Kwalifikacja i implementacja systemów kompilacji z użyciem efektów algebraicznych

(Categorization and implementation of build systems using algebraic effects)

Jakub Mendyk

Praca licencjacka

Promotor: dr Filip Sieczkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

29 sierpnia 2020

Streszczenie

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.

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 categorization 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.

Spis treści

1. Wprowadzenie **9**

1.1. Problemy z efektami ubocznymi 9

1.2. Radzenie sobie z efektami ubocznymi 9

1.3. Systemy kompilacji 10

1.4. O tej pracy 10

2. O efektach algebraicznych teoretycznie **13**

2.1. Notacja 13

2.2. Równania, efekt porażki i modyfikowalny stan 15

2.3. Poszukiwanie sukcesu 16

2.4. Dalsza lektura 16

3. O systemach kompilacji (i ich klasyfikacji) **19**

3.1. Przykłady systemów kompilacji 19

3.1.1. Make 19

3.1.2. Excel 21

3.1.3. Shake 22

3.1.4. Bazel 22

3.1.5. Wnioski 23

3.2. Abstrakcyjnie o systemach kompilacji 23

3.2.1. Nomenklatura 23

3.2.2. Zasób oraz zadania 23

3.2.3. System kompilacji 24

3.2.4. Polimorficzność zadania 25
3.3. Planiści i rekompilatorzy 27
3.4. Implementowanie systemów 29

4. Efekty algebraiczne i uchwyty w praktyce 31
4.1. Języki programowania z efektami algebraicznymi 31
4.2. Helium 32
4.3. Przykłady implementacji uchwytów 33
4.3.1. Błąd 33
4.3.2. Niedeterminizm 34
4.3.3. Modyfikowalny stan 36
4.3.4. Efekt rekursji 38
4.3.5. Wiele efektów naraz – porażka i niedeterminizm 39

5. Systemy kompilacji z użyciem efektów algebraicznych i uchwytów 41
5.1. Pomysł, typy i idea 41
5.1.1. Zasób (Store) 41
5.1.2. Modyfikowalny stan 43
5.1.3. Zadanie i efekt kompilacji 44
5.1.4. Kompilacja, planista, rekompilator 44
5.2. Przykład: system busy 45
5.3. Implementacja śladów 46
5.4. Uruchamianie i śledzenie działań 46
5.5. Implementacje systemów kompilacji 47
5.5.1. Excel 47
5.5.2. Shake 48
5.5.3. CloudShake 49
5.5.4. Nix 50
5.6. Nieobecny planista topologiczny 50
5.7. Istniejące podejścia do implementacji w innych językach 51

6. Podsumowanie i wnioski **53**

Bibliografia **55**

A Omówienie załączonego kodu źródłowego **57**

A.1. Podział implementacji na pliki 57

A.2. Implementacje śladów 58

A.2.1. Typ śladów 58

A.2.2. Ślady weryfikujące 58

A.2.3. Ślady konstruktywne 58

A.2.4. Głębokie ślady konstruktywne 59

Rozdział 1.

Wprowadzenie

1.1. Problemy z efektami ubocznymi

Programy komputerowe, dzięki możliwości interakcji z zewnętrznymi zasobami takimi jak nośniki pamięci, sieci komputerowe czy użytkownicy oprogramowania, mogą robić istotnie więcej, niż tylko zadane wcześniej obliczenia. W ten sposób przebieg programu i jego wynik staje się jednak zależny od tegoż świata zewnętrznego, a sam program nie jest tylko serią czystych obliczeń, ale także towarzyszących im efektów ubocznych.

Efekty uboczne powodują też, że rozumowanie i wnioskowanie o sposobie oraz prawidłowości działania programów staje się znacznie trudniejsze, a w konsekwencji ogranicza ich modularność i prowadzi do częstszych pomyłek ze strony autorów. Chcąc tego uniknąć, dąży się do wydzielania w programie jak największej części, która składa się z czystych obliczeń. Jednak to, czy jakiś moduł oprogramowania wykonuje obliczenia bez efektów ubocznych niekoniecznie jest jasne i często musimy zaufać autorowi, że w istocie tak jest.

1.2. Radzenie sobie z efektami ubocznymi

Jednym z rozwiązań tego problemu jest zawarcie informacji o posiadaniu efektów ubocznych w systemie typów. Możemy skorzystać wtedy z mechanizmów inferencji i weryfikacji typów do automatycznej identyfikacji funkcji, które nie są czyste – dzięki temu programista może łatwo wyczytać z sygnatury funkcji, które z efektów występują w czasie jej działania. Znanym przykładem umieszczenia efektów w typach jest wykorzystanie monad w języku programowania Haskell. Niestety, jednoczesne użytkowanie dwóch niezależnych zasobów reprezentowanych przez różne monady nie jest łatwe i wymaga dodatkowych struktur, takich jak transformery monad, które niosą ze sobą dodatkowe wyzwania – problem modularności został jedynie przesunięty w inny obszar.

Nowym, konkurencyjnym podejściem do ujarzmienia efektów ubocznych przez wykorzystanie systemu typów są efekty algebraiczne z uchwytami. Powierzchownie, zdają się być podobne do konstrukcji obsługi wyjątków w językach programowania lub wywołań systemowych w systemach operacyjnych. Dzięki rozdziałowi między definicjami operacji związanych z efektami ubocznymi a ich semantyką oraz interesującemu zastosowaniu kontynuacji, dają łatwość myślenia i wnioskowania o programach ich używających. Ponadto, w przeciwieństwie do monad, można bezproblemowo korzystać z wielu z nich jednocześnie.

1.3. Systemy kompilacji

Przykładem programów, których głównym zadaniem jest interakcja z zewnętrznymi zasobami są systemy kompilacji, w których użytkownik opisuje proces wytwarzania wyniku jako zbiór wzajemnie zależnych zadań wraz z informacją jak mają być one wykonywane w oparciu o wyniki innych zadań, zaś system jest odpowiedzialny za ich poprawne uporządkowanie i wykonanie. Ponadto, od systemu kompilacji oczekujemy, że będzie śledził zmiany w danych wejściowych i – gdy poproszony o aktualizację wyników – obliczał ponownie jedynie zadania, których wartości ulegną zmianie. Przykładami systemów kompilacji są Make oraz – co może wydawać się zaskakujące – programy biurowe służące do edycji arkuszy kalkulacyjnych (np. popularny Excel).

W publikacjach pod tytułem „Build systems à la carte" [1, 2], autorzy przedstawiają sposób klasyfikacji systemów kompilacji w oparciu o to, jak determinują one kolejność w jakiej zadania zostaną obliczone oraz jak wyznaczają, które z zadań wymagają ponownego obliczenia. Uzyskana klasyfikacja prowadzi autorów do skonstruowania platformy umożliwiającej definiowanie systemów kompilacji o oczekiwanych właściwościach. Platforma ta okazuje się być łatwa w implementacji w języku Haskell, a klasy typów Applicative oraz Monad odpowiadać mocy języka opisywania zależności między zadaniami do obliczenia.

1.4. O tej pracy

Celem tej pracy jest zapoznanie czytelnika, który miał dotychczas kontakt z językiem Haskell oraz podstawami języków funkcyjnych, z nowatorskim rozwiązaniem jakim są efekty algebraiczne oraz zademonstrowanie – idąc śladami Mokhov'a i innych – implementacji systemów kompilacji z wykorzystaniem efektów algebraicznych i uchwytów w języku programowania Helium. W konsekwencji możliwe jest porównanie obu implementacji oraz zaobserwowanie jak wygląda programowanie z efektami algebraicznymi i uchwytami.

W rozdziale drugim wprowadzony zostaje prosty i nieformalny model obliczeń

wykorzystujący efekty algebraiczne i uchwyty. Zostaje przedstawionych kilka przykładów reprezentacji standardowych efektów ubocznych w opisanym modelu.

Celem rozdziału trzeciego jest wprowadzenie do „Build systems à la carte", opisanie obserwacji poczynionych przez autorów i przedstawienie abstrakcji systemów kompilacji oraz ich konsekwencji. Treść źródłowego artykułu jest opisana w sposób dostateczny, aby zrozumieć implementacje systemów z wykorzystaniem efektów i uchwytów przedstawione w rozdziale piątym. Zachęca się przy tym czytelnika do samodzielnego zapoznania się z całą treścią publikacji Mokhov'a i innych. Jest to pozycja interesująca i łatwa w lekturze.

Rozdział czwarty rozpoczyna się zapoznaniem czytelnika z istniejącymi językami oraz bibliotekami umożliwiającymi programowanie z efektami i uchwytami. Następnie omówiony jest język Helium oraz przykładowe problemy wraz z programami je rozwiązującymi z użyciem efektów i uchwytów. Zademonstrowana jest ponadto łatwość wykorzystywania wielu efektów jednocześnie – w bardziej przystępnej formie niż w przypadku monad w Haskellu.

Zwieńczeniem pracy jest rozdział piąty, w którym przedstawiona jest implementacja planistów, rekompilatorów oraz systemów kompilacji w sposób inspirowany wynikami „Build systems à la carte", jednak używając języka z efektami algebraicznymi i uchwytami. Przedstawione są różnice między abstrakcyjnymi typami od których wyprowadza się implementację oraz w jaki sposób efekty i uchwyty wpływają na formę wyniku. Ponadto, pominięta zostaje implementacja jednego z planistów z wytłumaczeniem dlaczego ma to miejsce.

