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

4 września 2020

Jakub Mendyk

.....

.....
(adres zameldowania)

.....

.....
(adres korespondencyjny)

PESEL: .....

e-mail: .....

Wydział Matematyki i Informatyki
stacjonarne studia I stopnia
kierunek: Indywidualne Studia Matematyczno-Informatyczne
nr albumu: 301111

Oświadczenie
o autorskim wykonaniu pracy dyplomowej

Niniejszym oświadczam, że złożoną do oceny pracę zatytułowaną *Kwalifikacja i implementacja systemów kompilacji z użyciem efektów algebraicznych* wykonałem/am samodzielnie pod kierunkiem promotora, dr. Filipa Sieczkowskiego. Oświadczam, że powyższe dane są zgodne ze stanem faktycznym i znane mi są przepisy ustawy z dn. 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2006 r. nr 90, poz. 637, z późniejszymi zmianami) oraz że treść pracy dyplomowej przedstawionej do obrony, zawarta na przekazanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

Wrocław, 4 września 2020

(czytelny podpis)

Streszczenie

. . .

. . .

Spis treści

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 jednak, 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 z efektami ubocznymi nie koniecznie 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 inferencji i weryfikacji typów do automatycznej identyfikacji modułów zawierających efekty uboczne – dzięki temu programista może łatwo wyczytać z sygnatury funkcji, które z nich 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 je bezproblemowo składać.

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 takich systemów 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" [7, 8], 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 funkcyjny, z nowatorskim rozwiązaniem jakim są efekty algebraiczne oraz zademonstrowanie – idąc śladami Mokhov'a i innych [8] – implementacji systemów kompilacji z wykorzystaniem efektów algebraicznych i uchwytów w języku programowania Helium. Jak się okazuje, wykorzystanie tych narzędzi daje schludną implementację ale także prowadzi do problemów w implementacji systemów o pewnym sposobie determinowania zależności między zadaniami.

W rozdziale drugim wprowadzony zostaje prosty model obliczeń wykorzystujący efekty algebraiczne i uchwyty. Zostaje przedstawionych kilka przykładów reprezentacji standardowych efektów ubocznych w opisanym modelu.

Rozdział czwarty rozpoczyna się zapoznaniem 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.

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. 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 argumentami n oraz dalszą częścią obliczenia κ parametryzowaną wynikiem operacji, które *może (nie musi)* zostać wykonane po jej wystąpieniu,
- 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) efektów ubocznych.

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: $op_i\ n\ \kappa\ \Rightarrow\ op_i(n, \kappa)$.

Obliczanie wartości wyrażenia przebiega następująco:

- $[\![\, \textbf{return}\ v\,]\!] = v$ – wartością **return** jest wartość wyrażenia arytmetycznego,
- $[\![\, (\lambda x.\ e)\ y\,]\!] = [\![\, e\,[x/[\![\, y\,]\!]]\,]\!]$ – aplikacja argumentu do funkcji,
- $[\![\, \textbf{if}\ v_1 = v_2\ \textbf{then}\ e_t\ \textbf{else}\ e_f\,]\!] = \begin{cases} [\![\, e_t\,]\!] & \text{gdy } [\![\, v_1\,]\!] = [\![\, v_2\,]\!] \\ [\![\, e_f\,]\!] & \text{wpp} \end{cases}$
- $[\![\, \textbf{handle return}\ v\ \textbf{with}\ H\,]\!] = [\![\, \textbf{return}\ v\,]\!]$ – uchwyt nie wpływa na wartość obliczenia, które nie zawiera efektów ubocznych,
- $[\![\, \textbf{handle}\ op_i(a, f)\ \textbf{with}\ H\,]\!] = [\![\, \textbf{handle}\ h_i\,[n/[\![\, a\,]\!],\ \kappa/f]\ \textbf{with}\ H\,]\!]$, gdzie $H = \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}$, a h_i nie ma wystąpień op_i.

