstract type parameterised by types of keys, values and persistent build information:

```
data Store i k v
initialise :: i → (k → v) → Store i k v
getInfo :: Store i k v → i
putInfo :: i → Store i k v → Store i k v
getValue :: k → Store i k v → v
putValue :: Eq k => k → v → Store i k v → Store i k v
```

Authors define basic operations on a store for constructing it, and for getting and setting persistent information and values of keys.

3.2.3 Build systems

Type of a build systems comes directly from its definition – it gets tasks, a store and a key:

```
type Build c i k v = Tasks c k v → k → Store i k v → Store i k v
```

Let us consider an implementation of a simple build system expressed using our abstraction:

```
busy :: Eq k => Build Applicative () k v
busy tasks key store = execState (fetch key) store
  where
    fetch :: k → State (Store () k v) v
    fetch k = case tasks k of
        Nothing   → gets (getValue k)
        Just task → do v <- run task fetch
                       modify (putValue k v)
                       return v
```

System *busy* starts evaluation of a task in context of mutable state, which is used to remember values of built tasks. When the task is to be built, and it is an input task, its value is read from the store. Otherwise, the task is executed according to its definition. This build system, like the ones we will see later, mainly consists of the definition of function *fetch*, which determines its operation. The *busy* system is obviously not minimal, however it is correct and is a good starting point for creating true build systems.

This systems can be easily executed on sample store, which will be a dictionary implemented as a function – this way we can set a default value for all input tasks:

```
> store = initialise () (\key → if key == "A1" then 10 else 20)
> result = busy sprsh1 "B2" store
> getValue "B1" result
30
> getValue "B2" result
60
```

The system works and returns valid results. We can also see, that the existential qualification of f from the task definition is useful:

```
newtype Task c k v = Task (forall f. c f => (k → f v) → f v)
```

In this case `c = Applicative` and `f = State (Store () k v) v`, thus function `fetch` can perform side effect of modifying a mutable state which wraps the store.

3.2.4 Polymorphic task

Wrapping a result value let us perform side effects, and existential quantification gives the author of the build system freedom to choose a structure, which will be right for their need. In the case of `busy` it was mutable state for holding the state.

If `f` was completely arbitrary, nothing useful could be done with it, thus we require it to belong to some class `c`. What is surprising, such requirement defines how complicated dependencies between tasks can be. Let us consider three popular (and one extra) type classes in Haskell:

- `Functor` – gives possibility to apply functions to wrapped values. In visual terms – with functor we perform sequence of computations that modify a value.
- `Applicative` – enables merging of multiple values using a provided function. In this case, computations are directed acyclic graphs.
- `Monad` – we get an arbitrary graph, which can dynamically change (based on result values). Here we can inspect wrapped values and make decisions based on that.
- `Selective` [9] – is an intermediate form between applicatives and monads. It lets us perform decisions based on results, but the decision's options are known statically.

The authors come to an unexpected conclusion: tasks in which type `f` is an applicative can only have static dependencies, while dynamic dependencies are possible when `f` is a monad!

Thus, an example with INDIRECT in Excel – by using our abstraction – can be expressed in Haskell like this:

```
sprsh3 :: Tasks Monad String Integer
sprsh3 "B1" = Just $ Task $ \fetch → do
    c1 <- fetch "C1"
    fetch ("A" ++ show c1)
sprsh3 _ = Nothing
```

At the same time we see that we couldn't have used `Applicative` type class, since we wouldn't be able to access the wrapped return value of `fetch "C1"`.

The authors make the next observation, that not only in theory it is possible to construct dependency graph in case of static dependencies, but also in practise – as presented by this surprisingly simple Haskell function `dependencies`:

```
dependencies :: Task Applicative k v → [k]
dependencies task = getConst $ run task (\k → Const [k]) where
  run :: c f => Task c k v → (k → f v) → f v
  run (Task task) fetch = task fetch
```

The computations are performed using the `Const` functor which is an applicative when working on monoids – in this case lists. As we can see, we aren't using store, which is consistent with our intuition, that for the case of static dependencies the store is not needed.

Moreover, we could not possibly compute dependencies in the case of monadic tasks, since `Const` is not a monad. The best approximation of `dependencies` is `track`, which tracks calls of the `fetch` function using monad transformer `WriteT`:

```
track :: Monad m => Task Monad k v → (k → m v) → m (v, [(k, v)])
track task fetch = runWriterT $ run task trackingFetch
```

```
  where
    trackingFetch :: k → WriterT [(k, v)] m v
    trackingFetch k = do v <- lift (fetch k); tell [(k, v)]; return v
```

However, here we unfortunately need to use the store. We can test our `track` function using for example an `IO` monad, by providing values from keyboard:

```
> fetchIO k = do putStr (k ++ ": "); read <$> getLine
> track (fromJust $ sprsh2 "B1") fetchIO
C1: 1
B2: 10
(10,[("C1",1),("B2",10)])
> track (fromJust $ sprsh2 "B1") fetchIO
C1: 2
A2: 20
(20,[("C1",2),("A2",20)])
```

3.3 Schedulers and rebuilders

Mokhov et al. present a construction, in which build systems are defined by two mechanisms:

- scheduler – which determines the order in which tasks should be built,
- rebuilder – which decides if given tasks should be rebuilt or if instead its value can be read from the store.

While not doing it explicitly, we have already discussed different examples of schedulers and rebuilder. The authors specify three types of schedulers:

- topological – which uses the fact that it works only with static dependencies,
- restarting – which in case of dependency on not built task stops execution of the current one and will restart it later,
- suspending – which instead of stopping only suspends execution of current task until it computes required value.

Mokhov et al. give schedulers and rebuilders following types:

```
type Scheduler c i ir k v = Rebuilder c ir k v → Build c i k v
type Rebuilder c ir k v = k → v → Task c k v → Task (MonadState ir) k v
```

As such, build systems are created by joining some scheduler with some rebuilder. Rebuilders modify the task, which based on rebuilders findings will either

build the original task and store information for rebuilder for next builds, or will return value from store it it's up to date.

