Ekstrakcja beztypowego ewaluatora

Paweł Wieczorek

24 września 2011

1 Wstęp.

Podczas prac nad formalizacją NBE dla typów zależnych napotkałem problem nad ewaluacją termów. Idea była taka, aby ewaluować typy i termy w pewnej beztypowej dziedzinie D. Przykładowo, dla prostego rachunku lambda z typami prostymi dobra byłaby dziedzina spełniająca równość $D \cong [D \to D]$.

W języku funkcyjnym moglibyśmy posłużyć się poniższymi definicjami do reprezentacji takiej dziedziny oraz ewaluacji

```
type dom
  = Fun of (dom -> dom)

let app df dx = match df with
 | Fun f -> f dx
;;

let eval tm env = match tm with
 | Abs body  -> Fun (fun d => eval tm (dext env d))
 | App tf tx -> app (eval tf env) (eval tx env)
...
;;
```

Chciałem aby moja praca nad NBE umożliwiła ekstrakcję certyfikowanych funkcji do normalizacji i testowania konwertowalności, a więc musiałem odpowiednio sformalizować taką dziedzinę.

Przeniesienie tych definicji bezpośrednio do Coqa okazało się niemożliwe, mimo że język w tej teorii typów to też język funkcyjny oraz ma typy indukcyjne. Język ten musi być totalny, a więc nałożone są pewne ograniczenia nawet na definiowanie typów indukcyjnych - muszą być ściśle pozytywne. Oznacza to tutaj, że nie mogę zdefiniować typu używając jako parametrów konstruktora funkcji $(D \to D)$. Można wtedy definiować nienormalizujące się termy:

```
Inductive D : Set :=
| Fun: (D -> D) -> D
.

Definition app df dx = match df with
 | Fun f => f dx
.

Definition omega := Fun (fun d => app d d).
Definition Omega := app omega omega.
```

2 Reprezentacja dziedziny.

Rozwiązaniem reprezentacji tej dziedziny jest defunkcjonalizacja ewaluatora, tzn zamiast pamiętać funkcję to pamiętam ciało funkcji i jej środowisko. Pomysł ten zobaczyłem w pracy [4] dotyczącej NBE dla

λC, niestety nie została tam podana technika zamiany wykresu funkcji na normalną funkcję w teorii typów.

Poniżej definicje indukcyjnych typów w Coqu, rozszerzyłem język o liczby naturalne (0 i następnik, bez rekursora) by móc robić bardziej namacalne testy, gdy uda mi się ekstrakcja.

```
Inductive Tm : Set :=
| TmVar: nat -> Tm
| TmApp: Tm -> Tm -> Tm
| TmAbs: Tm -> Tm
| Tm0  : Tm
| TmS  : Tm
.

Inductive D : Set :=
| DClo : Tm -> DEnv -> D
| DNat : nat -> D
| DS : D
with DEnv : Set :=
| DE : (nat -> D) -> DEnv
.

Definition Dlookup (denv:DEnv) i :=
  let (f) := denv in f i
.

Definition Dext (denv:DEnv) d :=
  DE (fun i =>
    match i with
      | 0 => d
      | S k => let (f) := denv in f k
    end
  )
.
```

Z reprezentacją ewaluatora oraz aplikacji był jeszcze problem do rozwiązania, znów totalność. Z jednej strony funkcje nie były totalne, więc nie mogłem je zaprogramować w teorii typów, z drugiej by zrobić ekstrakcję potrzebowałem mieć jednak funkcje.

W książce *Coq'Art* jest dział poświęcony ogólnej rekursji (rozdział 15), a w nim przykład jak ekstraktować funkcję częściową - mianowicie logarytm (rozdział 15.4). Idea to zrobienie dodatkowego predykatu który by oznaczał dziedzinę funkcji oraz po którym byłaby rekursja strukturalna. Predykat byłby w uniwersum `Prop`, a więc zostałby wymazany przy ekstrakcji.

```
Inductive log_domain : nat - Prop :=
| log_domain_1 : log_domain 1
| log_domain_2 : forall p, log_domain (S (div2 p)) -> log_domain (S (S p))
.
```

