Formalnie zweryfikowane programowanie z monadami w Coqu

Wojciech Kołowski

10 września 2019

Motywacje praktyczne

- Monady bywają przydatne - parę razy potrzebowałem ich w kontekście opcji czy list, żeby za dużo nie pisać.
- W bibliotece standardowej Coqa nie ma monad.
- Próba wyszukania słowa "monad" na liście Coqowych pakietów `https://coq.inria.fr/opam/www/` nie daje żadnej sensownej odpowiedzi (choć jest kilka wystąpień).
- W Coqu nie ma żadnego pomysłu na zarządzanie efektami.
- Być może dlatego, że w Coqu nie ma żadnych "prawdziwych" efektów w stylu IO (choć ostatnio pojawiły się biblioteki/pakiety, które dają Coqowi IO).

Motywacje dydaktyczne

- Człowiek ucząc się Haskella poznaje głównie sposób użycia monad w praktyce.
- Gorzej z kwestiami takimi jak prawa: choć Haskell reklamuje się jako przyjazny rozumowaniu równaniowemu, to dowodzenie praw zazwyczaj odbywa się w komentarzach albo postach na blogach.
- Kto normalny dowodzi praw, jeżeli typechecker go nie zmusza? Ja nie.
- W Coqu jest wprost przeciwnie - typechecker zmusza do dowodzenia praw, a CoqIDE umożliwia robienie tego w wygodny, interaktywny sposób.
- (Re)implementacja monad w Coqu daje więc dużo większe pole do nauki niż Haskell.

Motywacje teoretyczne

- Prawa monad dużo lepiej wyglądają, gdy prezentuje się je za pomocą >=> zamiast >>=, więc dlaczego ludzie tak nie robią?
- Czy stare Haskellowe `Monad` to to samo co nowe `Monad`, którego nadklasą jest `Applicative`?
- Czy faktycznie wszystkie dowody przechodzą tak gładko, jak chcieliby Haskellowi miłośnicy rozumowań równaniowych?
- Czy na temat efektów da się w ogóle formalnie rozumować wewnątrz języka?

Co i jak z tego wyszło

- Początki projektu sięgają kwietnia 2017 – była to wówczas próba załatania braku w standardowej bibliotece Coqa różnych wygodnych rzeczy znanych z Haskella.
- Była to pierwsza tego typu Coqowa biblioteka albo nie umiem szukać (choć w międzyczasie powstała też podobna biblioteka z nieco innymi celami oraz używająca innych mechanizmów abstrakcji).
- Siła rzeczy (oraz kontakty z promotorem i seminarium z efektów algebraicznych) popychały ewolucję projektu w kierunku czegoś, co zaczęło przypominać mądrzejszą kopię Haskellowej biblioteki `mtl`.
- Myślę, że efekty sa całkiem dobre (przynajmniej dopóki patrzymy tylko na proste przykłady – co by było w dużej skali, nie wiadomo).

Do kogo adresowana jest praca?

- Wychodzę z założenia, że piszę po to, aby być czytanym.
- Oficjalnym celem pracy jest zgarnięcie papierka zwanego dyplomem inżyniera, a nieoficjalnym – nauczenie czytelnika (i samego siebie) czegoś wartościowego.
- Ciekawym skutkiem tego podejścia jest to, że każdy rozdział (a czasem nawet pojedyncze sekcje) są adresowane do zupełnie różnych grup odbiorców, choć fakt ten może być niewidzialny.
- Recenzje pracy są pozytywne, więc chyba nie jest taka zła.

Weryfikacja hard- i software'u

- Rozdział 1 pisałem z myślą o recenzencie zanim jeszcze dowiedziałem się, kto nim jest.
- Jego cel jest prosty – uzasadnienie potrzebności i ważności interesujących mnie rzeczy za pomocą sprytnej retoryki.
- Jeżeli spec od formalnej weryfikacji chce uzasadnić swoje istnienie, musi wspomnieć coś o spadających samolotach.
- Koniunktura sprzyja takiemu postawieniu sprawy, gdyż ostatnimi czasy namnożyło się bardzo poważnych błędów, jak Meltdown, Spectre czy Heartbleed. Spadające samoloty też były, więc retoryka jest nawet zgodna z prawdą.

Weryfikacja matematyki

- Podrozdział 1.2 to próba przemówienia do rozumu matematykom.
- Matematykom wydaje się, że logika jest sztuczna i nie odpowiada praktyce, a formalizacja nie jest warta zachodu.
- Klasycznym kontrprzykładem jest dowód twierdzenia o czterech barwach, wymagający sprawdzenia setek przypadków.
- Innym ciekawym przypadkiem są perypetia Vladimira Voevodskyego, laureata medalu Fieldsa, który w 1989 opublikował pewną pracę o ∞-grupoidach i teorii homotopii. Znalezienie w niej błędu zajęło innemu ekspertowi 9 lat, ale przez kolejne 15 lat Voevodsky nie wierzył w jego argumenty.

Efekty

- Rozdział 2 pisałem z myślą o typowym programiście, który niespecjalnie zdaje sobie sprawę z konieczności mądrego zarządzania efektami.
- W sekcjach 2.1 i 2.2 staram się wyjaśnić podstawowe pojęcia i prezentuję dość prymitywny przykład.
- Celem tego jest zbudowanie jakiejś prostej ideologii, przez pryzmat której zwykły programista może postrzegać to, co dzieje się w jego języku.
- W sekcji 2.3 porównuję 3 podejścia do efektów:
  - Monady, transformatory, klasy efektów (jak np. `MonadState` etc.) i reszta znana z `mtla`.
  - `extensible-effects`
  - Efektry algebraiczne, czyli najnowszy krzyk mody.

Abstrakcja i inżynieria dowodu

- Większa część rozdziału 3 (3.1, 3.2, 3.3) jest adresowana do mnie samego. Strasznie irytuje mnie mnogość powiązanych mechanizmów abstrakcji dostępna w Coqu i to, że żadne prace (o ile mnie pamięć nie myli) nie tłumaczą swojego wyboru mechanizmów abstrakcji. Musiałem coś z tym zrobić – wyszło porównanie klas typów, struktur kanonicznych i modułów.
- Pozostała część rozdziału (3.4 i 3.5) była adresowana do inżynierów oprogramowania. Jej celem było podkreślenie roli nowej dziedziny inżynierii, która wyłania się z formalizacji – inżynierii oprogramowania. Tu aksjomaty stają się częścią architektury, a automatyzacja dowodów – czymś analogicznym np. do systemów ORM (choć porównanie jest być może naciągane).

Implementacja

- Rozdział 4 skierowany jest głównie do wnikliwego czytelnika, który zna się na rzeczy, np. recenzenta.
- Jego celem jest sztampowy opis implementacji połączony z innowacyjnym opisem różnych dziwnych rzeczy, które odkryłem, a których się nie spodziewałem.
- Okazuje się, że znane z Haskella klasy wymagają nowych prawy (bo naturalność nie działa), a te sprawiają np., że nowe zestawy praw nie są minimalne – niektóre prawa da się wyprowadzić z innych.

Inne odkrycia

- Trzeba było też zrewidować kilka metod implementacji. Dla przykładu, wolne monady znane z Haskella nie są legalne w Coqu, więc trzeba posiłkować się kodowaniem Churcha.
- Ciekawym doświadczeniem było empiryczne poszukiwanie praw dla klasy `WriterT`.
- Były też i porażki – np. nie udało mi się pokazać, że `FreeT` spełnia prawa `MonadExcept`, czyli efektu wyjątków.