Rozdział 2.

O efektach algebraicznych teoretycznie

Wprowadzimy notację służącą opisowi prostych obliczeń, która pomoże nam – bez zanurzania się głęboko w ich rodowód matematyczny – zrozumieć jak prostym, a jednocześnie fascynującym tworem są efekty algebraiczne i uchwyty. Przedstawiona notacja jest intencjonalnie nieformalna, gdyż ma w dostępny sposób przedstawić abstrakcyjny opis obliczeń z efektami bez prezentowania konkretnego języka programowania.

Następnie przyjrzymy się, jak możemy zapisać popularne przykłady efektów ubocznych używając naszej notacji. Na koniec, czytelnikowi zostaną polecone zasoby do dalszej lektury, które rozszerzają opis z tego rozdziału.

2.1. Notacja

Będziemy rozważać obliczenia nad wartościami następujących trzech typów:

- boolowskim B – z wartościami T i F oraz standardowymi spójnikami logicznymi,
- liczb całkowitych $\mathbb{Z}$ – wraz z ich relacją równości oraz podstawowymi działaniami arytmetycznymi,
- typem jednostkowym U – zamieszkałym przez pojedynczą wartość u,
- oraz pary tychże typów.

Nasz model składać się będzie z wyrażeń:

- **return** v – gdzie v jest wyrażeniem boolowskim lub arytmetycznym,

- **if** $v_1 = v_2$ **then** e_t **else** e_f – wyrażenie warunkowe, gdzie $v_1 = v_2$ jest pytaniem o równość wartości dwóch wyrażeń arytmetycznych,

- abstrakcyjnych operacji oznaczanych $\{op_i\}_{i \in I}$ – powodujących wystąpienie efektów ubocznych – których działanie nie jest nam znane, zaś ich sygnatury to $op_i : A \to (B \to C) \to D$, gdzie A, B, C oraz D to pewne typy w naszym modelu. Wyrażenie $op_i(n, \kappa)$ opisuje operację z argumentem n oraz dalszą częścią obliczenia κ parametryzowaną wynikiem operacji, które *może (nie musi)* zostać wykonane po wykonaniu operacji,

- uchwytów, czyli wyrażeń postaci **handle** e **with** $\{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}_{i \in I}$, gdzie e to inne wyrażenie; uchwyt definiuje działanie (dotychczas abstrakcyjnych) operacji.

Przykładowymi obliczeniami w naszej notacji są więc:

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

Dla czytelności, pisząc w uchwycie zbiór który nie przebiega wszystkich operacji, przyjmujemy że uchwyt nie definiuje działania operacji; równoważnie, zbiór wzbogacamy o element: $op_i\ n\ \kappa\ \Rightarrow\ op_i(n, \kappa)$.

Nie będziemy wprost definiować przebiegu obliczeń, ale definiujemy kilka równoważności, które będą zachodzić:

- $(\lambda x.\ e_1)\ e_2 \equiv e_1\,[x/e_2]$ – aplikacja argumentu do funkcji,

- **if** $v_1 = v_2$ **then** e_t **else** $e_f \equiv \begin{cases} e_t & \text{gdy } v_1 \equiv v_2 \\ e_f & \text{wpp} \end{cases}$

- **handle return** v **with** $H \equiv$ **return** v – uchwyt nie wpływa na wartość obliczenia, które nie zawiera efektów ubocznych,

- **handle** $op_i(a, \lambda x.\ e)$ **with** $H \equiv h_i\,[n/a,\ \kappa/\lambda x.\ \textbf{handle}\ e\ \textbf{with}\ H]$,
  gdzie $H\ =\ \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}$.

Korzystając z równoważności uprośćmy ostatni z powyższych przykładów:

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

2.2. Równania, efekt porażki i modyfikowalny stan

Do tego momentu nie przyjmowaliśmy żadnych założeń na temat operacji powodujących efekty uboczne. Uchwyty mogły w związku z tym działać w sposób całkowicie dowolny. Ograniczymy się w tej dowolności i nałożymy warunki na uchwyty wybranych operacji. Przykładowo, ustalmy że dla operacji op_r, uchwyty muszą być takie, aby następujący warunek był spełniony:

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

Zauważmy, że istnieje tylko jeden naturalny uchwyt spełniający ten warunek, jest nim $H = \{\ op_r\ n\ \kappa\ \Rightarrow\ n\ \}$. Co więcej, jego działanie łudząco przypomina konstrukcję wyjątków w popularnych językach programowania:

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

Podobieństwo to jest w pełni zamierzone. Okazuje się, że nasz język z jedną operacją oraz równaniem ma już moc wystarczającą do opisu konstrukcji, która w większości popularnych języków nie może zaistnieć z woli programisty, a zamiast tego musi być dostarczona przez twórcę języka.

Rozważmy kolejny przykład. Dla poprawienia czytelności, zrezygnujemy z oznaczeń op_i na operacje powodujące efekty, zamiast tego nadamy im znaczące nazwy: *get* oraz *put*. Operacje te mają sygnatury $get : U \to (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$, $put : \mathbb{Z} \to (U \to \mathbb{Z}) \to \mathbb{Z}$. Spróbujemy wyrazić działanie tych dwóch operacji by otrzymać modyfikowalną komórkę pamięci. Ustalamy równania:

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

  kolejne odczyty z komórki bez jej modyfikowania dają takie same wyniki,

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

  umieszczenie w komórce wartości, która już tam się znajduje, nie wpływa na wynik obliczenia,

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

  obliczenie, które odczytuje wartość z komórki daje taki sam wynik, jak gdyby miało wartość komórki podaną wprost jako argument,

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

  komórka zachowuje się, jak gdyby pamiętała jedynie najnowszą włożoną do niej wartość.

Zauważmy, że choć nakładamy warunki na zewnętrzne skutki działania operacji *get* oraz *put*, to w żaden sposób nie ograniczyliśmy swobody autora w implementacji uchwytów dla tych operacji.

2.3. Poszukiwanie sukcesu

Kolejnym rodzajem efektu ubocznego, który rozważymy w tym rozdziale, jest niedeterminizm. Chcielibyśmy wyrażać obliczenia, w których pewne parametry mogą przyjmować wiele wartości, a ich dobór ma zostać dokonany tak, by spełnić pewien określony warunek. Przykładowo, mamy trzy zmienne x, y oraz z i chcemy napisać program sprawdzający, czy formuła $\phi(x, y, z)$ jest spełnialna. W tym celu zdefiniujemy operację $amb : U \to (B \to B) \to B$ związaną z efektem niedeterminizmu. Napiszmy obliczenie rozwiązujące nasz problem:

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

Gdy definiowaliśmy efekt wyjątku, obliczenie nie było kontynuowane. W przypadku niedeterminizmu kontynuujemy obliczenie dwukrotnie – podstawiając za niedeterministycznie określoną zmienną wartości raz prawdy, raz fałszu – w czytelny sposób sprawdzamy wszystkie możliwe wartościowania, a w konsekwencji określamy czy formuła jest spełnialna.

Możemy zauważyć, że gdybyśmy chcieli zamiast sprawdzania spełnialności, weryfikować czy formuła jest tautologią, wystarczy zmienić tylko jedno słowo – zastąpić spójnik **or** spójnikiem **and** otrzymując nowy uchwyt:

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

Przedstawiona konstrukcja efektów, operacji i uchwytów tworzy dualny mechanizm, w którym operacje są producentami efektów, a uchwyty ich konsumentami. Zabierając źródłom efektów ubocznych ich konkretne znaczenia semantyczne lub nakładając na nie jedynie proste warunki wyrażone równaniami, otrzymaliśmy niezwykle silne narzędzie umożliwiające proste, deklaratywne oraz – co najważniejsze, w kontraście do popularnych języków programowania – samodzielne konstruowanie zaawansowanych efektów ubocznych.

2.4. Dalsza lektura

Rozdział ten miał na celu w przystępny sposób wprowadzić idee, definicje i konstrukcje związane z efektami algebraicznymi i uchwytami, które będą fundamentem

do zrozumienia ich wykorzystania w praktycznych przykładach oraz implementacji systemów kompilacji w dalszych rozdziałach. Czytelnicy zainteresowani głębszym poznaniem historii oraz rodowodu efektów algebraicznych i uchwytów mogą zapoznać się z następującymi materiałami:

- „An Introduction to Algebraic Effects and Handlers" autorstwa Matija Pretnara [3],
- notatki oraz seria wykładów Andreja Bauera pt. „What is algebraic about algebraic effects and handlers?" [4] dostępne w formie tekstowej oraz nagrań wideo w serwisie YouTube,
- prace Plotkina i Powera [5, 6] oraz Plotkina i Pretnara [7] – jeśli czytelnik chce poznać jedne z pierwszych wyników prowadzących do efektów algebraicznych oraz wykorzystania uchwytów,
- społeczność skupiona wokół tematu efektów algebraicznych agreguje zasoby z nimi związane w repozytorium [8] w serwisie GitHub.

Rozdział 3.