In the case of rebuilders, variety is a bit larger, we specify rebuilders based on:

- dirty bit – either in a form of literal bit (as in Excel) or in some other way such as by comparing file modification dates (as in Make) – the mechanism is based on marking tasks if their inputs has changed.
- verifying traces – which during the build process accumulate hashes of results and remember that for instance: task A, when it had hash 1, depended on task B which at the time had hash 2. When hashes are equal, we assume that rebuild is not needed.
- constructive traces – same as verifying ones, but whole values are remembered instead of only their hashes,
- deep constructive traces – instead of tracking values of immediate dependencies, tracked are the values of input tasks that the task depends on. The flaw of this mechanism is no support for nondeterministic task, which are considered by Mokhov et al., and for performing early cut-off since we aren't looking at only immediate dependencies.

Presented by Mokhov et al. categorisation of build systems leads us to division of the systems space into 12 cells, of which 8 are inhabited by existing system:

	Scheduler		
Rebuilder	Topological	Restarting	Suspending
Dirty bit	Make	Excel	-
Verifying traces	Ninja	-	Shake
Constructive traces	CloudBuild	Bazel	-
Deep constructive traces	Buck	-	Nix

3.4 Implementing build systems

Having established categorisation of build systems and definitions of abstract constructions and types in Haskell, we are ready to implement schedulers and rebuilders. Then, creating implementations of known build system (and even those which until now were only empty cells in our table) is just a matter of applying rebuilders to schedulers. All implementations by authors of „Build systems à la carte" are available in their publications [1, 2] and in a repository[1] on GitHub. In chapter five we will see, how implementation of these build systems looks in a language with algebraic effects and handlers.

[1] `https://github.com/snowleopard/build`

Chapter 4

Algebraic effects and handlers in practise

4.1 Programming languages with algebraic effects

Interest in the subject of algebraic effects and handlers lead in the recent years to creation of multiple libraries for languages popular in the academic and functional language enthusiast communities – for Haskell (extensible-effects[1], fused-effects[2], polysemy[3]), Scala (Effekt[4], atnos-org/eff[5]) and Idris (Effects [6]).

Related to the OCaml language, is an initiative ocaml-multicore[7], whose members aim to create an implementation of OCaml that supports concurrency and shared memory by expressing these concepts using effects and handlers.

Research regarding effects and handlers also lead to introduction of a few experimental programming languages in which effects and handlers are first-class citizens. To name a few examples:

- Eff[8] – created by Andrej Bauer and Matija Pretnar language with ML-like syntax,
- Frank[9] [10] – initiated by Sam Lindley, Conor McBride and Craig McLaughlin, motivated by longing for ML and penchant for Haskell's discipline,

[1] https://hackage.haskell.org/package/extensible-effects
[2] https://hackage.haskell.org/package/fused-effects
[3] http://hackage.haskell.org/package/polysemy
[4] https://github.com/b-studios/scala-effekt
[5] https://github.com/atnos-org/eff
[6] https://www.idris-lang.org/docs/current/effects_doc/
[7] https://github.com/ocaml-multicore/ocaml-multicore/wiki
[8] https://www.eff-lang.org/
[9] https://github.com/frank-lang/frank

- Koka[10] – lead by Daan Leijen from Microsoft research project; Koka has syntax inspired by JavaScript,
- Helium[11] [11] – created at University of Wrocław's Institue of Computer Science, with ML-like module system and small traces of Haskell.

4.2 Helium

Using the Helium language we will see in practise how programming with algebraic effects and handlers looks like, and in the next chapter we will try to implement results from „Build systems à la carte" [1, 2]. The first time Helium appears in [11], by serving as a tool for experimenting and constructing more complicated examples and projects for testing effects and handlers in practise.

Let us consider an example of a simple program written in Helium, in which we define a helper function `is_negative` telling if a given number is negative and a function `question`, which asks the user for a number and informs if that number is negative:

```
let is_negative n = n < 0

let question () =
    printStr "What is your favourite number? ";
    let num = readInt () in
    if is_negative num
       then printStr "This number is negative\n"
       else printStr "This number is nonnegative\n";
    printStr "Question finished\n"
```

It is easy to guess that signature of the function `is_negative` as determined by the Helium's type system is `Int -> Bool`. However, when we ask the runtime environment about the type of function `question` we will receive an interesting signature `Unit ->[IO] Unit`. In Helium information about effects occurring during computation of a function are included in function signatures in square brackets. In the case of `question`, its execution causes occurrence of a side effect related to input/output.

```
printStr: String ->[IO] Unit
readInt: Unit ->[IO] Int
```

Type inference system knowing that i/o-operations are declared with above signatures and they are not wrapped by any handler concludes that `IO` effect will leak from the `question` function.

[10] `https://github.com/koka-lang/koka`
[11] `https://bitbucket.org/pl-uwr/helium/src/master/`

Effects `IO` and `RE` (runtime error) are special, since for them there global handlers declared in the standard library. If the effect instance is not handled and surfaces to the runtime environment, the global handler will take care of it. For effect `IO` environment uses standard input/output, while for `RE` computation will be halted with appropriate error message.

4.3 Examples of handler implementations

4.3.1 Error

Let us implement a couple of side effects, starting from error effect, including handlers for them. In Helium effect and operations that cause it are defined like this:

```
signature Error =
| error : Unit => Unit
```

We will create function similar to `question`, however it won't „like" negative numbers:

```
let no_negatives_question () =
  printStr "What is your favourite number? ";
  let num = readInt () in
    if is_negative num
       then error ()
       else printStr "This number is nonnegative\n";
    printStr "Question finished\n"

let main () =
  handle no_negatives_question () with
  | error () => printStr "Error occured!\n"
  end
```

We defined an `Error` side effect with effectful operation `error`. This operation is parameterised by type `Unit` and its (possible) value is also of type `Unit`. Moreover, we define function `main`, which calls `no_negatives_question`. However, the computation is performed inside a handler, which defines what should happen if error effect will occur causes by `error` operation. In this case, we will write a message to standard output. We will not be resuming the computation, thus the error will stop the handled computation. If we execute our program and input a negative number, the program will halt with message as defined in handler, and „Question finished" will not be displayed. As expected `no_negatives_question` was not resumed after error occurred.