Teraz, funkcja powinna mieć typ `log: forall  n , log_domain n -> nat`. Tzn, mamy funkcję całkowitą ograniczoną do tych liczb, które spełniają odpowiedni predykat. Zanim napiszemy funkcję trzeba jeszcze zastanowić się nad zrobieniem rekursji. Nie można zrobić dopasowania wzorca na predykacie `log_domain` bo to byłaby niedozwolona eliminacja `Prop` gdy cel nie jest w `Prop`. Rozwiązanie to napisanie swoich projekcji na tym typie:

```
Theorem log_domain_invert: forall n p,
 log_domain_ n -> n = S (S p) -> log_domain (S (div2 p)).
Proof.
...
Defined.
```

Ważne są dwie rzeczy przy definiowaniu tych pomocniczych funkcji. Ponieważ mają one służyć do wyciągania pod-termu by móc zrobić rekursję to a) skrypt nie może kończyć się poleceniem `Qed`. Wtedy funkcja log_domain_invert będzie jedynie zmienną która nigdy nie jest rozwijana - przez co termination-checker nie będzie widział że robimy rekursję strukturalną; b) term zbudowany przez skrypt musi zwracać pod-term, nie może to być dowolny dowód bo wtedy nie będzie to rekursja strukturalna.

```
Fixpoint log (x : nat) (h : log_domain x) {struct h} : nat :=
 match x as y with return x = y -> nat with
   | 0       => fun h' => False_rec nat (log_domain_non_0 x h h')
   | S 0     => fun h' => 0
   | S (S p) => fun h' =>
       S (log
           (S (div2 p))
           (log_domain_inv x p h h')
         )
 end (eq_refl)
.
```

Zdaje się, że podobnie działa robienie dobrze ufundowanej rekursji w Coqu, wtedy rekursja jest po predykacie *Acc*, który także jest w *Prop*.

Mimo dobrego przykładu nadal nie umiałem napisać tego ewaluatora. Problem jaki miałem to większa złożoność funkcji ewaluacji niż ten logarytm.

Powyższy przykład z logarytmem to predykat który mówi jedynie coś o argumentach funkcji, tzn tylko i wyłącznie o dziedzinie - nie ma tutaj potrzeby aby posługiwać się wynikami funkcji z rekurencyjnych wywołań. W przypadku beztypowego ewaluatora tak nie ma.

Rozważmy przykład: dane domknięcie `DClo body denv` oraz `dx : D` jest w dziedzinie aplikacji gdy ewaluacja termu `body` z rozszerzonym środowiskiem `Dext denv dx` jest w dziedzinie ewaluatora. To dało się wyrazić bez trudu - problem jest z aplikacją termów. Stwierdzenie, że `TmApp tf tx` ze środowiskiem `denv` jest w dziedzinie ewaluatora, wymaga aby powiedzieć że `tf` oraz `tx` są z tym środowiskiem w dziedzinie ewaluatora oraz, że wyniki ewaluacji tych termów są w dziedzinie aplikacji.

```
Inductive EvalDom : Tm -> Env -> Prop :=
...
 | evalApp: forall tf tx denv,
   EvalDom tf denv ->
   EvalDom tx denv ->
   AppDom ? ? ->
   EvalDom (TmApp tf tx) denv
with AppDom : D \to D \to Prop :=
 | appClo : forall body denv dx,
   EvalDom body (Dext denv dx) ->
   AppDom (DClo body denv) dx
.
```

Widać, że nie należy rozważać relacji `EvalDom : Tm -> Env -> Prop`, lecz cały wykres funkcji, czyli relację `EvalGraph : Tm -> Env -> D -> Prop`.

Nie potrafiłem jednak sensownie użyć tej relacji do zdefiniowania funkcji, problem jaki miałem to posługiwanie się wynikiem funkcji w tej relacji. Próba wyrażenia dziedziny relacji `EvalDom` za pomocą typu egzystencjalnego zawiodła:

```
Definition EvalDom tm env := exists d, EvalGraph tm env d.

Fixpoint eval tm denv  (H : EvalDom tm denv) : D :=
...
.
```

Ponieważ typ egzystencjalny nie jest tym po którym mogę robić rekursję potrzebną do ewaluacji termów. Miałem jako argument `H` „parę" składającą się z elementu `d` oraz dowodu, że `EvalGraph tm env d`, nie umiałem wykonać rekursji w takim przypadku.