O systemach kompilacji (i ich klasyfikacji)

Systemy kompilacji, choć są wykorzystywane w praktycznie wszystkich projektach programistycznych, są przez ich użytkowników na ogół zaniedbywane, traktowane jak zło konieczne, a czasem nawet wywołują lęk oraz złość. Mimo tak dużej popularności i większego – niż mogłoby się wydawać – stopnia skomplikowania, nie cieszyły się specjalnym zainteresowaniem ze strony badaczy. Przyglądnęli się im jednak bliżej Andrey Mokhov, Neil Mitchell oraz Simon Peyton Jones w artykułach „Build systems à la carte" [1] oraz „Build systems à la carte: Theory and practice" [2]. W tym rozdziale prześledzimy ich kroki i omówimy wyniki które otrzymali autorzy, aby w dalszej części pracy samodzielnie zaimplementować przedstawione systemy kompilacji w języku z efektami algebraicznymi oraz uchwytami.

3.1. Przykłady systemów kompilacji

Chcąc zrozumieć głębsze i nietrywialne relacje oraz podobieństwa między systemami kompilacji, przyglądnijmy się najpierw kilku przykładom takich systemów używanych w przemyśle.

3.1.1. Make

Make jest bardzo popularnym, szeroko dostępnym oraz względnie starym systemem kompilacji. Konfiguruje się go przez tworzenie plików zwanych *makefile'ami*, które definiują zadania, zależności między nimi oraz sposób ich zbudowania. Rozważmy przykład konfiguracji dla systemu Make do kompilacji prostego programu w języku C.

Przykładowa konfiguracja systemu 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
```

Przedstawiona konfiguracja definiuje sposób budowania trzech zadań: *util.o*, *main.o* oraz *main.exe*. W linii zawierającej definicję zadania zawarta jest informacja o innych zadaniach, od których definiowane zależy – np. dowiadujemy się że *util.o* zależy od zadań (tutaj: plików) *util.h* oraz *util.c*, a zadanie jest realizowane przez wykonanie polecenia *gcc -c util.c*. Jeśli zadanie nie ma zdefiniowanego sposobu zbudowania, na przykład *util.h* mówimy, że jest wejściem lub zadaniem wejściowym w tej konfiguracji.

Wszystkie informacje o zależnościach między zadaniami są wyrażone w tym jednym pliku *makefile*. Użytkownik, chcąc zbudować zadanie *main.exe*, uruchamia program używając polecenia *make main.exe*. Po uruchomieniu system określi, które zadania mają zostać zbudowane, by zrealizować otrzymane żądanie. Z racji tego, że procedura budowania zadań przebiega tak samo, niezależnie od wyników podzadań, będziemy o takim systemie mówić, że ma statyczne zależności. Dla takich systemów naturalnym porządkiem, w którym zadania powinny być budowane jest porządek topologiczny. W ten sposób każde zadanie będzie wykonane „na świeżych" zależnościach. W przeciwnym razie mogłaby istnieć potrzeba zbudowania zadania jeszcze raz.

Zauważmy, że przy ponownym uruchomieniu budowania może nie być potrzeby wykonywania niektórych zadań gdyż wejścia, od których zależą nie uległy, zmianie. Ta obserwacja prowadzi nas do konceptu minimalności, którą autorzy definiują następująco:

Definicja. (Minimalność) Mówimy, że system kompilacji jest minimalny, gdy w trakcie budowania każde zadanie jest wykonane co najwyżej raz i tylko gdy w przechodnim domknięciu zadań, od których zależy, istnieje takie zadanie wejściowe, które zmieniło swoją wartość od czasu ostatniego budowania.

Dla Make'a informacją, które zadania należy zbudować ponownie są czasy modyfikacji plików, od których zależy zadanie – jeśli plik wynikowy zadania jest starszy niż wejścia, to znaczy, że zadanie powinno być ponownie zbudowane.

Należy też zauważyć, że dla pewnych konfiguracji może nie istnieć porządek topologiczny z nimi związany, gdyż istnieje cykl w zależnościach między zadaniami – nie będziemy jednak rozważać takich przypadków.

3.1.2. Excel

Może się to wydawać zaskakujące, ale o arkuszach kalkulacyjnych (np. programie Excel) możemy myśleć jak o systemach kompilacji. Komórki, których wartości są podane wprost uznajemy za zadania wejściowe, zaś formuły dla pozostałych komórek są definicjami sposobu budowania wartości dla nich. Przy takiej interpretacji, arkusze kalkulacyjne stają się bardzo przyjemnym oraz przydatnym przykładem systemu kompilacji.

Rozważmy teraz przykład arkusza kalkulacyjnego przedstawiony przez autorów oryginalnego artykułu, by łatwiej myśleć o tym rodzaju systemu:

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

Funkcja INDIRECT dynamicznie określa, z której komórki zostanie pobrana wartość, a operator & jest składaniem napisów. Gdy C1 = 1, wartością komórki B1 będzie wartość A1, zaś gdy C1 = 2, wartość zostanie pobrana z A2. Jak widzimy, komórki których wartości są wykorzystywane do obliczenia B1 zależą od wartości C1. W tej sytuacji mówimy o dynamicznych zależnościach między komórkami (a ogólniej, w kontekście systemów kompilacji – zadaniami). Tutaj mamy tylko jeden stopień pośredniości, bo zależności B1 są determinowane przez wejście C1. Ogólniej, stopień pośredniości może być dowolnie duży. W takiej sytuacji mechanizm z sortowaniem topologicznym wykorzystywany w Make’u nie będzie właściwy, gdyż nie możemy a priori – bez spoglądnięcia na stany innych komórek – ustalić właściwego porządku budowania zadań.

Porządkowanie komórek w procesie ich obliczania jest w Excelu trochę bardziej skomplikowane. Mechanizm utrzymuje komórki w ciągu (zwanym łańcuchem). W procesie budowania Excel oblicza wartości komórek zgodnie ze skonstruowanym ciągiem. W sytuacji gdy komórka A potrzebuje wyniku innej, jeszcze nie obliczonej komórki N, Excel dokonuje restartu – przerywa obliczanie A i przesuwa N przed A w ciągu oraz wznawia obliczanie wartości zaczynając od N. Po zakończeniu budowania, otrzymany ciąg komórek ma taką własność, że ponowne budowanie przy niezmienionych wejściach odbędzie się bez restartów. Ciąg pełni funkcję aproksymacji właściwego porządku obliczania komórek. Chcąc określić, które komórki muszą być obliczone ponownie, Excel dla każdej komórki utrzymuje informację czy jest ona brudna. Komórki stają się brudne, gdy:

- są wejściem i ich wartość zostanie zmieniona,
- ich formuła zostanie zmieniona,
- zawierają w formule funkcje, które uniemożliwiają statyczne określenie zależności – jak na przykład INDIRECT czy IF.

Łatwo zauważyć, że Excel nie jest zatem minimalnym systemem budowania, gdyż z nadmiarem przyjmuje, które komórki muszą być obliczone ponownie. Ponadto, Excel śledzi nie tylko zmiany w wartościach wejść, ale także definicjach budowania zadań (formułach), co jest rzadką własnością w systemach kompilacji. Na ogół zmiana specyfikacji zadań wymusza na użytkowniku manualne rozpoczęcie pełnego procesu budowania.

3.1.3. Shake

Shake jest systemem kompilacji, w którym zadania definiuje się pisząc programy w języku specjalnego przeznaczenia osadzonym w Haskellu. Można w nim tworzyć konfiguracje z dynamicznymi zależnościami. Jednak w przeciwieństwie do Excela, Shake ma własność minimalności.

Zamiast konstruować ciąg zadań, jak robi to Excel, Shake generuje w trakcie budowania graf zależności. Ponadto, w przypadku wystąpienia zadania zależnego od innego dotychczas nieobliczonego, wstrzymuje wykonanie aktualnego i rozpoczyna budowanie wymaganego zadania. Gdy to się uda, wraca do wstrzymanego zadania znając już potrzebny wynik, by wznowić budowanie.

Inną własnością, którą posiada Shake, jest możliwość wykonywania wczesnego odcięcia – w sytuacji, gdy jakieś zadanie zostało obliczone ponownie, ale jego wynik się nie zmienia, nie ma potrzeby ponownego obliczania zadań, które od niego zależą. Make i Excel nie posiadają takiej optymalizacji.

3.1.4. Bazel

Ostatnim przykładem systemu kompilacji jest Bazel, który powstał w odpowiedzi na zapotrzebowanie ze strony dużych zespołów pracujących nad oprogramowaniem znacznej wielkości. W takich projektach wiele osób może niezależnie budować te same fragmenty oprogramowania, co prowadzi do marnowania zasobów obliczeniowych oraz czasu programistów.

Bazel jest chmurowym systemem budowania – gdy użytkownik chce zbudować oprogramowanie, system komunikuje się z serwerem i sprawdza, które z zadań mają niezmienione wejścia oraz czy zostały już przez kogoś zbudowane. Bazel skopiuje wyniki takich zadań do komputera użytkownika oszczędzając mu czas. Jako że pojedynczy programista na ogół wykonuje zmiany zamknięte w zaledwie kilku modułach, wyniki wielu zadań pozostają niezmienne i jedynie niewielka część z zadań będzie musiała być ponownie zbudowana.

System śledzi zmiany sprawdzając wartości funkcji skrótu plików źródłowych. Gdy skróty pliku na komputerze użytkownika oraz serwerze systemu nie są zgodne, zadanie jest uznawane ze nieaktualne i budowane od nowa. Następnie wynik oraz

nowe wartości funkcji skrótu są zapisywane na serwerze, funkcjonującym dla użytkowników jako „pamięć podręczna" wyników budowania zadań.

Bazel nie wspiera aktualnie dynamicznych zależności. W procesie budowania wykorzystuje mechanizm restartowania zadań, a w celu określenia, które zadania muszą być przebudowane, utrzymuje wartości i skróty wyników zadań oraz historię wykonanych komend budowania.

3.1.5. Wnioski

Przedstawione cztery systemy kompilacji pokazały nam różne stopnie dowolności dane autorowi zadań co do stopnia skomplikowania ich obliczania. Poznaliśmy mechanizmy służące budowaniu zadań i optymalizacje, które zmniejszają liczbę niepotrzebnie obliczanych zadań. Ich wykorzystanie umożliwia niektórym systemom kompilacji osiągnięcie minimalności.

3.2. Abstrakcyjnie o systemach kompilacji

Po przedstawieniu aktualnego stanu rzeczy, autorzy proponują nomenklaturę i abstrakcyjną reprezentację przestrzeni związanej z systemami kompilacji.

3.2.1. Nomenklatura

Obiektem, na którym operuje system kompilacji jest zasób (Store), który kluczom przypisuje wartości. W przypadku Excela jest to arkusz złożony z komórek, zaś w Make'u system plików. Celem systemu jest zmodyfikowanie stanu zasobu w takich sposób, by wartość związana ze wskazanym przez użytkownika kluczem stała się aktualna. System ma pamięć w formie utrzymywanych trwałych informacji na potrzeby kolejnych uruchomień. Użytkownik dostarcza opis zadań w formie instrukcji określających jak mają być skonstruowane w oparciu o wyniki innych zadań.

System kompilacji otrzymuje definicje zadań, zasób na którym działa oraz klucz, który ma zostać zaktualizowany, wraz z jego zależnościami. Po zakończeniu działania, wartość w Store związana ze wskazanym kluczem ma być aktualna.

3.2.2. Zasób oraz zadania

Autorzy proponują następującą abstrakcyjną reprezentację zadania oraz zadań (jako kompletu definicji tychże):