If we want to use the same handler multiple times, we can assign it to an identifier – in Helium handlers are values:

```
let abortOnError =
```

```
  handler
  | error () => printStr "Error occured!\n"
  end
```

let us modify function `main` to use the defined handler:

```
let main () =
  handle no_negatives_question () with abortOnError
```

For example's sake, let us consider more „peaceful" handler for `error`, that will display a warning but will not stop the computation:

```
let warnOnError =
  handler
  | error () => printStr "Error occured, continuing...\n"; resume ()
  end
```

If we use this handler in our program, after displaying a warning the execution of `no_negatives_question` will resume and message „Question finished" will be outputted. Special function `resume`, available in handler corresponds to continuation of computation which has been paused when effectful operation was encountered.

4.3.2 Nondeterminism

Let us go back to the problem from chapter 2 which was an inspiration for considering nondeterminism – checking if logical formula is satisfiable and if is a tautology. We introduced two handlers for both instances of our problems. Implementation of nondeterminism's effect, operation `amb` and handlers along with their usages is as follows:

```
signature NonDet =
| amb : Unit => Bool

let satHandler =
  handler
  | amb () / r => r True || r False
  end
```

```
let tautHandler =
  handler
  | amb () / r => r True && r False
  end

let formula1 x y z = (not x) && (y || z)

let main () =
    let ret = handle
                let (x, y, z) = (amb (), amb (), amb ()) in
                formula1 x y z
              with satHandler in
    if ret then printStr "Formula is satisfiable\n"
           else printStr "Formula is not satisfiable\n"
```

We will be verifying if formula represented by a function `formula1` is satisfied. To do so, in `main` – inside the handler – we nondeterministically set values of variables `x`, `y`, `z`, and compute `formula1`. Value of handled expression, which we assign to `ret` is used to display the message. Moreover, to demonstrate capabilities of the language, instead of using `resume` we name the continuation function as `r`.

In Helium handlers can have cases not only for effectful operations but also for two special cases: `return` and `finally`. The first one is executed when computation which is wrapped by the handler finishes with some value and its argument is exactly this value. For the case of `finally`, it receives as argument whole computation inside the handler and is run at the beginning of handler's execution. By default both handlers plainly return received values.

We can cleverly use them. For instance, instead of just checking if formula is satisfiable, we can count the number of valuations that make it true:

```
let countSatsHandler =
  handler
  | return x => if x then 1 else 0
  | amb () / r => r True + r False
  end

let main () =
  let ret = handle
              let (x, y, z) = (amb (), amb (), amb ()) in
              formula1 x y z
            with countSatsHandler in
    printStr (stringOfInt ret ++ " satisfying interpretations\n")
```

When computation is finished, instead of returning boolean value telling if formula was satisfied, we return 1 or 0. While handling nondeterministic choice we resume computation with both boolean values and add the results. By using `finally`

we can include the message about number of valuations into the handler:

```
let countAndWriteSatsHandler =
  handler
  | return x => if x then 1 else 0
  | amb () / r => r True + r False
  | finally ret => printStr (stringOfInt ret ++ " satisfying
    interpretations\n")
  end

let main () =
  handle
    let (x, y, z) = (amb (), amb (), amb ()) in
    formula1 x y z
  with countAndWriteSatsHandler
```

Here we abuse `finally` for example's sake, however we will soon see that this construct is really useful.

4.3.3 Mutable state

Let us consider handler's case for `return`:

```
handler
(* ... *)
| return x => fn s => x
end
```

Final value of the computation, instead of being a simple value, is a function. As the result, in this handler continuations are not simple values but functions. This way we can parameterise continuations not only with values returned by operations (according to their signatures), but also with our – handler author's – own. Let us notice, that since the result of handled computation is now a function and not a simple value – in order for the handler's user not to notice type mismatch – we need to execute this function with some parameter. This is a great moment to use `finally` construct.

We define state effect with operations for reading and modifying it:

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

The effect and its operations are parameterised by types of stored state. Now let us define a standard handler for state effect. We will take advantage of the fact that handlers are values in Helium, and as such can be a resulting value of a function. The function will be parameterised by state's initial value:

```
let evalState init =
```

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

We the computation is finished, instead of returning a value we return a function which ignores its argument (that is current state's value) and returns said value. As the result, cases for operations need to return functions. For `put` we don't read state and resume computation with updated value. However, as we know resumed computation still requires one argument, which we provide from operation's parameter. In case of `get` we read current state's value and resume computation by providing said value both as argument for continuation and as next state value. Finally, we have to decide what to do in the case of `finally`. Since we transformed the computation from returning simple value to the one returning a function that expects value of state, we can call it with initial value of state – provided by handler's user.

If we want to return not only result of computation but also final value of state, we just modify handler's case for `return`:

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

Now, after defining the side effect, its operations and handlers, we can perform computations with mutable state:

```
let stateful () =
  let n = 2 * get () in
  let m = 10 + get () in
    put (n + m);
    m - n
```

```
let main () =
  let init = 2 in
  let (state, ret) = handle stateful () with runState init in
  printStr "Started with "; printInt init;
  printStr "Finished with "; printInt state;
  printStr "Returned "; printInt ret

(* Started with 2
   Finished with 16
   Returned 8 *)
```

4.3.4 Effect of recursion

In some ML-like languages (such as OCaml or Helium) in order for function's identifier to be available inside its body, we need to declare the function using keyword `let rec`:

```
let rec fib n = if n = 0 then 0 else
                if n = 1 then 1 else
                   fib (n-1) + fib (n-2)
```

Surprisingly, thanks to possibility to define custom effects and operations, we can create recursive function that don't appear as such:

```
signature Recurse (A: Type) (B: Type) =
| recurse : A => B

let fib n = if n = 0 then 0 else
            if n = 1 then 1 else
               recurse (n-1) + recurse (n-2)

let rec withRecurse f init =
  handle `a in f `a init with
  | recurse n => resume (withRecurse f n)
  end
```

The `handle `a in ...` construct lets us give labels to effect instances created by handlers – it is useful in ambiguous cases when we use multiple instances of the same effect or for better readability.