3 Indukcja-rekursja.

Szukając informacji jak reprezentować funkcje częściowe w teorii typów napotkałem się na dokumenty wykorzystujące schemat indukcji-rekursji do definiowania predykatu `Dom`. Dzięki temu schematowi można rozwiązać w sposób czytelny mój problem z definicją predykatu dla ewaluatora i funkcji aplikacji. Moje rozwiązanie w systemie Coq to „przeniesienie" tego schematu.

Indukcja-rekursja to schemat dla teorii typów, dzięki któremu możemy definiować typ indukcyjny oraz funkcję przeplatając je ze sobą. Gdy pierwszy raz wyczytałem o tym schemacie to wydał się bardzo egzotyczny i trudno było mi znaleźć namacalne zastosowanie dla niego. Przykładem jaki podaje się przy omawianiu tych definicji jest możliwość zdefiniowania uniwersum ala Tarski, tzn typu danych `U` którego elementy to kody typów oraz funkcji `T: U -> Set` która tłumaczy te kody na typy. Potrzeba przeplatania tutaj funkcji `T` z typem `U` wynika z kodowania typów zależnych.

Zobaczmy przykład jak by wyglądał w „składni Coqa" kod oznaczający Π-typ (źródło tych definicji to książka „Intuitionistic type theory" [1] - ostatni rozdział gdzie została podana taka definicja uniwersum oraz przykład indukcji-rekursji na stronie Agdy - przy definicji tego uniwersum [2]).

```
Inductive U : Set :=
 ...
 | codePi: forall a:U, (T a -> U) -> U
with Fixpoint T : U -> Set := fun u =>
 match u with
  ...
 | codePi a b -> forall (x : T a), T (b x)
 end
.
```

Kod typu `forall x:A, B x` chce znać kod typu `A` oraz funkcję która z elementu o typie `x : A` zwraca kod typu `B x` - definicja tego kodu musi posługiwać się funkcją `T`.

Trudny przykład i zdaje mi się, że mało mówiący ludziom nie ocierającym się o problemy formalizowania teorii typów wewnątrz teorii typów.

Dopiero w pracy [3] znalazłem namacalne zastosowanie indukcji-rekursji, które rozwiązywało mój problem. Autorzy nadali semantykę denotacyjną, wewnątrz teorii typów, językowi IMP, który z powodu posiadania pętli `While` nie jest totalny.

Łatwo zlokalizować mój problem w tym języku, najlepiej obrazuje go denotacja złożenia instrukcji:

$$[\![c_1; c_2]\!]\ \sigma = [\![c_2]\!]\ ([\![c_1]\!]\ \sigma)$$

Gdyby spróbować wyrazić `EvalDom : Cmd -> State -> Prop` dla złożenia, to mielibyśmy identyczną sytuację jak przy ewaluacji aplikacji termów:

```
Inductive EvalDom : Cmd -> State -> Prop :=
 ...
| evalComp : forall c1 c2 s1,
  EvalDom c1 s1 ->
  EvalDom c2 ?  ->
  EvalDom (Comp c1 c2) s1
.
```

Potrzebujemy powiedzieć, że `c2` wraz ze stanem po ewaluacji `c1` należy do dziedziny ewaluatora. Rozwiązaniem tego problemu to definicja predykatu `Eval` jednocześnie z definicją funkcji `eval`. Wtedy podczas definiowania predykatu można wykorzystywać funkcję, by mówić o wynikach z rekurencyjnych wywołań, a sama funkcja może wykorzystywać predykat by robić rekursję.

Rozwiązanie tego problemu możemy zobaczyć w praktyce dzięki Agdzie, gdzie udało mi się wykorzystać technikę z pracy do zaimplementowania mojego ewaluatora. Poniżej definicja typu `EvalDom`, typu `AppDom`, funkcji `eval`, funkcji `dapp` z użyciem schematu indukcja-rekursja.

```
mutual

  data EvalDom : Tm -> DEnv -> Set where

       evalDom_var : (n : N) -> (env : DEnv) ->
                     EvalDom (tvar n) env

       evalDom_abs : (t : Tm) -> (env : DEnv) ->
                     EvalDom (tabs t) env

       evalDom_app : (t0 t1 : Tm) -> (env : DEnv) ->
                     (H0 : EvalDom t0 env) ->
                     (H1 : EvalDom t1 env) ->
                     AppDom (eval t0 env H0) (eval t0 env H1) ->
                     EvalDom (tapp t0 t1) env
      ...
```

