Raport tygodniowy 6-10 lipca

Jakub Mendyk

11 lipca 2020

1. Na dobry początek odświeżyłem sobie sobie treść „Build Systems à la Carte”.

2. Wybrałem zapewne ostateczne reprezentacje efektu budowania i zasobów budowania (`Store`):

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

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

Są to raczej naturalne sposoby reprezentacji. Zastanawiałem się, czy może byłoby lepiej gdyby `Store` był typem danych z operacjami – co rozwiązałoby problemy, o których wspomnę dalej – ale na razie spróbuję zostawić `Store` tak jak jest.

1. Zaimplementowałem uchwyty efektu `BuildEff` dla planistów `busy` oraz `suspending` – gdyż wydają się być dobrą wprawką do implementacji pozostałych schedulerów i rebuilderów. Schedulery reprezentuję właśnie jako uchwyty efektu `BuildEff`, nie zdecydowałem jeszcze jak reprezentować rebuildery – też jako uchwyty czy może lepiej jako zwykłe funkcje. W czasie semestru jako ideał rozwiązania widziałem konstrukcję postaci

```
handle
    handle task () with scheduler tasks
with rebuilder
```

Na razie mam działającą konstrukcję

```
handle
    withSuspending task_id tasks
with funInfoHandler sComp 0 initState
```

gdzie `funInfoHandler` jest uchwytem dla efektu `Store`.

1. Problem z `handle ... with` vs `handler ... end`.

Załóżmy że chcemy mieć efekt `Counter` z operacją `count`, który zlicza liczbę wywołań tejże procedury i zwraca parę (wartość obliczenia, liczba wywołań `count`). Taki efekt możemy zaimplementować następująco:

```
signature Counter =
| count : Unit => Unit

let fun () =
    count ();
    count ();
    count ();
    42

let hCount1 =
    handler
    | count () => fn n => resume () (n+1)
    | return x => fn n => (x, n)
    | finally f => f 0
    end

(* Uzywajac handlera *)

let _ =
    let (x, n) = handle fun () with hCount1 in
    printInt x; printInt n

(* Lub tworzac handler ad-hoc: *)

let _ =
    let (x, n) = handle fun () with
                 | count () => fn n => resume () (n+1)
                 | return x => fn n => (x, n)
                 | finally f => f 0
                 end in
    printInt x; printInt n
```

W obu przypadkach, handlery przekazują stan w ostatnim argumencie `resume`. Łatwo zauważyć, że można by wykorzystać już istniejący efekt `State` do przekazywania stanu.

Tworząc handler ad-hoc, możemy wykorzystać efekt stanu następująco:

```
let _ =
    let (x, n) = handle
                     handle fun () with
                     | count () => put (1 + get ()); resume ()
                     | return x => (x, get ())
                     | finally f => f
                     end
                 with execState 0 in
    printInt x; printInt n
```

Wewnętrzny handler tworzony ad-hoc obsługuje efekt `Counter`, jednocześnie wykonując działania z efektem ubocznym `State`, który jest obsługiwany przez zewnętrzną konstrukcję `handle (...) with execState 0`. Chcielibyśmy jednak nie musieć za każdym razem tworzyć ad-hoc han-

dlera dla efektu `Counter`, tylko przypisać go do zmiennej tak jak zrobiliśmy z `hCount1`. Nie chcemy jednak robić handlera:

```
let hCount2 =
    handler
    | count () => put (1 + get ()); resume ()
    | return x => (x, get ())
    end
```

bo efekt `State` wycieka poza `hCount2`, a nie chcemy obarczać użytkownika obowiązkiem obsługi tego efektu. Pewnym łatwym rozwiązaniem jest

```
(* niestety
   let withCounting t = (...)
   nie dziala:
       error: This expression has effect
       [<unnamed>], but an expression was expected of effect ['b,'a]. *)

let withCounting {T: Type} (t: ((`e: Counter) -> Unit ->[`e] T)) =
    handle
        handle t () with
        | count () => put (1 + get ()); resume ()
        | return x => (x, get ())
        | finally f => f
        end
    with execState 0

let _ =
    let (x, n) = withCounting fun in
    printInt x; printInt n
```

jednak trzeba podać typy explicite co może być problematyczne jeśli obliczenie `t` ma inne nieznane efekty, i całościowo takie rozwiązanie wygląda trochę mało satysfakcjonująco.

Tutaj jest właśnie problem - jak zrobić handler (używając konstrukcji `handler ... end`), tak by złapać efekt `State` jeszcze w definicji `hCount2`. Przetestowałem kilka pomysłów:

```
let hCount21 =
    let handle with execState 0 in
        handler
        | count () => put (1 + get ()); resume ()
        | return x => (x, get ())
        end
```

```
let hCount22 =
    handle
        handler
        | count () => put (1 + get ()); resume ()
        | return x => (x, get ())
        end
    with execState 0
```

```
let hCount23 =
    let h = handler
            | count () => put (1 + get ()); resume ()
            | return x => (x, get ())
            end
    in handle h with execState 0
```

```
let hCount24 =
    let h () = handler
            | count () => put (1 + get ()); resume ()
            | return x => (x, get ())
            end
    in handle h () with execState 0
```

```
let hCount25 =
    handler
    | count () => put (1 + get ()); resume ()
    | return x => (x, get ())
    | finally f => handle f with execState 0
    end
```

niestety typechecker ich nie akceptuje – czy to na poziomie samej definicji, czy w miejscu użycia ze względu na nieudane złapanie efektu. **Pytanie:** Czy Helium ma możliwość, by w handlerze wykonywać operacje z efektami ubocznymi, a jednocześnie złapać te efekty w definicji handlera?

1. W „Build Systems à la Carte" autorzy poczynili ciekawą obserwację, że constraint `Applicative` odpowiada systemom ze statycznym określaniem zależności, podczas gdy `Monad` tym z dynamicznymi zależnościami. Niestety nie widzę, by korzystając z efektów algebraicznych dało się zrobić podobnie ciekawą konstrukcję. Najlepsze co na razie wymyśliłem to nowy rodzaj sygnatury efektów, tym razem z dwiema operacjami

```
signature ApplicativeLikeEff (K: Type) (V: Type) =
| fetch : K => V
| require: [K] => Unit
```

gdzie `require` musiałoby zostać wywołane przez jakimkolwiek `fetch` i informowałoby o zależnościach potrzebnych do zbudowania zadania. Niestety takie rozwiązanie byłoby mało solidne, bo ewentualną pomyłkę twórcy zadania – zapomnienie dodania zależności do wymaganych – można by wychwycić dopiero w czasie działania.

To kończy raport na ten tydzień. W tygodniu 13–17 lipca planuję przemyśleć układ początku tekstu pracy – opis czym są efekty algebraiczne, systemy budowania, co autorzy opisują i zauważają w „Build Systems à la Carte" i kontynuować implementację schedulerów i rebuilderów.