Using the effect of recursion, we can also define functions that are mutually recursive:

```
let is_even n = if n = 0 then True
                else recurse (n - 1)

let is_odd n = if n = 0 then False
               else recurse (n - 1)
```

```
let rec withMutualRec me init other =
  handle 'a in me 'a init with
  | recurse n => resume (withMutualRec other n me)
  end

let even n = withMutualRec is_even n is_odd

let main () =
  let n = 10 in
  printInt n;
  if even n
  then printStr "is even"
  else printStr "is odd"
```

We keep an information which function is currently executed and when it asks to perform recursive call, we execute the other function, and finally give back the return value.

4.3.5 Multiple effects at once – failure and nondeterminism

As the last example in this chapter, we will see how easy it is to compose effects in Helium. Let us define effects of nondeterminism and failure and very simple handlers for them:

```
signature NonDet = amb : Unit => Bool

signature Fail = fail : {A: Type}, Unit => A

let failHandler =
  handler
  | fail () => False
  end

let ambHandler =
  handler
  | amb () / r => r True || r False
  end
```

We now define a function that checks if given formula with three free variables is satisfiable:

```
let is_sat (f: Bool -> Bool -> Bool -> Bool) =
  handle
    handle
      let (x, y, z) = (amb (), amb (), amb ()) in
      if f x y z then True else fail ()
    with failHandler
  with ambHandler
```

If formula with given valuation is not satisfied, failure effect occurs. Let us bring our attention to the order of handlers – nondeterminism on the outside, failure on the inside. Thus, when failure occurs, its handler will return false and backtracking to last available choice will be performed by handler of nondeterminism. As the result, value of`is_sat f` is equal to false if and only if failure occurred for each valuation. Now let us consider a function that check if formula is a tautology:

```
let is_taut (f: Bool -> Bool -> Bool -> Bool) =
  handle
    handle
      let (x, y, z) = (amb (), amb (), amb ()) in
      if f x y z then True else fail ()
    with ambHandler
  with failHandler
```

Here, handler of failure is on the outside – occurrence of failure during nondeterministic valuation means that there is a valuation that makes the formula not satisfied, an thus the formula is not a tautology. We can now write a nice function that will tell us if `formula1` is satisfiable and if it is a tautology:

```
let main () =
  printStr "Formula is ";
  if is_sat formula1
  then printStr "satisfiable and "
  else printStr "not satisfiable and ";
  if is_taut formula1
  then printStr "a tautology\n"
  else printStr "not a tautology\n"

(* Formula is satisfiable and not a tautology *)
```

Without hassle we wrote a program, that uses multiple effects at the same time, despite neither of them knowing about the other one. Combining effects is very easy, and order in which we handle them let us make simple and readable definitions of program behaviours in the case when some effectful operation is encountered.

Thanks to Helium, we got a better look at algebraic effects and handlers, we saw examples of handlers and their implementations, and solutions to sample problems. We are now ready to start implementing build systems using effects and handlers – which we will do in the next chapter.

Chapter 5

Build systems using algebraic effects and handlers

In this chapter we will repeat implementation of build systems as presented in „Build systems à la carte" [1], however we will with Helium using algebraic effects and handlers. For starters we will come up with counterparts for Haskell abstract types related to build systems, and later we will implement all rebuilders and all but one schedulers. Finally, we will see what characterises the absent scheduler and learn about other implementation attempts inspired by work of Mokhov et al.

5.1 Concept and types

Let us go back to components of Haskell implementation and figure out its counterparts in Helium.

5.1.1 Store

```
data Store i k v
initialise :: i → (k → v) → Store i k v
getInfo :: Store i k v → i
putInfo :: i → Store i k v → Store i k v
getValue :: k → Store i k v → v
putValue :: Eq k => k → v → Store i k v → Store i k v
```

Authors of „Build systems à la carte" [1] represented the store as a type with read and write functions for persistent information and task results. However, each time it was used was inside some mutable state. Thus, we can simplify our implementation by merging the store with mutable state by making `Store` an effect, and turning its functions into effectful operations.

```
signature StoreEff (I: Type) (K: Type) (V: Type) =
```

```
| getInfo : Unit => I
| putInfo : I => Unit
| getValue : K => V
| putValue : K, V => Unit
```

Similarly as in original implementation of `Store`, `StoreEff` is parameterised by types of persistent information as well as keys and values of tasks. Equations for its operations are analogous to the ones of mutable state but with a fixed key for operations related to result values. Additionally, we define a handler called `funStoreHandler`, in which the key–value dictionary is represented as a function – like in examples from „Build systems à la carte".

```
let funStoreHandler {I K V: Type} (module Key: Comparable K) (store:
    FunStoreType I K V) =
    let (FunStore i lookup) = store in
    handler
    | getInfo ()   => fn i lookup => resume i i lookup
    | putInfo  i   => fn _ lookup => resume () i lookup
    | getValue k   => fn i lookup => resume (lookup k) i lookup
    | putValue k v => fn i lookup =>
                      let lookup' x = if Key.equals x k
                                      then v else lookup x in
                          resume () i lookup'
    | return   x   => fn i lookup => (x, FunStore i lookup)
    | finally f    => f i lookup
    end
```

The implementation is similar to mutable state from chapter 4. For sanity, initial persistent information and return value dictionary is wrapped in `FunStoreType I K V` type.