Problem zostaje rozwiązany bezpośrednio, w miejscu gdzie potrzebowaliśmy znać wynik rekurencyjnego wywołania wstawiamy po prostu to wywołanie funkcji.

```
      ...

  data AppDom : D -> D -> Set where

       appDom_clo : (tclo : Tm) -> (env : DEnv) -> (dx : D) ->
                    EvalDom tclo (Dext env dx) ->
                    AppDom (dclo tclo env) dx


  eval : (t : Tm) -> (env : DEnv) -> EvalDom t env -> D
  eval (tvar n) (de f)  HDom = f n
  eval (tabs t) (env)   HDom = dclo t env
  eval _ _ (evalDom_app t0 t1 env H0 H1 H2) = dy
   where
     df = eval t0 env H0
     dx = eval t1 env H1
     dy = dapp df dx H2
```

Definicja ewaluacji jest przejrzysta, rekursja jest po `EvalDom`. Wszelkie rekurencyjne wywołania również wyglądają przejrzyście.

```
  dapp : (df dx : D) -> (AppDom df dx) -> D
  dapp (appDom_clo tclo env dx H)  = eval tclo (Dext env dx) H
```

Nie udało mi się w Agdzie poruszyć tematu ekstrakcji, istotne przy powyższej definicji byłoby wymazywanie argumentów typu `AppDom` oraz `EvalDom`. Wymazywanie można uzyskać używając *irrelevant-arguments*, jednak nie udało mi się zaprogramować rekursji po argumencie oznaczonym do wymazania, tzn termination-checker nie widział że coś maleje.

Być może ekstrakcja definicji przy użyciu powyższego schematu nie jest prosta. Gdyby włączyć niejawne podawanie argumentów (oznaczane wąsatymi klamrami w Agdzie) to uzyskalibyśmy poniższy kod a dzięki niemu pewną obserwację:

```
eval : {t : Tm} -> {env : DEnv} -> EvalDom t env -> D
eval {tvar n} {de f}  HDom = f n
eval {tabs t} {env}   HDom = dclo t env
eval (evalDom_app t0 t1 env H0 H1 H2) = dapp H2

dapp : {df dx : D} -> (AppDom df dx) -> D
dapp (appDom_clo tclo env dx H)  = eval H
```

Aplikacja `dapp` oraz `eval` jest tylko do dowodów, ponieważ z nich można odczytać resztę argumentów. Zastanawiający jest przypadek aplikacji termów, aplikujemy bezpośrednio `dapp` do dowodu że można uruchomić funkcję dla wyników ewaluacji dla `t0` oraz `t1`, ale kiedy je policzyliśmy?

Nie mam teraz w definicji funkcji żadnego „przepisu" na liczenie tych podtermów, rekurencyjne wywołania są zakodowane w dowodzie `H2`. Trudno mi sobie wyobrazić co to właściwie by oznaczało dla ekstrakcji. Kiedy i gdzie by te wywołania z dowodu były wstawione w kod?

4 Coq

Autorzy [3] rozważali ekstrakcję ewaluatora w Coqu, jednak ich rozwiązanie jest złe dla moich celów. Zaproponowane rozwiązanie, mówiąc w skrócie i pomijając szczegóły, polega na tym by widzieć ewaluator jako punkt stały pewnego operatora F. W praktyce natomiast posługiwać się skończonymi aproksymacjami tego punktu stałego. W tym celu predykat dziedziny ewaluatora to „istnieje taka liczba k, że F^k potrafi policzyć ten argument".

Problem z tą definicją jest taki, że ewaluator chce wyłuskać ten k z dowodu, a to oznacza że ten typ egzystencjalny nie może być w `Prop` (`{ k | ... }` zamiast `exists k, ....`) i nie zostanie wymazany przy ekstrakcji.