```
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)
```

Zadanie oblicza swoją wartość korzystając z dostarczonej funkcji służącej uzyskiwaniu wartości innych zadań. Jest ono parametryzowane typem v zwracanej wartości, typem kluczy k. Jak widzimy, wartość nie jest zwracana wprost, a w nieznanym nośniku f, który spełnia jednak warunek c. Przykładami warunków w tym kontekście będą *Applicative* oraz *Monad*.

Grupa zadań jest funkcją, która kluczowi być może przyporządkowuje definicję jak skonstruować zadanie identyfikowane wskazanym kluczem. Zadania wejściowe nie mają do swoich kluczy przyporządkowanych definicji, a ich wartości są pobierane ze Store'a. Przykładowo, następującą instancję arkusza kalkulacyjnego:

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

możemy wyrazić w naszej abstrakcji tak:

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

Zasób jest abstrakcyjnym typem danych parametryzowanym typami kluczy, wartości oraz trwałej informacji wykorzystywanej przez system kompilacji:

```
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
```

Autorzy definiują podstawowe operacje na zasobie do konstruowania go, pozyskiwania i aktualizacji trwałej informacji oraz wartości kluczy.

3.2.3. System kompilacji

Typ systemu kompilacji wynika wprost z jego definicji – otrzymuje zadania, zasób oraz klucz, a po zakończeniu działania, wartość w Store związana ze wskazanym kluczem ma być aktualna:

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

Rozważmy implementację bardzo prostego systemu budowania wyrażonego z użyciem przedstawionej abstrakcji:

```
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* uruchamia obliczenie zadania w kontekście modyfikowalnego stanu, służy on spamiętywaniu wartości obliczonych zadań. Gdy zadanie ma być obliczone, jeśli jest wejściowym, to odczytana zostaje jego wartość ze Store'a, w przeciwnym razie zostaje wykonana jego definicja. System ten, podobnie jak kolejne, które zobaczymy później, składa się głównie z funkcji *fetch*, która determinuje jego sposób działania. System *busy* nie jest oczywiście minimalny, chociaż działa poprawnie i jest punktem początkowym do konstrukcji właściwych systemów.

System taki możemy łatwo uruchomić na przykładowym zasobie. Będzie on słownikiem realizowanym przez funkcję – w ten sposób możemy łatwo ustalić wartość domyślną dla wszystkich wejściowych pól:

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

System działa i daje poprawne wyniki. Widzimy też, że przydaje nam się skwantyfikowanie ogólne parametru f w definicji zadania:

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

W tym przypadku `c = Applicative` oraz `f = State (Store () k v) v`, w ten sposób funkcja `fetch` może jako efekt uboczny wykonywać operacje na modyfikowalnym stanie opakowującym Store.

3.2.4. Polimorficzność zadania

Opakowanie wartości wynikowej umożliwia wykonywanie obliczeń z efektami ubocznymi, zaś kwantyfikator ogólny daje autorowi systemu kompilacji pełną swobodę doboru struktury, która będzie właściwa do jego potrzeb. W przypadku systemu `busy` jest to modyfikowalny stan, w którym przechowywany jest zasób.

Gdyby `f` było w pełni dowolne, nie dałoby się nic pożytecznego z nim zrobić, stąd musi być ograniczone przez pewne `c`. Co zaskakujące, to ograniczenie definiuje jak skomplikowane mogą być zależności między zadaniami. Rozważmy trzy popularne (i jedną dodatkową) klasy typów w Haskellu:

- `Functor` – umożliwia nakładanie funkcji na wartość, którą opakowuje. Myśląc graficznie – pracując z funktorem, tworzymy ciąg obliczeń modyfikujących wartość.
- `Applicative` – umożliwia scalanie wielu wartości przez nakładanie na nie funkcji. Tutaj obliczenia prezentują się jako skierowany graf acykliczny.
- `Monad` – w tym przypadku otrzymujemy dowolny graf, który jest ponadto dynamiczny (ze względu na wartości wynikowe). W procesie obliczeń możemy wyłuskiwać wartości i podejmować w oparciu o nie decyzje.
- `Selective` [9] – jest formą pośrednią między funktorami aplikatywnymi, a monadami. Możliwe jest podejmowanie decyzji w oparciu o wyniki, jednak opcje do wyboru są zdefiniowane statycznie.

Autorzy dokonują więc niezwykle ciekawego odkrycia: zadania, w których typ `f` jest funktorem aplikatywnym, mogą mieć jedynie statyczne zależności, zaś dynamiczne są możliwe gdy `f` jest monadą!

Tak więc, przykład z INDIRECT w Excelu – korzystając z naszej abstrakcji – możemy w Haskellu przedstawić następująco:

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

Jednocześnie widzimy, że nie moglibyśmy wyrazić go z użyciem funktora aplikatywnego, gdyż nie mielibyśmy jak wyłuskać wartości komórki z wywołania `fetch "C1"`.

Autorzy czynią kolejną obserwację, że nie tylko w teorii istnieje możliwość skonstruowania grafu zależności w zadaniach o statycznych zależnościach, ale także w praktyce – realizuje to w Haskellu zaskakująco prosta funkcja `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
```

Obliczenie wykonujemy korzystając z funktora `Const`, który jest funktorem aplikatywnym, gdy pracuje na monoidach – w tym przypadku listach. Jak widzimy,

nigdzie nie jest wspomniany Store, co idzie w zgodzie z intuicją, że w przypadku zależności statycznych nie jest on nam potrzebny.

Jednocześnie nie moglibyśmy w takich sposób poznać zależności zadań z monadą, czyli dynamicznymi zależnościami, gdyż typ `Const` nie jest monadą. Najlepszym przybliżeniem funkcji `dependencies` jest `track`, która śledzi wywołania funkcji pozyskującej wartość zadania z wykorzystaniem transformera monad `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
```

W tym przypadku musimy już niestety pracować z zasobem. Przykładowo, możemy przetestować funkcję `track` korzystając z monady `IO`, a wartości wprowadzając za pomocą klawiatury:

```
> 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. Planiści i rekompilatorzy

Autorzy proponują konstrukcję, w której system kompilacji jest definiowany przez dwa mechanizmy:

- planistę (scheduler) – który decyduje w jakiej kolejności zadania powinny być budowane oraz
- rekompilatora (rebuilder) – który określa czy dane zadanie powinno być ponownie zbudowane, czy raczej wystarczy odczytać jego wartość wynikową ze Store'a.

Nie robiąc tego wprost rozważaliśmy już różne przykłady schedulerów i rebuilderów. Autorzy wyszczególniają trzy rodzaje planistów:

- topologicznego (topological) – który wykorzystuje fakt, że zadania mają statyczne zależności,

- restartującego (restarting) – który gdy w czasie obliczania zadania napotka na inne, niezaktualizowane zadanie, przerywa obliczanie bieżącego i kiedyś zacznie je od nowa,
- wstrzymującego (suspending) – który zamiast zaczynać od nowa, wstrzymuje jedynie obliczanie zadania do czasu uzyskania żądanej wartości.

Autorzy abstrakcyjnie przedstawiają planistów i rekompilatorów jako typy:

