Kwalifikacja i implementacja systemów kompilacji z użyciem efektów algebraicznych

Jakub Mendyk

Instytut Informatyki Uniwersytetu Wrocławskiego

5 września 2020

Plan prezentacji

1. Wstęp pracy
   - Problemy z efektami ubocznymi
   - Radzenie sobie z efektami ubocznymi
   - Systemy kompilacji
2. Efekty algebraiczne i uchwyty
   - W teorii
   - W praktyce
3. Podsumowanie i wnioski

Efekty uboczne

Zalety

+ komunikacja z innymi systemami
+ trwała pamięć – system plików, bazy danych
+ interaktywność

Wady

– zależność od świata zewnętrzengo
– utrudnione rozumienie, brak modularności
– częstsze pomyłki

Efekty uboczne są problematyczne

Pomysł

Rozdzielić program na część czystą oraz część mającą efekty uboczne.

Efekty uboczne są problematyczne

Pomysł

Rozdzielić program na część czystą oraz część mającą efekty uboczne.

Problem

Musimy zaufać autorowi, że funkcja rzeczywiście nie powoduje efektów ubocznych.

Radzenie sobie z efektami ubocznymi

Potrzebujemy znaleźć kogoś, kto będzie pilnował czy funkcje, które twierdzą że nie mają efektów ubocznych rzeczywiście takie są.

Radzenie sobie z efektami ubocznymi

Potrzebujemy znaleźć kogoś, kto będzie pilnował czy funkcje, które twierdzą że nie mają efektów ubocznych rzeczywiście takie są.

Pomysł

Wykorzystajmy system typów – jest dobry w sprawdzaniu czy deklaracje programisty (adnotacje typów) są zgodne ze stanem faktycznym (implementacjami funkcji). Inferencja wyręczy nas od potrzeby pisania typów w wielu przypadkach (w przeciwieństwie do np. języka C).

Monady

+ umożliwiają bezpieczne programowanie z efektami
+ informacje o efektach ubocznych w sygnaturze
+ efekty nie mogą „uciec”
– potrzeba transformerów monad by użyć wielu efektów naraz
– modularność wciąż problematyczna

Efekty algebraiczne i uchwyty

+ umożliwiają bezpieczne programowanie z efektami
+ informacje o efektach ubocznych w sygnaturze
+ efekty nie mogą „uciec”
+ łatwość użycia wielu efektów jednocześnie
+ modularność i przejrzystość

Systemy kompilacji

- interakcja z zewnętrznymi zasobami
- uznawane za zło konieczne, zbyt skomplikowane
- powszechne wykorzystanie w „przemyśle”
- rzadko obiekt zainteresowań badaczy

Build Systems à la Carte

ANDREY MOKHOV, Newcastle University, United Kingdom
NEIL MITCHELL, Digital Asset, United Kingdom
SIMON PEYTON JONES, Microsoft Research, United Kingdom

Build systems are awesome, terrifying – and unloved. They are used by every developer around the world, but are rarely the object of study. In this paper we offer a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in landscape rather than as isolated phenomena. By teasing apart existing build systems, we can recombine their components, allowing us to prototype new build systems with desired properties.

CCS Concepts: • **Software and its engineering**; • **Mathematics of computing**;

Additional Key Words and Phrases: build systems, functional programming, algorithms

ACM Reference Format:
Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à la Carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (September 2018), 29 pages. https://doi.org/10.1145/3236774

1 INTRODUCTION

Build systems (such as Make) are big, complicated, and used by every software developer on the planet. But they are a sadly unloved part of the software ecosystem, very much a means to an end, and seldom the focus of attention. For years Make dominated, but more recently the challenges of scale have driven large software firms like Microsoft, Facebook and Google to develop their own build systems, exploring new points in the design space. These complex build systems use subtle algorithms, but they are often hidden away, and not the object of study.

In this paper we offer a general framework in which to understand and compare build systems, in a way that is both abstract (omitting incidental detail) and yet precise (implemented as Haskell code). Specifically we make these contributions:

https://dl.acm.org/doi/pdf/10.1145/3236774

Prosty i nieformalny rachunek

Proste wyrażenia:

- **return** v,
- **if** $v_1 = v_2$ **then** e_t **else** e_f,
- abstrakcyjne operacje – $\{op_i\}_{i \in I}$,
- uchwyty – **handle** e **with** $\{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}_{i \in I}$.