Cel jaki sobie postawiłem to eliminacja tego kompromisu i wyekstraktowanie ewaluatora który przypominałby kod bezpośrednio napisany przez człowieka. By go zrealizować musiałem posłużyć się nie jedną, lecz dwoma pomocniczymi relacjami. Pierwsza to wykres funkcji, jego definicja nie ociera się o żadne problemy:

```
Inductive EvalGraph: Tm -> DEnv -> D -> Prop :=
| evgVar : forall i env,
  EvalGraph (TmVar i) env (Dlookup env i)

| evgAbs : forall t env,
  EvalGraph (TmAbs t) env (DClo t env)

| evg0 : forall env,
  EvalGraph Tm0 env (DNat 0)

| evgS : forall env,
  EvalGraph TmS env DS

| evgApp : forall tf df tx dx dr env,
  EvalGraph tf env df ->
  EvalGraph tx env dx ->
  AppGraph df dx dr ->
  EvalGraph (TmApp tf tx) env dr

with AppGraph: D -> D -> D -> Prop :=
| apgClo : forall tm denv dx dr,
  EvalGraph tm (Dext denv dx) dr ->
  AppGraph (DClo tm denv) dx dr

| apgS : forall n,
```

```
  AppGraph (DS) (DNat n) (DNat (S n))
.
```

Drugi predykat „argument należy do dziedziny" służy by móc zrobić rekursję (oraz rzecz jasna, zamienić funkcję częściową `eval: D -> D` na totalną `eval: forall d, EvalDom d -> D`). Czyli jego funkcja jest identyczna jak przy logarytmie i tyczy się tych samych ograniczeń (dodatkowe funkcje by nie robic zakazanego dopasowania wzorca).

```
Inductive EvalDom: Tm -> DEnv -> Prop :=
| evgVarD : forall i env,
  EvalDom (TmVar i) env

| evgAbsD : forall t env,
  EvalDom (TmAbs t) env

| evgAppD : forall tf tx env,
  EvalDom tf env ->
  EvalDom tx env ->
  (forall df dx,
    EvalGraph tf env df ->
    EvalGraph tx env dx ->
    AppDom df dx) ->
  EvalDom (TmApp tf tx) env

| evg0D : forall env,
  EvalDom Tm0 env

| evgSD : forall env,
  EvalDom TmS env

with AppDom: D -> D -> Prop :=
| apgCloD : forall tm denv dx,
  EvalDom tm (Dext denv dx) ->
  AppDom (DClo tm denv) dx

| appSD : forall n,
  AppDom DS (DNat n)
```

Zanim omówię trick jak zdefiniować funkcję, chciałbym zaznaczyć jedną rzecz która dla mnie nie jest oczywista a jest istotna by zrobić rekursję. Weźmy wartość tego typu `v : EvalDom (TmApp tf tx) env`, która jest konstruktorem dla aplikacji: `v = evgAppD Htf Htx f`. Istotne dla mnie jest, czy gdyby robić rekursję to czy wyniki funkcji f będą uznawane jako strukturalnie mniejsze od v ? Można zrobić eksperyment lub szybciej, wygenerować zasadę indukcji wzajemnej dla tych predykatów i popatrzeć w jej kod/typ - są uznawane za mniejsze.

```
Inductive EvalDom: Tm -> DEnv -> Prop :=
...
| evgAppD : forall tf tx env,
  EvalDom tf env ->
  EvalDom tx env ->
  (forall df dx,
    EvalGraph tf env df ->
    EvalGraph tx env dx ->
    AppDom df dx) ->
  EvalDom (TmApp tf tx) env
...
```

Problem aplikacji zostaje rozwiązany dzięki użyciu wykresu funkcji, w miejscu gdzie chcielibyśmy zrobić wywołanie, oraz dzięki przytoczonej powyżej obserwacji. Z konstruktora `evgAppD` funkcja będzie mogła wyłuskać „pozwolenie" na rekurencyjne wywołania dla podtermów. Gdyby miała dowód, że jej wyniki mają związek z wykresem `EvalGraph` to mogłaby uzyskać „pozwolenie" na uruchomienie aplikacji w dziedzinie.

