

# Kwalifikacja i implementacja systemów kompilacji z użyciem efektów algebraicznych

Jakub Mendyk

Instytut Informatyki Uniwersytetu Wrocławskiego

7 września 2020

# Plan prezentacji

- 1 Wstęp pracy
  - Problemy z efektami ubocznymi
  - Radzenie sobie z efektami ubocznymi
  - Systemy kompilacji
- 2 Efekty algebraiczne i uchwyt
  - W teorii
  - W praktyce
- 3 O systemach kompilacji
  - Przykłady systemów
  - Abstrakcyjnie o systemach kompilacji
  - Reprezentacja
- 4 Systemy z użyciem efektów i uchwytów
  - Reprezentacja
  - Implementacje planistów
  - Implementacje rekompilatorów
- 5 Podsumowanie i wnioski

# Efekty uboczne

## Zalety

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

## Wady

- zależność od świata zewnętrznego
- 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 uchwyt

- + 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 –  $\{op_i\}_{i \in I}$ ,
- uchwyt – **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 –  $\{op_i\}_{i \in I}$ ,
- uchwyt – **handle**  $e$  **with**  $\{op_i \mid \kappa \Rightarrow h_i\}_{i \in I}$ .

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

# Prosty i nieformalny rachunek

## Przykład

```

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

```

# Równania dla efektów

Porażka:

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

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:

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

Sprawdzanie tautologiczności:

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

Ten sam efekt, inne zachowanie dzięki uchwytom.



# Efekty i uchwyt

Konstrukcja efektów, operacji i uchwytów tworzy dualny mechanizm, w którym operacje są producentami efektów, a uchwyt 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 uchwyt

```
let abortOnError =  
  handler  
  | error () => printStr "Error occurred!\n"  
end  
  
let warnOnError =  
  handler  
  | error () => printStr "Error occurred, 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 uchwyt umożliwia łatwe i czytelne definiowanie zachowania programu w przypadku wystąpienia któregoś z efektów.

# Make

- bardzo popularny
- szeroko dostępny
- względnie starym
- pliki *makefile*'e

```
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
```

- statyczne zależności: kompilacja przebiega tak samo, niezależnie od wyników podzadań
- naturalna kolejność kompilacji: porządek topologiczny – każde zadanie skompilowane co najwyżej raz

## Definicja (Minimalność)

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.

## Przykład

Make jest minimalny

## Definicja (Wejście/zadanie wejściowe)

Jeśli zadanie nie ma zdefiniowanego sposobu zbudowania, na przykład *util.h* mówimy, że jest wejściem lub zadaniem wejściowym w tej konfiguracji.



# Excel

Excel to system kompilacji:

- komórki – zadania wejściowe
- formuły – definicja sposobu budowania

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

A2: 20

Dynamiczne zależności – bez sprawdzenia C1 nie da się określić od jakich zadań zależy B1.

# Nomenklatura

- Zasób (Store) – modyfikowany przez system (Excel – komórki, Make – system plików)
- Trwała informacja – pamięć systemu między uruchomieniami

## System kompilacji

**Cel:** zmodyfikowanie stanu zasobu w takich sposób, by wartość związana ze wskazanym przez użytkownika kluczem stała się aktualna.

**Otrzymuje:** definicje zadań, zasób na którym działa oraz klucz, który ma zostać zaktualizowany. Po zakończeniu działania, wartość w Store związana ze wskazanym kluczem ma być aktualna.

# Zadania i zasób

```
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)
```

```
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
```

## Przykład

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

# System kompilacji

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

## Przykład

```
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
```

# Rodzaje zależności

Klasy typów to rodzaje zależności!

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

- $f = \text{Functor}$  – sekwencyjne obliczenia
- $f = \text{Applicative}$  – statyczne zależności
- $f = \text{Monad}$  – dynamiczne zależności
- $f = \text{Selective}$  – opcje wyboru znane statycznie

# Określanie zależności

```
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

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
```

# Klasyfikacja

System kompilacji = planista + rekompilator

```
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
```

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

# Zasób (Store)

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

```
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
```



# Modyfikowalny stan

```
let embedState {E: Effect} {V: Type} (getter: Unit ->[E] V) (setter: V ->[E]  
  Unit) =  
  handler  
  | get () => resume (getter ())  
  | put s => setter s; resume ()  
end  
  
handle 'store in  
  (* ... *)  
  handle 'state in  
    (* ... *)  
    with embedState (getInfo 'store) (putInfo 'store)  
      (* ... *)  
  with (* ... *)
```

# Zadanie i efekt kompilacji

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

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

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

(brak ograniczenia rodzaju zależności, o tym później)

# Kompilacja, planista, recompiler

```

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

```

```

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

```

## Przykład: system 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
```

# Planista restartujący

```

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))

```

# Planista restartujący (cd.)

```

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

```

# Planista wstrzymujący

```

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

```

# Rekompilatory z brudnym bitem oraz śladami weryfikującymi

```
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)

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))
```



# Rekompilatory ze śladami konstruktywnymi

```
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)
```

```
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)
```

## Co z planistą topologicznym?

Wydaje się, że nie mamy jak uniemożliwić zadaniom inspekcję wyników wywołań `fetch`.

Opakowywanie w nieznany twórcy zadania typ – powrót do oryginalnej implementacji – mało satysfakcjonujące.

# 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*