```
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
```

Tak więc, system kompilacji powstaje przez scalenie jakiegoś schedulera z jakimś rebuilderem. Rebuilder otrzymując klucz zadania oraz jego aktualną wartość i sposób obliczania tworzy nowe zadanie, które w oparciu o wnioski rekompilatora albo zbuduje zadanie i zagreguje dane dla rebuildera na potrzeby kolejnych uruchomień, albo zwróci wartość ze Store'a jeśli jest ona aktualna.

W przypadku rekompilatorów różnorodność jest trochę większa, wyszczególniamy rebuildery oparte o:

- brudny bit – czy to w formie dosłownego bitu dla każdej komórki, jak to ma miejsce w Excelu, czy nietrywialnie przez weryfikowanie dat modyfikacji jak w Make'u – mechanizm jest oparty na oznaczaniu wszystkich zadań wejściowych, których wartości się zmieniły od ostatniego uruchomienia systemu.
- ślady weryfikujące – które w procesie budowania rejestrują wartości funkcji skrótu uzyskanych wyników zadań i pamiętają, że na przykład zadanie A, gdy miało wartość o skrócie 1 było zależne od zadania B, gdy to miało wartość o skrócie 2. W sytuacji, gdy skróty są zgodne uznaje się, że ponowne obliczenie nie jest potrzebne.
- ślady konstruktywne – podobne do poprzedników, jednak funkcja skrótu jest funkcją identycznościową. Innymi słowy – spamiętujemy całe wartości wynikowe zadań.
- głębokie ślady konstruktywne – zamiast rejestrować wartości bezpośrednich zależności, rejestrowane są wartości zadań wejściowych od których zadanie zależy (niezależnie czy bezpośrednio czy nie). Wadą tego mechanizmu jest brak wsparcia dla niedeterministycznych zadań, które rozważają autorzy w dalszej części swojej publikacji oraz brak możliwości wykonania wczesnego odcięcia, gdyż nie spoglądamy na wartości od których zadanie zależy bezpośrednio.

Sposób skategoryzowania systemów kompilacji przedstawiony przez autorów prowadzi do podziału przestrzeni systemów na 12 komórek, z czego 8 jest zamieszkałych przez istniejące rozwiązania:

	Planista		
Rekompilator	Topologiczny	Restartujący	Wstrzymujący
Brudny bit	Make	Excel	-
Ślady weryfikujące	Ninja	-	Shake
Ślady konstruktywne	CloudBuild	Bazel	-
Głębokie ślady konstruktywne	Buck	-	Nix

3.4. Implementowanie systemów

Mając już ustaloną klasyfikację oraz definicje abstrakcyjnych konstrukcji i typów w Haskellu, można zaimplementować planistów i rekompilatorów. Wtedy utworzenie implementacji znanych systemów kompilacji (a nawet tych, które dotychczas były tylko pustymi polami w tabeli) jest zwykłym zaaplikowaniem rebuildera do schedulera. Wszystkie implementacje przedstawione przez autorów „Build systems à la carte" są dostępne w tekstach artykułów [1, 2] oraz w repozytorium[1] w serwisie GitHub. W rozdziale 5 zobaczymy, jak implementacja takich systemów wygląda w języku z efektami algebraicznymi i uchwytami.

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

Rozdział 4.

Efekty algebraiczne i uchwyty w praktyce

4.1. Języki programowania z efektami algebraicznymi

Zainteresowanie efektami algebraicznymi oraz uchwytami doprowadziło do powstania w ostatnich latach wielu bibliotek dla języków popularnych w środowisku akademickim i pasjonatów języków funkcyjnych – Haskella (extensible-effects[1], fused-effects[2], polysemy[3]), Scali (Effekt[4], atnos-org/eff[5]) i Idris (Effects [6]).

Związana z językiem OCaml jest inicjatywa ocaml-multicore[7], której celem jest stworzenie implementacji OCamla ze wsparciem dla współbieżności oraz współdzielonej pamięci, a cel ten jest realizowany przez wykorzystanie konceptu efektów i uchwytów.

Badania nad efektami i uchwytami przyczyniły się także do powstania kilku eksperymentalnych języków programowania w których efekty i uchwyty są obywatelami pierwszej kategorii. Do języków tych należą:

- Eff[8] – powstający z inicjatywy Andreja Bauera i Matija Pretnara język o ML-podobnej składni,
- Frank[9] [10] – pod przewodnictwem Sama Lindley'a, Conora McBride'a oraz Craiga McLaughlin'a, projektowany z tęsknoty do ML'a, a jednocześnie upodobania do Haskell-owej dyscypliny,

[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] – kierowany przez Daana Leijena z Microsoft projekt badawczy; Koka ma składnię inspirowaną JavaScriptem,

- Helium[11] [11] – powstały w Instytucie Informatyki Uniwersytetu Wrocławskiego, z ML-podobnym systemem modułów i lekkimi naleciałościami z Haskella.

4.2. Helium

Używając właśnie języka Helium zobaczymy, jak w praktyce wygląda programowanie z efektami algebraicznymi oraz uchwytami, zaś w następnym rozdziale spróbujemy zaimplementować wyniki uzyskane w „Build systems à la carte" [1, 2]. Po raz pierwszy Helium pojawia się w [11], służąc za narzędzie do eksperymentowania i umożliwienia konstrukcji bardziej skomplikowanych przykładów oraz projektów w celu przetestowania efektów i uchwytów w praktyce.

Rozważmy przykład prostego programu napisanego w Helium, w którym definiujemy pomocniczą funkcję `is_negative` ustalającą, czy liczba jest ujemna oraz funkcję `question`, która pyta użytkownika o liczbę i informuje, czy liczba ta jest ujemna:

```
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"
```

Sygnatura funkcji `is_negative` wyznaczona przez system typów Helium – to jak łatwo się domyślić – `Int -> Bool`. Gdy jednak zapytamy środowisko uruchomieniowe o typ funkcji `question` otrzymamy interesującą sygnaturę `Unit ->[IO] Unit`. W Helium informacje o efektach występujących w trakcie obliczania funkcji są umieszczone w sygnaturach funkcji w kwadratowych nawiasach. W przypadku funkcji `question`, jej obliczenie powoduje wystąpienie efektu ubocznego związanego z mechanizmem wejścia/wyjścia.

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

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

System inferencji typów wiedząc, że operacje we/wy są zadeklarowane z powyższymi sygnaturami wnioskuje, że skoro wystąpienia tychże operacji w kodzie `question` nie są obsługiwane przez uchwyt, to efekt `IO` wyjdzie poza tą funkcję.

Efekty `IO` oraz `RE` (runtime error) są szczególne, gdyż są dla nich zadeklarowane globalne uchwyty w bibliotece standardowej – jeśli efekt nie zostanie obsłużony i dotrze do poziomu środowiska uruchomieniowego, to ono zajmie się jego obsługą. Dla efektu `IO` środowisko skorzysta ze standardowego wejścia/wyjścia, zaś w przypadku wystąpienia efektu `RE`, obliczenie zostanie przerwane ze stosownym komunikatem błędu.

4.3. Przykłady implementacji uchwytów

4.3.1. Błąd

Zaimplementujemy kilka efektów ubocznych, zaczynając od efektu błędu, wraz z uchwytami dla nich. W Helium efekt oraz powodujące go operacje definiuje się następująco:

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

Stwórzmy funkcję podobną do `question` z tym, że nie będzie ona „lubić” wartości ujemnych:

```
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
```

Zdefiniowaliśmy efekt uboczny `Error` wraz z operacją `error`, która go powoduje. Operacja ta jest parametryzowana wartością typu `Unit`, a jej (możliwy) wynik to także wartość z `Unit`. Definiujemy też funkcję `main`, w której wywołujemy `no_negatives_question`. Jednakże obliczenie wykonujemy w uchwycie, w którym definiujemy co ma się wydarzyć, gdy w czasie obliczenia wystąpi efekt błędu spowodowany operacją `error`. W tym przypadku mówimy, że będzie to skutkowało wypisaniem wiadomości na standardowe wyjście. Nie wznawiamy obliczenia, stąd błąd skutkuje zakończeniem nadzorowanego obliczenia. Jeśli uruchomimy teraz program i

podamy ujemną liczbę, zakończy się on komunikatem zdefiniowanym w uchwycie, a tekst „Question finished" nie zostanie wypisany. Zgodnie z oczekiwaniami – obliczenie `no_negatives_question` nie zostało kontynuowane po wystąpieniu błędu.

Jeśli pewnego uchwytu zamierzamy używać wiele razy, możemy przypisać mu identyfikator – w Helium uchwyty są wartościami:

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

zmodyfikujmy funkcję `main` by korzystać ze zdefiniowanego uchwytu:

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

Na potrzeby przykładu możemy rozważyć „spokojniejszy" uchwyt dla wystąpień `error`, który wypisze ostrzeżenie o błędzie ale będzie kontynuował obliczenie:

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

Jeśli skorzystamy z tego uchwytu w programie, po wyświetleniu ostrzeżenia obliczenie `no_negatives_question` zostanie wznowione i na ekranie zobaczymy komunikat „Question finished". Specjalna funkcja `resume`, dostępna w uchwycie reprezentuje kontynuację obliczenia, które zostało przerwane wystąpieniem operacji powodującej efekt uboczny.

4.3.2. Niedeterminizm

Powróćmy do problemu, który w rozdziale drugim był inspiracją do rozważania niedeterminizmu – sprawdzanie czy formuła jest spełnialna oraz czy jest tautologią. Przedstawiliśmy wtedy uchwyty dla obu tych problemów w naszej notacji. Implementacja efektu niedeterminizmu, operacji `amb` oraz uchwytów wraz z wykorzystaniem ich wygląda następująco:

```
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"
```

Będziemy sprawdzać, czy formuła wyrażona za pomocą funkcji `formula1` jest spełnialna. W tym celu w `main` – wewnątrz uchwytu – niedeterministycznie ustalamy wartości zmiennych `x`, `y`, `z`, po czym obliczamy wartość `formula1`. Wartość obsługiwanego wyrażenia, którą przypisujemy do zmiennej `ret`, jest następnie wykorzystana do wypisania komunikatu. Ponadto – w celu demonstracji możliwości języka – w uchwytach zamiast kontynuować obliczenie używając `resume`, przypisujemy kontynuacji nazwę `r`.

W Helium uchwyty mogą posiadać przypadki nie tylko dla operacji związanych z jakimś efektem ale także dwa specjalne: `return` oraz `finally`. Pierwszy jest wykonywany, gdy obliczenie pod kontrolą uchwytu kończy się zwracając wynik. Przypadek `return` jako argument otrzymuje wynik obliczenia. Zaś `finally` otrzymuje jako argument obliczenie obsługiwane przez uchwyt i jest uruchamiane na początku działania uchwytu. Domyślnie przypadki zwracają otrzymane wartości.

Możemy je jednak sprytnie wykorzystać. Przykładowo, zamiast tylko sprawdzać czy formuła jest spełnialna, możemy sprawdzić przy ilu wartościowaniach jest prawdziwa:

```
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")
```

Gdy obliczenie się kończy – zamiast zwracać, czy formuła jest spełniona – zwracamy 1 albo 0, w zależności, czy formuła przy aktualnym wartościowaniu jest

spełniona. Gdy obsługujemy niedeterministyczny wybór, kontynuujemy obliczenie dla obu możliwych wartości boolowskich po czym dodajemy wyniki. Wykorzystując `finally` możemy włączyć komunikat o liczbie wartościowań do uchwytu:

```
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
```

Tutaj wykorzystanie `finally` jest lekkim nadużyciem, jak jednak za chwilę zobaczymy, konstrukcja ta jest bardzo przydatna.

4.3.3. Modyfikowalny stan

Rozważmy następujący przypadek dla `return` w uchwycie:

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

Wartość obliczenia, zamiast być jego wynikiem, jest funkcją. Co za tym idzie, w tym uchwycie kontynuacje nie będą funkcjami zwracającymi wartości, lecz funkcje. W ten sposób możemy parametryzować dalsze obliczenia nie tylko wartościami zwracanymi przez operacje (zgodnie z ich sygnaturą), ale także wymyślonymi przez nas – autorów uchwytu. Zauważmy jednak, że parametr ten nie jest widoczny w obsługiwanym obliczeniu, a jedynie w uchwycie. Co więcej, skoro wynik obsługiwanego obliczenia jest teraz funkcją, a nie wartością, to – by użytkownik uchwytu nie zauważył niezgodności typów – musimy funkcję tą uruchomić z jakimś parametrem. Tutaj właśnie przychodzi naturalny moment na wykorzystanie konstrukcji `finally`.

Definiujemy efekt stanu z operacją jego odczytu oraz modyfikacji:

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

Efekt, jak i operacje są polimorficzne ze względu na typ wartości stanu. Zdefiniujemy teraz standardowy uchwyt dla efektu stanu. Skorzystamy z faktu, że uchwyty

są w Helium wartościami, stąd w szczególności mogą być wynikiem funkcji. Funkcja ta będzie u nas parametryzowana wartością początkową stanu:

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

Gdy obliczenie się kończy, zamiast wartość zwracamy funkcję, która ignoruje argument (będzie nim bieżąca wartością stanu), a zwraca właściwy wynik obliczenia. W konsekwencji przypadki dla operacji też muszą być funkcjami. Dla `put` nie musimy odczytywać aktualnej wartości stanu, stąd wartość tą ignorujemy. Obliczenie wznawiamy z wartością jednostkową. Jak jednak wiemy, wynikiem nie będzie zwykła wartość, lecz funkcja, której u nas dajemy wartość stanu. Stąd podajemy jej nowy stan, którym parametryzowana była operacja `put`. W przypadku `get` postępujemy podobnie – jednak tym razem odczytamy argument funkcji i przekażemy go do kontynuacji. Niezmiennie kontynuacja zwraca funkcję, której przekażemy aktualną wartość stanu. Pozostaje rozstrzygnąć, co zrobić w przypadku `finally`. Skoro jednak przerobiliśmy obliczenie ze zwracającego wartość do takiego, które zwraca funkcję oczekującą wartości stanu, to możemy podać mu wartość początkową – określoną przez użytkownika uchwytu.

Jeśli chcemy, aby obliczenie zwracało nie tylko wartość wynikową, ale także końcowy stan, wystarczy że zmodyfikujemy przypadek dla `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
```

Dzięki zdefiniowanemu efektowi ubocznemu, operacjom oraz uchwytom możemy teraz łatwo wykonywać obliczenia ze stanem:

```
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. Efekt rekursji

W niektórych językach ML-podobnych (jak na przykład OCaml czy Helium) chcąc, by w ciele definicji funkcji był widoczny jej identyfikator, trzeba zadeklarować ją używając słów kluczowych `let rec`:

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

Co ciekawe, dzięki własnym efektom i operacjom możemy tworzyć funkcje rekurencyjne, które nie używają jawnie rekursji:

```
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
```

Konstrukcja `handle 'a in ...` służy doprecyzowaniu, który efekt ma być obsłużony przez uchwyt – jest przydatna w przypadku niejednoznaczności, gdy używamy wielu instancji tego samego efektu lub dla ułatwienia rozumienia kodu.

Korzystając z efektu rekursji, możemy także definiować funkcje wzajemnie rekurencyjne:

```
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"
```

Utrzymujemy informację, która funkcja jest aktualnie wykonywana i gdy prosi o wywołanie rekurencyjne, uruchamiamy obliczanie drugiej funkcji, po czym wynik przekazujemy do kontynuacji.

4.3.5. Wiele efektów naraz – porażka i niedeterminizm

Na koniec rozdziału zobaczymy jak łatwo w Helium komponuje się efekty. Definiujemy efekty niedeterminizmu i porażki oraz bardzo proste uchwyty dla tych efektów:

```
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
```

Definiujemy teraz funkcję sprawdzającą, czy otrzymana formuła z trzema zmiennymi wolnymi jest spełnialna:

```
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
```

Jeśli formuła przy ustalonym wartościowaniu nie jest spełniona, powoduje efekt porażki. Zwróćmy uwagę w jakiej kolejności są umieszczone uchwyty – niedeterminizmu na zewnątrz, zaś porażki wewnątrz. W ten sposób, gdy wystąpi porażka, jej uchwyt zwróci fałsz, w wyniku czego nastąpi powrót do ostatniego punktu niedeterminizmu, w którym jest jeszcze wybór. Dzięki temu wartość `is_sat f` jest równa fałszowi tylko, gdy przy każdym wartościowaniu nastąpi porażka. Zobaczmy teraz funkcję sprawdzającą, czy otrzymana formuła jest tautologią:

```
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
```

Tutaj uchwyt dla porażki znajduje się na zewnątrz – wystąpienie porażki oznacza, że istnieje wartościowanie przy którym formuła nie jest prawdziwa, a w konsekwencji nie może być tautologią. Możemy teraz napisać zgrabną funkcję, która wypisze nam czy `formula1` jest spełnialna oraz czy jest tautologią:

```
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 *)
```

Z łatwością napisaliśmy program, który korzysta z wielu efektów ubocznych jednocześnie, mimo że żaden z nich (ani uchwyty) nie wiedzą o istnieniu drugiego. Łączenie efektów jest bardzo proste, a kolejność w jakiej umieszczamy uchwyty umożliwia nam łatwe i czytelne definiowanie zachowania programu w przypadku wystąpienia któregokolwiek z efektów.

Dzięki językowi Helium przyjrzeliśmy się z bliska efektom algebraicznym oraz uchwytom, zobaczyliśmy przykłady implementacji uchwytów oraz rozwiązań prostych problemów. Jesteśmy gotowi do podjęcia próby zaimplementowania systemów kompilacji z użyciem efektów i uchwytów – czego dokonamy w następnym rozdziale.

Rozdział 5.

Systemy kompilacji z użyciem efektów algebraicznych i uchwytów

W tym rozdziale powtórzymy implementację systemów kompilacji przedstawioną w „Build systems à la carte” [1], jednak dokonamy jej w języku programowania Helium używając efektów i uchwytów. Na początku wymyślimy własne odpowiedniki abstrakcyjnych struktur z Haskella związanych z systemami, następnie zaimplementujemy wszystkie rekompilatory oraz wszystkich (poza jednym) planistów. Na koniec przyglądniemy się czym charakteryzuje się pominięty planista i poznamy przykłady innych implementacji systemów inspirowanych wynikami Mokhov'a i innych.

5.1. Pomysł, typy i idea

Przypomnijmy sobie reprezentacje składowych implementacji z Haskella oraz wprowadźmy ich odpowiedniki w Helium.

5.1.1. Zasób (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
```

Autorzy „Build systems à la carte” [1] reprezentowali Store jako typ z operacjami odczytu i zapisu trwałej informacji dla systemu oraz wartości wynikowych. Każdorazowo jednak, zasób był przechowywany w modyfikowalnym stanie. Możemy więc uprościć implementację przez scalenie zasobu z modyfikowalnym stanem przez uczynienie `Store` efektem, a działania na nim operacjami powodującymi ten efekt.

```
signature StoreEff (I: Type) (K: Type) (V: Type) =
| getInfo : Unit => I
| putInfo : I => Unit
| getValue : K => V
| putValue : K, V => Unit
```

Podobnie jak `Store` w oryginalnej implementacji, `StoreEff` jest parametryzowany typem trwałej informacji, kluczy oraz wartości wynikowych kompilacji. Równania dla niego są analogiczne jak dla zwykłego modyfikowanego stanu z dokładnością do ustalenia klucza w operacjach na wartościach wynikowych. Definiujemy ponadto uchwyt `funStoreHandler`, w którym słownik klucz–wartość zadania utrzymywane są przez funkcję – jak w przykładach w „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
```

Implementacja jest zbliżona do przykładu modyfikowalnego stanu z rozdziału 4. Dla porządku wartość początkowa trwałej informacji oraz słownika wartości jest opakowana w typ `FunStoreType I K V`.

Jako że Helium, podobnie jak inne języki używające ML-owego systemu modułów, nie posiada klas typów znanych z Haskella, definiujemy kilka sygnatur odpowiadających klasom typów użytym w oryginalnej implementacji. W przypadku `funStoreHandler` moduł o sygnaturze `Comparable K` jest używany do porównywania kluczy identyfikujących zadania. Można zauważyć, że alternatywnym rozwiązaniem byłoby reprezentowanie odpowiedników klas typów jako efekty.