Uzyskanie związku wyników funkcji ze zdefiniowanym wcześniej wykresem jest proste, zamiast zwracać wartości typu `D` zwracam wartości typu `{ d : D | EvalGraph tm denv d }`. Poniżej kod całej funkcji, duże ułatwienie ze strony Coqa to rozszerzenie `Program`, wystarczyło że zwracałem `d`, a dowód że ta wartość jest w wykresie funkcji był generowany automatycznie.

```
Program Fixpoint eval (tm:Tm) (denv:DEnv) (H:EvalDom tm denv) {struct H}:
  { d:D | EvalGraph tm denv d } :=
  match tm as T
    return (tm = T -> { d:D | EvalGraph T denv d })
    with

    | TmVar i => fun _  =>
      Dlookup denv i

    | Tm0 => fun _ =>
      DNat 0

    | TmS => fun _ =>
      DS

    | TmAbs tm0 => fun  _ =>
      DClo tm0 denv

    | TmApp tm0 tm1 => fun H'  =>
      let (df,Hdf) := eval (EvalDom_app_invF H H') in
      let (dx,Hdx) := eval (EvalDom_app_invX H H') in
      let (dy,Hdy) := app  (EvalDom_app_invA H H' Hdf Hdx) in
      dy

  end (eq_refl tm)

with app (df dx:D) (H:AppDom df dx) {struct H} :
  { d:D | AppGraph df dx d } :=
  match df as DF
    return df = DF -> { d:D | AppGraph DF dx d }
    with

    | DClo tm denv => fun H' =>
      let (dy, Hdy) := eval (AppDom_clo_inv H H') in
      let H         := apgClo Hdy in
        exist (fun d => AppGraph (DClo tm denv) dx d)
        dy H

    | DS => fun H' =>
      match dx as DX return dx = DX -> { d:D | AppGraph df DX d } with
        | DNat n => fun H'' =>
          (DNat (S n))

        | DClo tm denv => fun H' =>
          _
```

```
        | DS => fun H' =>
          _
      end (eq_refl dx)

    | DNat n => fun H' =>
      -

  end (eq_refl df)
.
```

Przeanalizujmy przypadek aplikacji:

```
    | TmApp tm0 tm1 => fun H'  =>
      let (df,Hdf) := eval (EvalDom_app_invF H H') in
      let (dx,Hdx) := eval (EvalDom_app_invX H H') in
      let (dy,Hdy) := app  (EvalDom_app_invA H H' Hdf Hdx) in
      dy
```

Argumenty są ustawione na niejawne, więc funkcje są aplikowane jedynie do dowodów, które tutaj są wyłuskiwane za pomocą projekcji na `EvalDom`. Funkcja `EvalDom_app_invF H H'` zwraca dowód, że podterm jest w dziedzinie ewaluacji. Z rekurencyjnego wywołania otrzymujemy `df` oraz dowód że `EvalGraph tm0 env df`. Podobnie dla drugiego podtermu, ostatecznie `EvalDom_app_invA H H'` zwraca ostatni podterm konstruktora `evgAppD` czyli funkcję co zaaplikowana do dowodów że `df` i `dx` są w wykresie funkcji zwraca strukturalnie mniejszy dowód, że są one w dziedzinie funkcji *app* . Co oznacza, że mogę uruchomić funkcję aplikacji i będzie to rekursja strukturalna.

Wyekstraktowany kod do języka Haskell wygląda tak, jakby go napisał człowiek (zakładając ze również zrobiłby defunkcjonalizację) oraz oczywiście pomijając formatowanie.

```
eval :: Tm -> DEnv -> D
eval tm denv =
  case tm of {
   TmVar i -> dlookup denv i;
   TmApp tm0 tm1 -> app (eval tm0 denv) (eval tm1 denv);
   TmAbs tm0 -> DClo tm0 denv;
   Tm0 -> DNat O;
   TmS -> DS}

app :: D -> D -> D
app df dx =
  case df of {
   DClo tm denv -> eval tm (dext denv dx);
   DNat n -> Prelude.error "absurd case";
   DS ->
    case dx of {
     DNat n -> DNat (S n);
     _ -> Prelude.error "absurd case"}}
```

5 Podsumowanie

Zamierzony cel udało mi się osiągnąć, są kwestie nad którymi wydaje mi się, że należy się zastanowić:

Czy skonstruowana funkcja jest używalna do obliczeń wewnątrz teorii typów? Moim zdanie nie, rodzi ona kilka problemów. Pierwszy jest taki, że obliczenia są zdefiniowanie rekursją strukturalną po dowodzie. Ponieważ dowody zwykle kończymy poleceniem `Qed` to obliczenia byłyby bezwzględnie zatrzymane w pierwszym wywołaniu. Nie można Coqa zmusić by takie funkcje rozwijał, taktyki `unfold` ani `compute` nie mogą z tym nic zrobić.

Myślę, że nawet gdyby przyjąć kończenie wszelkich dowodów za pomocą `Defined` to musielibyśmy się pilnować by nie użyć żadnych twierdzeń z biblioteki standardowej - bo tam są kończone w standardowy sposób. Ta strategia wydaje się być niepraktyczna.