Zobaczmy jak zatem wygląda obliczenie ostatniego z powyższych przykładów:

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

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:

$$\forall n\ \forall e.\ [\![\, \mathbf{handle}\ op_r(n, \lambda x.\ e)\ \mathbf{with}\ H\,]\!] = n \tag{2.3}$$

Zauważmy, że istnieje tylko jeden naturalny uchwyt spełniający tej 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*. Spróbujemy wyrazić działanie tych dwóch operacji by otrzymać modyfikowalną komórkę pamięci. Ustalmy też bardziej naturalne sygnatury operacji – $get : U \to (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$, $put : \mathbb{Z} \to (U \to \mathbb{Z}) \to \mathbb{Z}$. Ustalamy równania:

- $\forall e.\ [\![\, get(u, \lambda\_.\ get(u, \lambda x.\ e))\,]\!] = [\![\, get(u, \lambda x.\ e)\,]\!]$

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

- $\forall e.\ [\![\, get(u, \lambda n.\ put(n, \lambda u.\ e))\,]\!] = [\![\, e\,]\!]$

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

- $\forall n.\ \forall f.\ [\![\, put(n, \lambda u.\ get(u, \lambda x.\ f\ x))\,]\!] = [\![\, f\ n\,]\!]$

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

- $\forall n_1.\ \forall n_2.\ \forall e.\ [\![\, put(n_1, \lambda u.\ put(n_2, \lambda u.\ e))\,]\!] = [\![\, put(n_2, \lambda u.\ e)\,]\!]$

  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 $amb : U \to (Bool \to Bool) \to Bool$ 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 lekki 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 rzodziałach. 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 [13],

- notatki oraz seria wykładów Andreja Bauera pt. „What is algebraic about algebraic effects and handlers?” [1] dostępne w formie tekstowej oraz nagrań wideo w serwisie YouTube,
- prace Plotkina i Powera [10, 11] oraz Plotkina i Pretnara [12] – 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 [14] w serwisie GitHub.

Rozdział 3.

O systemach kompilacji (i ich klasyfikacji)

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 and Matija Pretnara język o ML-podobnej składni,
- Frank[9] [5] – 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] [3] – 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" [7, 8]. Po raz pierwszy Helium pojawia się w [3], 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 *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"
```

Sygnatura funkcji *is_negative* wyznaczona przez system typów Helium, to jak łatwo się domyśleć $Int \to Bool$. Gdy jednak zapytamy środowisko uruchomieniowe o typ funkcji *question* otrzymamy interesującą sygnaturę $Unit \to [IO]\,Unit$. 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
```

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.

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

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* oraz 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 skutkuje ono wypisaniem wiadomości na standardowe wyjście. Nie wznawiamy obliczenia, stąd wystąpienie błędu 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 wystąpieniu błędu 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 funkcji *main*, wewnątrz uchwytu, niedeterministycznie ustalamy wartości zmiennych x, y, z po czym obliczamy wartość funkcji *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 te są zaimplementowane jako:

```
handler
| return x => x
| finally f => f
end
```

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 speł-

niona. 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, a zwraca właściwy wynik obliczenia. Ten argument będzie bieżącą wartością stanu. 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 będzie nie zwykła wartość lecz funkcja, której u nas dajemy wartość stanu. Stąd podajemy jej nową wartość stanu, którą 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)
  | get () => fn s => resume  s s
  | put  s => fn _ => resume () 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.

Możemy w ten sposób definiować także 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 na raz – porażka i niedeterminizm

Na koniec rozdziału, zobaczymy jak łatwo w Helium komponuje się efekty. Definiujemy efekty niedeterminizmu oraz porażki:

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

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

oraz bardzo proste uchwyty dla tych efektów:

```
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 wyniki 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 wystąpienie 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

Rozdział 6.

Podsumowanie i wnioski

...

Bibliografia

[1] A. Bauer. What is algebraic about algebraic effects and handlers?, 2018.

[2] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[3] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[4] D. Biernacki, M. Piróg, P. Polesiuk, and F. 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.

[5] S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. *CoRR*, abs/1611.09259, 2016.

[6] C. McBride. The frank manual. *Unpublished manual*, 2012.

[7] A. Mokhov, N. Mitchell, and S. P. Jones. Build systems à la carte: Theory and practice. *Journal of Functional Programming*, 30, 2020.

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

[9] R. P. PIETERS, T. SCHRIJVERS, and E. RIVAS. Generalized monoidal effects and handlers.

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

[11] G. Plotkin and J. Power. Computational effects and operations: An overview. 2002.

[12] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *arXiv preprint arXiv:1312.1399*, 2013.

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

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