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

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 końcowy wynik staje się jednak zależny od tegoż świata zewnętrznego, a sam program nie 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 wykorzystać wtedy dedukcji i weryfikacji typów do automatycznej identyfikacji modułów zawierających efekty uboczne. Programista może łatwo wyczytać z sygnatury funkcji, że w czasiej jej działania występują efekty uboczne. Znanym przykładem takiego rozwiązania jest wykorzystanie monad w języku programowania Haskell. Niestety, jednoczesne użytkowanie dwóch niezależnych zasób reprezentowanych przez różne monady nie jest możliwe i wymaga dodatkowych struktur, takich jak transformery monad, które niosą ze sobą dodatkowe problemy. 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 sematyką 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

Przykładami 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 zadania są wykonywane w oparciu o wyniki innych podzadań, a system jest odpowiedzialny za poprawne uporządkowanie i wykonanie otrzymanych zadań. W czasie działania, system agreguje wyniki obliczeń (np. na dysku lub w pamięci) i decyduje, która zadania powinny być obliczone ponownie – np. system Make lub popularne narzędzie biurowe Excel.

W publikacjach pod tytułem „Build systems à la carte" [6] [5], 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 konstrukcję systemów kompilacji o oczekiwanych właściwościach. Platforma ta okazuje się być łatwa w implementacji w języku Haskell, a klasy typów Applicative oraz Functor odpowiadać mocy języka opisywnia 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 Makhov i innych [6] – 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 z topologicznym planistą.

Rozdział 2.

O efektach algebraicznych teoretycznie

Wprowadzimy notację służącą opisowi prostych obliczeń, która pomoże i doprowadzi nas do zrozumienia czym są efekty algebraiczne i uchwyty. Następnie przyjrzymy się, jak możemy zapisać popularne przykłady efektów ubocznych używając naszej notacji. Na koniec, czytelnikowi zostaną polecone dobre zasoby, dzięki którym będzie mógł jeszcze bardziej zagłębić się w teorii efektów algebraicznych oraz ścieżce prowadzącej do skonstruowania stojącej za nimi teorii.

2.1. Notacja

Będziemy rozważać obliczenia nad wartościami boolowskimi, liczbami całkowitymi wraz z ich równością oraz podstawowymi działaniami arytmetycznymi i funkcjami ich używającymi. Ponadto nasz model składać się będzie także z wyrażeń:

- **return** v – gdzie v jest wyrażeniem 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 $op_i : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$. 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** $\{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}_{i \in I}$ które definiują działanie (dotychczas abstrakcyjnych) efektów ubocznych.

Przykładowymi obliczeniami w naszej notacji są więc:

$$\textbf{return } 0,\ \textbf{return } 2+2,\ op_1(2, \lambda x.\ \textbf{return } x+1)$$
$$\textbf{handle } op_1(2, \lambda x.\ \textbf{return } x+1) \textbf{ with } \{\ op_1\ n\ \kappa\ \Rightarrow\ \kappa\ (2 \cdot n)\ \}$$

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,
- $[\![\, \textbf{handle } op_i(a, f) \textbf{ with } \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \} \,]\!] = [\![\, \textbf{handle } h_i\,[n/[\![\, a \,]\!],\ \kappa/f] \textbf{ with } \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \} \,]\!]$, przy czym h_i nie ma wystąpień op_i.

Zobaczmy jak zatem wygląda obliczenie ostatniego z powyższych przykładów:

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

2.2. Efekty i równania

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:

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

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 mały język z jednym równaniem ma już moc wystarczającą do opisu konstrukcji, która w większości języków nie może zaistnieć z woli programisty, a zamiast tego musi być dostarczona przez twórcę języka.

Rozważmy kolejny przykład. Dla poprawienia czytelności, operacje powodujące efekty będą miały nazwy *get* oraz *put*. Spróbujemy wyrazić działanie tych dwóch operacji by otrzymać modyfikowalną komórkę pamięci. Ustalmy też bardziej naturalny sygnatury operacji – $get : U \to \mathbb{Z}$, $put : \mathbb{Z} \to U$. Gdzie U jest nowym typem – jednostką – zamieszkałym przez pojedyńczą wartość u. Zastanówmy się jakie warunki takie operacje powinny spełniać:

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

  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 pamięta 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.

[1]

Rozdział 3.

O systemach kompilacji (i ich klasyfikacji)

Rozdział 4.

Efekty algebraiczne i uchwyty w praktyce

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

[6] 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.

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