```
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. Modyfikowalny stan

Implementację modyfikowalnego stanu zobaczyliśmy w przykładach w rozdziale 4 i wykorzystamy ją konstruując systemy kompilacji. Nazwy uchwytom dla stanu, w zależności od zwracanych wartości, nadajemy zgodnie z ich odpowiednikami w Haskellu – `runState`, `evalState`, `execState`. Definiujemy także proste funkcje `gets` i `modify`, które używając podanego przekształcenia odpowiednio odczytują i modyfikują stan, oraz nieco bardziej skomplikowaną funkcję `embedState`.

Definicje `gets`, `modify` oraz `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
```

Funkcja `embedState` tworzy uchwyt dla efektu modyfikowalnego stanu, w którym modyfikacje – zamiast być wykonywane przez uchwyt – są przekazywane podanym funkcjom `getter` oraz `setter`, które w czasie swojego działania mogą powodować jakiś efekt uboczny. Z takiego zanurzenia modyfikowalnego stanu w innym efekcie będziemy korzystać podczas implementacji planistów, którzy trwałą informację z zasobu będą przekazywać do rekompilatorów jako właśnie modyfikowalny stan.

Przykład wykorzystania `embedState`

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

5.1.3. Zadanie i efekt kompilacji

W oryginalnej implementacji zadanie było funkcją przyjmującą procedurę kompilacji wskazanego zadania, a wynik był zwracany w jakimś typie `f` ograniczonym przez klasę typów `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)
```

Możemy jednak zauważyć, że kompilacja zadania jest oczywistym efektem ubocznym działania systemu kompilacji, stąd w naszej implementacji zamiast przekazywać funkcję, która była przez autorów zazwyczaj nazywana `fetch`, zdefiniujemy efekt `BuildEff`, który będzie występował w czasie kompilacji zadań. Z efektem tym związana będzie jedna operacja `fetch`.

```
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 [])))
```

Zadanie będzie funkcją wymagającą informacji o instancji efektu budowania i będzie polimorficzna ze względu na typ kluczy i wartości oraz ewentualnych efektów ubocznych nie będących efektem budowania (będzie to przydatne przy implementacji rebuilderów). Zwróćmy uwagę, że definicja typu zadania nie zawiera informacji analogicznych do klasy typów `c`, której element `f` „opakowywał” wynik w oryginalnej implementacji – do tej różnicy powrócimy w dalszej części rozdziału.

5.1.4. Kompilacja, planista, rekompilator

Pozostaje zdefiniować trzy ostatnie typy związane ze wspomnianymi w podtytule obiektami.

```
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
```

Kompilacja, tak jak w oryginalnej implementacji, wymagać będzie wskazania zbioru zadań oraz klucza który ma być zbudowany. Ponadto w naszej implementacji kompilacja powoduje efekt uboczny zmiany zasobu.

Nasi planiści także będą mieli sygnatury zbliżone do swoich odpowiedników z Haskella, wzbogacone oczywiście o efekt uboczny zasobu, a także moduł definiujący opisane wcześniej podstawowe działania na kluczach i wartościach.

```
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
```

Rebuilder przypomina swój odpowiednik z oryginalnej implementacji. Jednak, zamiast zwracać zadanie ze zmienionym *constraint'em*, zadanie jest wzbogacone o dodatkowy efekt stanu mogący występować w czasie kompilacji zadania.

5.2. Przykład: system busy

Skoro ustaliliśmy jak abstrakcja systemów kompilacji w Haskellu przenosi się na naszą w Helium, możemy spróbować zaimplementować prosty system budowania `busy` przedstawiony przez autorów.

System kompilacji `busy`

```
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
```

Rdzeniem implementacji, podobnie jak oryginalnej w Haskellu, jest definicja uchwytu (tam: funkcji) dla `fetch`. Jego ciało to przetłumaczenie oryginalnej implementacji z tą różnicą, że zamiast kontynuować obliczenie niejawnie – przez zwracanie wyniku – jest ono kontynuowane jawnie przez wywołanie `resume` w ciele uchwytu.

5.3. Implementacja śladów

Podobnie jak w „Build systems à la carte", implementacje funkcji pracujących ze śladami nie są interesujące – w naszym przypadku odpowiadają oryginałom poza kilkoma szczegółami w postaci wykorzystania efektu `Writer` zamiast infrastruktury zbudowanej wokół typu `Maybe` oraz list comprehensions w Haskellu. Implementacje wraz z komentarzami dostępne są w Dodatku A oraz kodzie źródłowym.

5.4. Uruchamianie i śledzenie działań

W implementacjach planistów i rekompilatorów będziemy chcieli uruchamiać zadania oraz śledzić od jakich zadań zależy aktualnie rozważane. W tym celu, podobnie jak autorzy „Build systems à la carte", definiujemy prostą funkcję `run` oraz nieco ciekawszą `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
```

Funkcja `track` otrzymuje etykietę `‘b` uchwytu dla efektu kompilacji oraz zadanie `task`, które ma być uruchomione pod jego nadzorem, a `track` ma wyznaczyć zadania, od których `task` zależy. W tym celu konstruowany jest dodatkowy uchwyt `hTrack`, pod nadzorem którego uruchamiamy zadanie. W sytuacji, gdy uruchomione zadanie potrzebuje wyniku innego zadania, `hTrack` „przechwyci" wystąpienie `fetch` i oddeleguje wystąpienie operacji do uchwytu o etykiecie `‘b`, a następnie odnotuje, że miało miejsce wywołanie `fetch`.

Implementacja funkcji `track` jest ciekawym przykładem skonstruowania pośrednika (proxy) pomiędzy obliczeniem, które ma efekty uboczne, a właściwym dla niego uchwytem.

5.5. Implementacje systemów kompilacji

5.5.1. Excel

```
let excel {K V: Type} {‘s: StoreEff (Pair (K -> Bool) (List K)) K V} (module KV: KeyValue K
    V) =
    restarting KV dirtyBitRebuilder
```

Już na starcie widzimy, że udało nam się dopełnić obietnicy, którą postulują autorzy „Build systems à la carte” – systemy kompilacji powstają przez zaaplikowanie rekompilatora do planisty.

Funkcja `dirtyBitRebuiler` modyfikuje zadanie tak, aby przy uruchomieniu sprawdzało, czy klucz zadania jest oznaczony jako brudny. Gdy tak jest, zadanie zostanie skompilowane, w przeciwnym razie można wykorzystać wartość dostarczoną do rekompilatora, gdyż to ją zwróciłoby wykonanie pierwotnego zadania.

```
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)
```

W planiście restartującym utrzymujemy łańcuch, który ma aproksymować kolejność kompilacji, w której minimalizujemy liczbę restartów. Działanie rozpoczyna się od wykorzystania łańcucha z poprzedniej kompilacji, a jego wersję wykorzystywaną i modyfikowaną w czasie działania utrzymujemy w instancji stanu o etykiecie `‘chain`. Ponadto, w stanie `‘done` odnotowujemy, które zadania skompilowaliśmy w tej instancji procesu, aby nie musieć uruchamiać ich ponownie oraz tworzymy uchwyt, wykorzystując opisaną wcześniej funkcję `embedState` dla modyfikowalnego stanu odpowiadającego trwałej informacji systemu kompilacji.

```
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))
```

Właściwa część implementacji tego planisty składa się z uchwytu efektu kompilacji `restartingHandler` oraz funkcji `loop`. Funkcja ta wykonuje zadania w kolejności zadanej przez łańcuch z poprzedniej instancji, modyfikując zadania z użyciem rekompilatora, po czym je uruchamiając. Jednocześnie konstruowany jest nowy łańcuch, który jest wartościową zwracaną przez `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
```

W czasie kompilacji zadania wystąpienia `fetch` są przechwytywane przez uchwyt, który sprawdza, czy zadanie jest już obliczone. W przeciwnym razie modyfikuje łańcuch tak, by potrzebne zadanie znalazło się przed zadaniem aktualnie obliczanym. W uchwycie wykorzystana jest opcja dla `return`, która odnotowuje, że zadanie skończyło się kompilować, a następnie wywołuje `loop`.