Since Helium, like other ML-like languages, does not have typeclasses known in Haskell, we define couple of signatures corresponding to typeclasses used in original implementation. For the case of `funStoreHandler` module with signature `Comparable K` is used for comparing keys that identify tasks. One can notice, that an alternative approach could be to represent typeclasses as effects and related functions as effectful operations.

```
type Comparable (T: Type) = sig
    type this = T
    val compare: T -> T ->[] Ord
    val equals:  T -> T ->[] Bool
end

type Hashable (T: Type) = sig
    val hash: T ->[] Hash T
end

type Showable (T: Type) = sig
    val toString: T ->[] String
end
```

```
type Entity (T: Type) = sig
    include (Comparable T)
    include (Hashable T)
    include (Showable T)
end

type KeyValue (K V: Type) = sig
    val Key: Entity K
    val Value: Entity V
end
```

5.1.2 Mutable state

We saw an implementation of mutable state effect in chapter 4 and we use it here. Names for handlers of state, depending on the return value, are assigned as in corresponding functions in Haskell – `runState`, `evalState`, `execState`. We also define simple helper functions `gets` and `modify`, which use provided transformation for reading and modifying state, accordingly. Moreover, we define slightly more complicated function `embedState`.

Definitions of `gets`, `modify` and `embedState`

```
let gets f = f (get ())
let modify f = put (f (get ()))
let embedState {E: Effect} {V: Type} (getter: Unit ->[E] V) (setter: V
    ->[E] Unit) =
    handler
    | get () => resume (getter ())
    | put  s => setter s; resume ()
    end
```

The `embedState` function creates a handler for mutable state effect, in which operations – instead of being done by handler itself – are passed to provided `getter` and `setter` functions, that cause some unknown side effect while being executed. We will use such embedding of mutable state in some other effect for implementing schedulers, which will pass persistent information to rebuilders as embedded mutable state.

5.1.3 Task and build effect

In „Build systems à la carte” task was a function that took operation responsible for building requested tasks, and the return result was wrapped in some type `f` that

Usage example of `embedState`

```
handle ‘store in
    (* ... *)
    handle ‘state in
        (* ... *)
    with embedState (getInfo ‘store) (putInfo ‘store)
    (* ... *)
with (* ... *)
```

belonged to typeclass `c`.

```
newtype Task c k v = Task (forall f. c f => (k → f v) → f v)
type Tasks c k v = k → Maybe (Task c k v)
```

We can notice however, that building a task is an obvious case of a side effect of build system execution, thus in our implementation instead of providing a function usually called `fetch`, we will define a `BuildEff` effect which will occur during task building. The effect will have one associated operation called `fetch`.

```
signature BuildEff (K V: Type) = fetch : K => V

data TaskType (K V: Type) (E: Effect) = Task of ({‘a: BuildEff K V} ->
    Unit ->[E, ‘a] V)

type Tasks (K: Type) (V: Type) = (K -> Option (TaskType K V (effect [])))
```

The task will be a function taking an instance of `BuildEff` and will be polymorphic in types of keys, values and possible side effects different from build effect (this will be useful while implementing rebuilders). Let us notice, that compared to original implementation, definition of task does not include information analogous to typeclass `c`, whose member `f` wrapped the result value – we will go back to the subject of this difference later in the chapter.

5.1.4 Build, scheduler, rebuilder

What is left is to define three types mentioned in the above heading.

```
type Build c i k v = Tasks c k v → k → Store i k v → Store i k v
type Scheduler c i ir k v = Rebuilder c ir k v → Build c i k v
type Rebuilder c ir k v = k → v → Task c k v → Task (MonadState ir) k v
```

Building, like in Haskell, takes a set of tasks and the key to build. However, in our implementation instead of also taking a store, our build has a related side effect.

Our planners will also have signatures similar to its Haskell equivalents, but of course enriched with store's side effect, and with module that defines previously mentioned basic operations on keys and values.

```
type Build (I K V: Type) = {‘s: StoreEff I K V} -> Tasks K V -> K ->[‘s]
    Unit
type Rebuilder (IR K V: Type) = {‘s: State IR} -> KeyValue K V -> K -> V
    -> TaskType K V (effect []) -> TaskType K V (effect [‘s])
type Scheduler (I IR K V: Type) = {‘a: StoreEff I K V} -> KeyValue K V ->
    Rebuilder IR K V -> Build I K V
```

The `Rebuilder` type looks almost like the one from original implementation. However, instead of returning a task with modified constraint, the task has new possible side effect that can occur while the task is built.

5.2 An example: system busy

since we finished defining how abstraction of build systems from Haskell relates to our implementation in Helium, we can write our version of `busy` build system.

The `busy` build system

```
let busy {I K V: Type} {‘a: StoreEff I K V} (tasks: Tasks K V) (key: K) =
    let rec busyH =
        handler
        | fetch k => match tasks k with
                     | None => resume (getValue k)
                     | Some task => let handle with busyH in
                                    let v = run task in
                                        putValue k v;
                                        resume v
                     end
        end
    in
    handle fetch key with busyH
```

The core of the implementation, as in Haskell, is definition of handler (there: function) for `fetch`. Its body is almost a plain translation of original implementation, with a slight difference that instead of resuming the computation implicitly – by returning a value – it is continued explicitly be calling `resume` inside handler's body.

5.3 Implementation of traces

As in „Build systems à la carte", implementation of functions that operate on traces are not particularly interesting – in our cases they almost fully correspond to original ones with exception of using `Writer` effect instead of Haskell's infrastructure built around `Maybe` type and list comprehensions. Implementation with some remarks is available in appendix A and in the source code.