Prosty i nieformalny rachunek

Proste wyrażenia:

- **return** v,
- **if** $v_1 = v_2$ **then** e_t **else** e_f,
- abstrakcyjne operacje – $\{op_i\}_{i \in I}$,
- uchwyty – **handle** e **with** $\{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}_{i \in I}$.

Zachodzące równoważności:

- $(\lambda x.\ e_1)\ e_2 \equiv e_1\,[{}^{x}/_{e_2}]$,
- **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,
- **handle** $op_i(a, \lambda x.\ e)$ **with** $H \equiv h_i\,[{}^{n}/_{a},\ {}^{\kappa}/_{\lambda x.\ \textbf{handle}\ e\ \textbf{with}\ H}]$, gdzie $H = \{\ op_i\ n\ \kappa\ \Rightarrow\ h_i\ \}$.

Prosty i nieformalny rachunek

Przykład

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

Równania dla efektów

Porażka:

- $\forall n\ \forall e.$ **handle** $op_r(n, \lambda x.\ e)$ **with** $H \equiv n$

Modyfikowalny stan:

- $\forall e.\ get(u, \lambda \_.\ get(u, \lambda x.\ e)) \equiv get(u, \lambda x.\ e)$
- $\forall e.\ get(u, \lambda n.\ put(n, \lambda u.\ e)) \equiv e$
- $\forall n.\ \forall f.\ put(n, \lambda u.\ get(u, \lambda x.\ f\ x)) \equiv f\ n$
- $\forall n_1.\ \forall n_2.\ \forall e.\ put(n_1, \lambda u.\ put(n_2, \lambda u.\ e)) \equiv put(n_2, \lambda u.\ e)$

Inne przykłady: niedeterminizm

Sprawdzanie spełnialności formuły boolowskiej:

$$\textbf{handle}\ amb(u, \lambda x.\ amb(u, \lambda y.\ amb(u, \lambda z.\ \phi(x, y, z))))$$
$$\textbf{with}\ \{\ amb\ u\ \kappa\ \Rightarrow\ \kappa\ (T)\ \textbf{or}\ \kappa\ (F)\ \}$$

Sprawdzanie tautologiczności:

$$\textbf{handle}\ amb(u, \lambda x.\ amb(u, \lambda y.\ amb(u, \lambda z.\ \phi(x, y, z))))$$
$$\textbf{with}\ \{\ amb\ u\ \kappa\ \Rightarrow\ \kappa\ (T)\ \textbf{and}\ \kappa\ (F)\ \}$$

Ten sam efekt, inne zachowanie dzięki uchwytom.

Efekty i uchwyty

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 (...), otrzymaliśmy niezwykle silne narzędzie umożliwiające (...) samodzielne konstruowanie zaawansowanych efektów ubocznych.

Biblioteki:

- extensible-effects (Haskell)
- fused-effects (Haskell)
- atnos-org/eff (Scala)
- Effects (Idris)

Języki programowania:

- Eff
- Frank
- Koka
- Helium

Przykład programu w Helium

```
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"
```

Efekt błędu

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

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
```

Jeden efekt, różne uchwyty

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

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

Niedeterminizm

```
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"
```

Niedeterminizm 2

```
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")
```

Modyfikowalny stan

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

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

Rekursja

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

```
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
```

Rekursja

```
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"
```

Wiele efektów naraz

```
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
```

Wiele efektów naraz – sprawdzanie spełnialności

```
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
```

Wiele efektów naraz – sprawdzanie tautologiczności

```
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
```

Wiele efektów naraz

Łączenie efektów jest bardzo proste, a kolejność w jakiej umieszczamy uchwyty umożliwia łatwe i czytelne definiowanie zachowania programu w przypadku wystąpienia któregokolwiek z efektów.

Podsumowanie i wnioski

Programowanie z efektami algebraicznymi i uchwytami:

- jest możliwe,
- jest przyjemne
- i uwalnia autora od ograniczeń, które dotychczas wydawały się nie do uniknięcia.

Obserwacje po implementacji

- swoboda użycia wielu efektów uspokoiła 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

Dziękuję za uwagę