5.5.2. Shake

```
let shake {K V: Type} {‘s: StoreEff (VT K V) K V} (module KV: KeyValue K V) =
    suspending KV vtRebuilder
```

W systemie Shake rebuilder wykorzystuje ślady weryfikujące. Rekompilator używając `verifyVT` sprawdza, czy zadanie jest świeże. Jeśli tak, nie musi być obliczane ponownie. W przeciwnym razie zadanie jest kompilowane pod nadzorem funkcji `track`, która akumuluje listę bezpośrednich zależności zadania, by utworzyć z nich nowe ślady do trwałego zachowania z użyciem `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))
```

Implementacja planisty wstrzymującego jest znacznie krótsza od restartującego. Utrzymujemy tylko dwa stany: pierwszy (`‘done`) dla odnotowania już skompilowanych zadań oraz drugi dla osadzenia trwałej informacji w stanie na potrzeby działania rekompilatora – podobnie jak w planiście restartującym.

```
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
```

Uchwyt `suspendingHandler` jest niezwykle prosty – wywołuje jedynie funkcję `build`, po czym wznawia kompilację z wynikiem potrzebnego zadania uzyskanym ze Store'a. Procedura `build` sprawdza, czy zadanie jest nietrywialne (czy nie jest wejściem) oraz czy nie zostało już obliczone. Wtedy konstruowane jest nowe zadanie z użyciem rekompilatora, po czym następuje jego uruchomienie. W innych przypadkach zadanie jest aktualne i na pewno nie ma potrzeby kompilować go ponownie.

5.5.3. CloudShake

```
let cloudShake {K V: Type} {‘s: StoreEff (CT K V) K V} (module KV: KeyValue K V) =
    suspending KV ctRebuilder

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)
```

W przypadku śladów konstruktywnych rebuilder sprawdza, czy podana wartość zadania jest już wśród znanych wartości. W przeciwnym razie można zwrócić dowolną znaną wartość lub – gdy żadna wartość nie jest znana – następuje kompilacja zadania. Podobnie jak w przypadku rekompilatora opartego o ślady weryfikujące, tutaj kompilacja też odbywa się ze śledzeniem zadań, od których kompilowane zależy.

5.5.4. Nix

```
let nix {K V: Type} {‘s: StoreEff (DCT K V) K V} (module KV: KeyValue K V) =
    suspending KV dctRebuilder
```

Rekompilator używający głębokich śladów konstruktywnych przypomina swoich poprzedników. Jednak – zgodnie ze swoją nazwą – sprawdza, od których zadań wejściowych w istocie badane zadanie zależy.

```
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. Nieobecny planista topologiczny

Jak zobaczyliśmy w rozdziale 3, planiści restartujący i wstrzymujący radzą sobie z zadaniami o dynamicznych jak i statycznych zależnościach. Inaczej sytuacja ma się w przypadku planisty topologicznego, który działa jedynie z zadaniami o statycznych zależnościach, które w „Build systems à la carte” są modelowane z wykorzystaniem klasy `Applicative`.

Wydaje się, że nie mamy jak uniemożliwić zadaniom inspekcję wyników wywołań `fetch`. Moglibyśmy opakowywać je w nieznany twórcy zadania typ, co wydaje się powracać do oryginalnej implementacji. Oddalamy się jednak od efektów algebraicznych i uchwytów będących tematem tej pracy, stąd nie będziemy badać dokładniej tematu modelowania statycznych zależności.

5.7. Istniejące podejścia do implementacji w innych językach

Opisane wyżej wyniki są pierwszą – według wiedzy autora – próbą implementacji systemów kompilacji inspirowanych „Build systems à la carte" używając języka z efektami algebraicznymi oraz uchwytami. W „Build systems à la carte: Theory and practice" Mokhov i inni wspominają jednak o dwóch znanych im próbach implementacji systemów kompilacji w popularnych językach programowania: Rust [12] oraz Kotlin [13]. Jak jednak zauważają, w obu przypadkach ograniczenia użytych języków doprowadziły do utracenia precyzji i schludności rozwiązań w porównaniu z oryginalną implementacją w Haskellu.

O ile brak planisty topologicznego w naszej implementacji rzeczywiście oddala nas od oryginału, o tyle planiści oraz rekompilatorzy zaimplementowani przez nas – z dokładnością do różnic składniowych języków – nie odbiegają jakością oraz czytelnością od swoich pierwowzorów.

Rozdział 6.

Podsumowanie i wnioski

Celem pracy było zapoznanie i zaciekawienie czytelnika tematem efektów algebraicznych i uchwytów oraz zaprezentowanie nowej implementacji systemów kompilacji podążając krokami autorów „Build systems à la carte". Implementacja w eksperymentalnym języku Helium miała zademonstrować, jak wygląda programowanie z efektami algebraicznymi i uchwytami oraz umożliwić zaobserwowanie, jak różni się ono od radzenia sobie z efektami ubocznymi przez użycie monad w języku Haskell.

Traktując `fetch` jako operację efektu ubocznego kompilacji, a nie jako argument do zadania, udało się nam wykorzystać możliwości języka z efektami i uchwytami w centralnej części implementacji systemów kompilacji.

Uzyskana – dzięki programowaniu z efektami i uchwytami, a nie monadami – swoboda użycia wielu efektów jednocześnie uspokoiła nasze obawy i zachęciła do eksperymentowania. Etykietowanie różnych instancji tego samego efektu umożliwiło utrzymywanie w modyfikowalnym stanie wielu wartości bez szkody dla czytelności oraz rozumieniu kodu. Dało to też możliwość tworzenia pośredników między różnymi efektami.

Reprezentacja zasobu, nad którym odbywała się kompilacja jako efektu ubocznego, nie tylko zapobiegła potrzebie każdorazowego umieszczania go w modyfikowalnym stanie, ale także lepiej oddała jego naturę bycia trwałym i zewnętrznym tworem.

Tym, co utraciliśmy, była precyzja opisu skomplikowania relacji między zadaniami. Transparentność wyników operacji z efektami, w porównaniu do „opakowywania" ich instancjami funktorów aplikatywnych lub monad, uniemożliwiła łatwe reprezentowanie zadań o statycznych zależnościach.

Problem ten był jednak łatwy do zauważenia już na początku rozważań nad własną implementacją „Build systems à la carte". Oprócz tego, w czasie implementowania systemów kompilacji, autor nie napotkał znacznych trudności w programowaniu z efektami algebraicznymi i uchwytami. Pozostałe były związane z ograniczonym

doświadczeniem autora z językiem Helium lub eksperymentalną naturą języka i chwilowymi problemami środowiska uruchomieniowego z klarownym objaśnieniem źródła niezgodności typów.

Podsumowując, programowanie z efektami algebraicznymi i uchwytami jest możliwe, jest przyjemne i uwalnia autora od ograniczeń, które dotychczas wydawały się nie do uniknięcia. Jak zobaczyliśmy, można spróbować powtórzyć wyniki przeprowadzone w znanym funkcyjnym języku programowania i z zachwytem odkryć, że implementacja z efektami i uchwytami jest równie interesująca.

Bibliografia

[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.

Dodatek A

Omówienie załączonego kodu źródłowego

A.1. Podział implementacji na pliki

- `Common.he` oraz `Signature.he` – definicje podstawowych typów oraz sygnatur,
- `Store.he` – efekt zasobu oraz uchwyty dla niego,
- `Schedulers.he` – implementacje planistów,
- `Rebuilders.he` – implementacje rekompilatorów,
- `Traces.he` – funkcje do interakcji ze śladami,
- `Track.he` – implementacje funkcji `run` oraz `track`,
- `Systems.he` – przykłady definicji omawianych systemów,
- `Spreadsheet.he` – implementacja obiektów związanych z arkuszami kalkulacyjnymi,
- `scratch.he` – demonstracja wykorzystania systemów w celu kompilacji przykładowych zadań,
- `Utils.he` – implementacje funkcji pomocniczych,
- `PatchedWriter.he` – rozszerzenie modułu `Writer` z biblioteki standardowej,
- `State.he` – implementacja efektu modyfikowalnego stanu wraz z uchwytami,
- `Logger.he` – moduł umożliwiający śledzenie działania programu,
- `SchedulersWithLogging.he` – implementacje planistów wzbogacone o wyświetlanie informacji diagnostycznych

A.2. Implementacje śladów

Dokładny opis śladów w rozdziale 5 został pominięty, gdyż nie wydają się one być ciekawą częścią implementacji systemów kompilacji. Jednak dla porządku ich implementacje są przedstawione poniżej.

A.2.1. Typ śladów

Dla śladów definiujemy typ analogiczny to tego z oryginalnej implementacji w Haskellu.

```
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))
```

Warto zwrócić uwagę, że kompilacja zadań z użyciem `fetch` w funkcjach weryfikujących lub konstruujących ślady odbywa się w kontekście specjalnego uchwytu, który zwraca nie wartości lecz ich skróty.

A.2.2. Ślady weryfikujące

```
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. Ślady konstruktywne

```
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
```

W funkcjach `constructCT` oraz `deepDependencies` wykorzystywany jest efekt `Writer`. W „Build systems à la carte” wykorzystywane były typ `Maybe` i funkcja `catMaybes` oraz list comprehensions. Autor zachęcony brakiem infrastruktury wokół `Some` (odpowiednika `Maybe` w Helium) postanowił wypróbować moduł `Writer` i zobaczyć jaki da to efekt.

A.2.4. Głębokie ślady konstruktywne

```
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
```