5.4 Running and tracking tasks

While implementing schedulers and rebuilder we will want to execute tasks and track its dependencies. To do so, just like authors of „Build systems à la carte" did, we will define a simple function `run` and slightly more interesting `track`.

```
let run {K V: Type} {E: Effect} {‘b: BuildEff K V} (task: TaskType K V E) =
    let (Task t) = task in t ‘b ()

let track {I K V: Type} {‘b: BuildEff K V} (task: TaskType K V (effect
    [])) =
    let handle with Writer.runListHandler in
    let hTrack = handler
                 | fetch k => let v = fetch ‘b k in
                                  Writer.tell (k, v);
                                  resume v
                 end in
        handle ‘tb in run ‘tb task with hTrack
```

The `track` function takes a label `‘b`, that corresponds to the handler of an instance of build effect, and a task which should be supervised by said handler. The `track` function is responsible for tracking dependencies of a given task. To do so, an additional handler `hTrack` is created which will handle the given task. In the case of task requiring a dependency, `hTrack` will intercept the call to `fetch` and delegate it to the handler label with `‘b`, to finally note that a call to `fetch` took place.

Implementation of `track` is an interesting example of constructing an effect proxy between proper computation and some specialised handler.

5.5 Implementing build systems

5.5.1 Excel

```
open Traces
```

Right at the beginning we see that the promise given by authors of „Build systems à la carte" was fulfilled – build systems are created by applying rebuilders to schedulers.

The `dirtyBitRebuilder` modifies the task, so when executed it will check if key is marked as dirty. If that is the case, the task will be built. Otherwise, the provided value can be returned, since that would be result of building of the initial task.

```
let dirtyBitRebuilder {K V: Type} {‘s: State (K -> Bool)} (module KV: KeyValue K V) (key: K)
    (value: V) (task: TaskType K V (effect [])) = Task (fn (‘b: BuildEff K V) () =>
    let isDirty = get ‘s () in
    if isDirty key then run task
                   else value)
```

In the restarting handler we maintain a chain, which should approximate the order of building tasks which minimises the number of restarts. The execution begins with reuse of chain from last build and its modified version will be stored in an instance of mutable state label `‘chain`. Moreover, in `‘done` state we accumulate tasks which has already been built in this build run, so we do not have to build them again. Then we define a handler for mutable state corresponding to persistent information using the `embedState` function introduced earlier in the chapter.

```
let restarting {IR K V: Type} (module KV: KeyValue K V) {‘ste: StoreEff (Pair IR (List K)) K
    V} (rebuilder: Rebuilder IR K V) (tasks: Tasks K V) (key: K) =
    open KV in
    (* Setup and handling of calculation chain *)
    let chainInsert dep chain =
        let uniqPrepend x xs = x :: filter (not <.> Key.equals x) xs in
            let (curr, rest) = uncons chain in
                uniqPrepend dep rest @ [curr] in
    let newChain =
        let chain = snd (getInfo ()) in
            chain @ (if member Key key chain then [] else [key]) in
    let handle ‘chain with evalState newChain in
    (* Tasks that are up to date in this build session *)
    let type ST = Set Key in
    let handle ‘done with evalState ST.empty in
    (* Embedded state for tasks modified by rebuilder *)
    let handle with embedState (fst <.> getInfo ‘ste) (modifyInfo ‘ste <.> setFst) in
    let rec restartingHandler = (* ... *)
        and loop () = (* ... *)
    in
    let resultChain = loop () in
        modifyInfo (mapSnd (fn _ => resultChain))
```

The core part of this scheduler's implementation consists of build effect handler `restartingHandler` and function `loop`. This function executes tasks in the order determined by the chain from previous build, modifies the tasks using rebuilder and

finally executes them. Concurrently, new chain is constructed and will be returned by `loop`.

```
let rec restartingHandler =
        handler
        | fetch k => if gets ‘done (ST.mem k) then
                        resume (getValue k)
                     else (let (curr, rest) = gets ‘chain uncons in
                               modify ‘chain (chainInsert k);
                               loop ())
        | return x => let (curr, rest) = gets ‘chain uncons in
                          modify ‘done (ST.add curr);
                          put ‘chain rest;
                          putValue curr x;
                          curr :: loop ()
        end
    and loop () =
        match get ‘chain () with
        | [] => []
        | (key::keys) =>
          match tasks key with
          (* Input task *)
          | None => modify ‘done (ST.add key);
                    put ‘chain keys;
                    key :: loop ()
          (* Not built yet, rebuilder takes over *)
          | Some task => let value = getValue key in
                         let newTask = rebuilder KV key value task in
                             handle run newTask with restartingHandler
          end
        end
```

During task's build process calls to `fetch` are intercepted by the handler and checked if requested task has already been built. If that is not the case, the chain is modified so that requested task is in front of the one being currently built. In the handler we use `return` case for noting that the task was finished and for calling `loop`.

5.5.2 Shake

```
restarting KV dirtyBitRebuilder
```

In the Shake build system, rebuilder uses verifying traces. With `verifyVT` the rebuilder checks if the task is „fresh". If so, it does not need to be built again. Otherwise, the task is built while being supervised by `track` which accumulates task's dependencies and creates new traces that will be recorded later by `recordVT`.

```
let vtRebuilder {K V: Type} {‘s: State (VT K V)} (module KV: KeyValue K V) (key: K) (value:
     V) (task: TaskType K V (effect [])) = Task (fn (‘b: BuildEff K V) () =>
    open KV.Value in
    let upToDate = handle verifyVT KV key (hash value) with hashedFetch hash in
        if upToDate then value
        else (let (newValue, deps) = track task in
                 recordVT key (hash newValue) (List.map (fn (k, v) => (k, hash v)) deps);
                 newValue))
```

The implementation of suspending scheduler is significantly shorter. We have only to mutable values: the first one (`‘done`) for remembering already built tasks and the other one for embedding persistent information for the needs of the rebuilder – just like in restarting scheduler.

```
let suspending {IR K V: Type} {‘ste: StoreEff IR K V} (module KV: KeyValue K V) (rebuilder:
     Rebuilder IR K V) (tasks: Tasks K V) (key: K) =
    open KV in
    (* Tasks that are up to date in this build session *)
    let type ST = Set Key in
    let handle ‘done with evalState ST.empty in
    (* Embedded state handler for task modified by rebuilder *)
    let handle with embedState (getInfo ‘ste) (putInfo ‘ste) in
    let rec suspendingHandler =
            handler
            | fetch k => build k; resume (getValue k)
            end
        and build key =
            match (tasks key, gets ‘done (ST.mem key)) with
            (* Not built yet, rebuilder takes over *)
            | (Some task, False) =>
              let value = getValue key in
              let handle with suspendingHandler in
              let newTask = rebuilder KV key value task in
              let newValue = run newTask in
                  modify ‘done (ST.add key);
                  putValue key newValue
            | _ => ()
            end
    in
    build key
```

The `suspendingHandler` is incredibly simple – it just calls `build` function and resumes the computation with result read from the store. In `build`, task is checked for being non-trivial (not an input) and if it has not already been built. In such case, new task is constructed with the help from rebuilder and then executed. Otherwise, the task is up to date and there is not need to build it.

5.5.3 CloudShake

```
    suspending KV vtRebuilder
```

```
let ctRebuilder {K V: Type} {‘s: State (CT K V)} (module KV: KeyValue K V) (key: K) (value:
     V) (task: TaskType K V (effect [])) = Task (fn (‘b: BuildEff K V) () =>
    open KV.Value in
    let cachedValues = handle constructCT KV key (get ‘s ()) with hashedFetch hash in
        if Utils.member KV.Value value cachedValues
        then value
```

```
        else match cachedValues with
             | (cachedValue::_) => cachedValue
             | [] => let (newValue, deps) = track task in
                         recordCT 's key newValue (List.map (fn (k, v) => (k, hash v)) deps);
                         newValue
             end)
```

In thee case of constructive traces, the rebuilder checks if provided value is in the set of known values. Otherwise any known value can be returned or – if no values are known – the task is built. As with the rebuilder using verifying traces, dependencies are tracked while task is being built.

5.5.4 Nix

```
    suspending KV ctRebuilder
```

The rebuilder that uses deep constructive traces is similar to its predecessors. However, as the name suggests, it checks on which input tasks the current one depends.

```
let dctRebuilder {K V: Type} {'s: State (DCT K V)} (module KV: KeyValue K V) (key: K)
     (value: V) (task: TaskType K V (effect [])) = Task (fn ('b: BuildEff K V) () =>
    open KV.Value in
    let cachedValues = handle constructDCT KV key (get 's ()) with hashedFetch hash in
        if Utils.member KV.Value value cachedValues
        then value
        else match cachedValues with
             | (cachedValue::_) => cachedValue
             | [] => let (newValue, deps) = track task in
                     let handle 'b with hashedFetch hash in
                         recordDCT 's 'b KV key newValue (List.map fst deps);
                         newValue
             end)
```

5.6 Absent topological scheduler

As we have seen in chapter 3, restarting and suspending schedulers can deal with both dynamic and static dependencies. That is not the case for the topological scheduler, which only supports tasks with static dependencies, which are modelled in „Build systems à la carte" as members of `Applicative` typeclass.

It seems, that we have no away of stopping inspection of values returned from calls to `fetch`. We could try to wrap them in a type not known to the task's author, which seems to be a step towards the original implementation. That way, we are moving away from algebraic effects and handlers which are the topic of with paper, thus we will not study the subject of modelling static dependencies.

5.7 Existing implementations in other languages

The results presented above are the first – to author's knowledge – attempt to implement build systems inspired by „Build systems à la carte" using algebraic effects and handlers. Having said that, in „Build systems à la carte: Theory and practice" Mokhov et al. mention two attempts of implementing build systems in popular programming languages: Rust [12] and Kotlin [13]. However, in both cases limitations of used languages lead to loss of precision and cleanliness compared to Haskell implementation.

While lack of topological scheduler does space us away from original implementation, the rest of schedulers and rebuilder that we implemented – ignoring syntactic differences between languages – are of similar quality and readability as their originals.

Chapter 6

Summary and conclusions

The aim of this work was to introduce the reader and get interested in the subject of algebraic effects and handlers as well as to present new implementation of build systems following the steps of „Build systems à la carte"'s authors. The implementation in an experimental language Helium intended not only to demonstrate how programming with algebraic effect and handlers looks but also to observe how it differs from dealing with computation effects using monads in Haskell.

Thanks to treating `fetch` as an effectful operation of building, and not as a task's argument, we were able to use capabilities of the language with effect and handlers in the central part of build systems implementation.

Gained – due to programming with effects and handlers instead of monads – freedom of using multiple effects at the same time calmed our fears and encouraged us to experiment. Labelling of different instances of the same effect enabled us to maintain multiple values in mutable state without loss of readability or understanding of the source code. It let us create proxies between different effects.

Representation of store, which was the object that build systems work on, as a side effect not only prevented us from needing to hold it in a mutable state, but also better showed its nature of being a persistent and external resource.

What we lost, was precision in describing level of dependencies between tasks. Transparency in results of effectful operations, compared to wrapping the values with instances of applicative functors or monads, prevented easy representation of tasks with static dependencies.

This problem however, was easy to spot from the start of working on own implementation of results from „Build systems à la carte". Apart from that, during implementation process of build systems, the author didn't encounter any substantial obstacles in programming with algebraic effects and handlers. Others were related to limited experience of the author with Helium, experimental nature of this language and momentary issues of runtime environment with clear explanation where type

mismatches came from.

In summary, programming with algebraic effects and handlers is possible, is enjoyable and frees the author from limitations, that until now seemed to be impossible to avoid. As we have seen, one can try to repeat results done in a popular functional programming language, and with delight discover that implementation using effects and handlers is equally fascinating.

Bibliography

[1] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

[2] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte: Theory and practice. *Journal of Functional Programming*, 30, 2020.

[3] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science*, 319:19–35, 2015.

[4] Andrej Bauer. What is algebraic about algebraic effects and handlers?, 2018.

[5] Gordon Plotkin and John Power. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science*, 45:332–345, 2001.

[6] Gordon Plotkin and John Power. Computational effects and operations: An overview. 2002.

[7] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *arXiv preprint arXiv:1312.1399*, 2013.

[8] Jeremy Yallop and contributors. Effects bibliography. `https://github.com/yallop/effects-bibliography/`, 2016.

[9] Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. Selective applicative functors. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[10] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. *CoRR*, abs/1611.09259, 2016.

[11] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[12] Varun Gandhi. Translation of Build Systems à la Carte to Rust. `https://web.archive.org/web/20191020001014/https://github.com/cutculus/bsalc-alt-code/blob/master/BSalC.rs`, 2018.

[13] Paco Estevez and Devesh Shetty. Translation of Build Systems à la Carte to Kotlin. `https://web.archive.org/web/20191021224324/https://github.com/arrow-kt/arrow/blob/paco-tsalc/modules/docs/arrow-examples/src/test/kotlin/arrow/BuildSystemsALaCarte.kt`, 2019.

[14] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[15] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[16] RUBEN P PIETERS, TOM SCHRIJVERS, and EXEQUIEL RIVAS. Generalized monoidal effects and handlers.

[17] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, 1993.

Appendix A

Notes on the attached source code

A.1 Division of implementation into files

- `Common.he` and `Signature.he` – definitions of basic types and signatures,
- `Store.he` – Store effect and handlers for it,
- `Schedulers.he` – implementations of schedulers,
- `Rebuilders.he` – implementations of rebuilders,
- `Traces.he` – functions for interacting with traces,
- `Track.he` – implementations of `run` and `track` functions,
- `Systems.he` – example definitions of described systems,
- `Spreadsheet.he` – implementations of entities related to spreadsheets,
- `scratch.he` – (demo) usage of build systems to build exemplary tasks,
- `Utils.he` – helper functions,
- `PatchedWriter.he` – extension of `Writer` module from standard library,
- `State.he` – implementation of mutable state effect along with handlers,
- `Logger.he` – utility module for tracking programs' behaviour,
- `SchedulersWithLogging.he` – implementation of schedulers as in `Schedulers.he` enriched with runtime diagnostic messages about their behaviour

A.2 Implementations of traces

Description of traces in chapter 5 was omitted, since they didn't seem like an interesting part of implementation. However, for orderliness it's implementations are provided here.

A.2.1 Trace types

For traces we define a type analogous to the one from original implementation in Haskell.

```
data TraceType (K V A: Type) = Trace of K, (List (Pair K (Hash V))), A
type Deps (K V: Type) = List (Pair K (Hash V))
```

It is worth noting, that building of tasks using `fetch` in verifying and constructive traces happens in a specialised handler, which returns hashes instead of plain values.

A.2.2 Verifying traces

```
type VT (K V: Type) = List (TraceType K V (Hash V))

let recordVT {K V: Type} {‘s: State (VT K V)} (key: K) (hash: Hash V) (deps: Deps K V) =
    modify (fn ts => (Trace key deps hash) :: ts)

let verifyVT {K V: Type} {‘s: State (VT K V)} {‘b: BuildEff K (Hash V)} (module KV: KeyValue
     K V) (key: K) (hash: Hash V) =
    let fetchedHashMatches (k, h) = h = fetch k in
    let matchFor (Trace k deps result) =
        if not (KV.Key.equals k key) || result <> hash then False
        else List.forAll fetchedHashMatches deps
    in List.exists matchFor (get ())
```

A.2.3 Constructive traces

```
type CT (K V: Type) = List (TraceType K V V)

let recordCT {K V: Type} {‘s: State (CT K V)} (key: K) (value: V) (deps: Deps K V) =
    modify (fn ts => (Trace key deps value) :: ts)

let constructCT {K V: Type} {‘b: BuildEff K (Hash V)} (module KV: KeyValue K V) (key: K)
     (ts: CT K V) =
    let fetchedHashMatches (k, h) = h = fetch k in
    let matchFor {‘r: Writer V} (Trace k deps result) =
        if not (KV.Key.equals k key) then ()
        else (let same = List.forAll fetchedHashMatches deps in
                  if same then Writer.tell result else ()) in
    let handle ‘r with Writer.listHandler in
        List.iter (matchFor ‘r) ts
```

In functions `constructCT` and `deepDependencies` we are using `Writer` effect. In „Build systems à la carte” type `Maybe`, function `catMaybes` and list comprehensions were used instead. The author, encouraged by lack of infrastructure around `Some` (Helium's counterpart of `Maybe`), decided to test `Writer` module and see what results it gives.

A.2.4 Deep constructive traces

```
type DCT (K V: Type) = List (TraceType K V V)

let deepDependencies {K V: Type} (module KV: KeyValue K V) (ts: DCT K V) (valueHash: Hash V)
     (key: K) =
    open KV in
    let f (Trace k deps v) = if Key.equals k key && Value.hash v = valueHash
                             then Writer.tell (List.map fst deps)
                             else () in
    let depsList = handle ‘w in List.iter (f ‘w) ts with Writer.listHandler in
        match depsList with
        | []        => [key]
        | (deps::_) => deps (* Authors assume there is only one record for a (k, v) pair *)
        end

let recordDCT {K V: Type} {‘s: State (DCT K V)} {‘b: BuildEff K (Hash V)} (module KV:
     KeyValue K V) (key: K) (value: V) (deps: List K) =
    open KV.Value in
    let deepDeps = Utils.concatMap (deepDependencies KV (get ‘s ()) (hash value)) deps in
    let hs = List.map (fetch ‘b) deepDeps in
    let depends = Utils.zip deepDeps hs in
        modify (fn ts => (Trace key depends value) :: ts)

let constructDCT {K V: Type} {‘b: BuildEff K (Hash V)} (module KV: KeyValue K V) (key: K)
     (ts: DCT K V) = constructCT KV key ts
```