Drugi problem to moim zdaniem małe utrudnienie przy przepisywaniu termów. Przy mojej pierwszej próbie formalizacji NBE zamieniałem pewne relacje na funkcje wierząc że to zwiększa przejrzystość i ułatwia mi zadanie. Wierzyłem że jeżeli będę pisał

$$\forall x,\ ...(f\ x)...$$

zamiast

$$\forall x,\ \exists y, W_f\ x\ y \wedge ...(y)...$$

to zyskam prostotę. Niestety to jest funkcja częściowa, więc musiałem zawsze mieć dowód na temat argumentu.

$$\forall x, \exists h : Dom\ x,\ ...(f\ x\ h)...$$

Robiąc dowody, gdy chciałem przepisać jakiś argument x na inny x' napotykałem problem z typami zależnymi. Mianowicie założenia „argument jest w dziedzinie" dotyczyły x, a nie x', więc przepisywania wymagały abym co chwilę uruchamiał taktykę `generalize dependent` i dopiero przepisywał. Gdy używałem automatyzacji za pomocą `autorewrite` to musiałem również pisać taktyki co generalizują wszelkie założenia mogące utrudnić przepisywanie. W efekcie nie panowałem momentami co się dokładnie dzieje w pewnych fragmentach niektórych dowodów.

Czy można coś wnioskować na temat stworzonej funkcji? Myślę że napotykało to by problemy z poprzedniego paragrafu, mamy jednak wykres funkcji, obecnie myślę że można dowodzić rzeczy o formalizowanej funkcji wygodnie posługując się relacją.

Obecnie przy formalizacji NBE wyrażam takie własności jak „ewaluacja dobrze otypowanego termu kończy się".

```
Theorem Terminating_on_WtTm:
   forall Gamma tm tp denv,
     Gamma |-- tm : tp ->
     EvalCxt Gamma denv ->
     exists dtm, EvalTm tm denv dtp
.
```

Poza wyrażaniem tego typu twierdzeń potrafię obecnie także przygotowywać certyfikowane funkcje do ekstrakcji:

```
Definition wtTm_eval Gamma t A (H: Gamma |-- t : A)
  : { dt | exists denv, EvalCxt Gamma denv /\ EvalTm t denv dt } :=
  let HValTm    := Valid_for_WtTm H in
  let HValTp    := ValTp_from_Tm HValTm in
  let HValCxt   := ValCxt_from_Tp HValTp in
  let HTermCxt  := Terminating_on_ValCxt1 HValCxt in
  let HDomCxt   := Exists_apply EvalCxt EvalCxt_Dom Gamma
                   HTermCxt EvalCxt_Dom_corr in
  let (de,Hde)  := evalCxt HDomCxt in (*!!! <-- TUTAJ !!!*)
  let HTermTp   := Terminating_on_WtTm H Hde in
  let HDomTp    := Exists_apply (EvalTm t) (EvalTm_Dom t)
                   de HTermTp (@EvalTm_Dom_corr t) in
  let (dt,Hdt)  := evalTm HDomTp in (*!!! <-- TUTAJ !!!*)
  let Hprf      := conj Hde Hdt in
  exist _ dt  (ex_intro _ de Hprf)
.
```

Funkcja dostaje term, który jest dobrze otypowany i zwraca jakiś `dt : D`, dzięki grafowi funkcji wygodnie wyraziłem że jest to element będący ewaluacją właśnie tego termu w beztypowej dziedzinie

- czyli to co chcemy. Ponieważ jest to funkcja w teorii typów to bez wątpienia jest określona, tzn ten ekstraktowany algorytm, dla wszystkich dobrze otypowanych termów.

Można zwrócić uwagę także na kod tej funkcji, cały algorytm to uruchomienie beztypowej ewaluacji, kod wygląda jak assembler ponieważ wnioskuję, że beztypowa ewaluacja jest określona. Estetykę można poprawić przenosząc to do jednego twierdzenia lub używając taktyki `refine`. Istotne wydaje mi się, że nie programowałem ewaluacji dla otypowanych termów od nowa, tylko mogłem posłużyć się „wymęczoną" funkcją beztypowego ewaluatora do ekstrakcji.

Jakie jest porównanie ze schematem indukcja-rekursja? Widzę jeden plus na korzyść użycia dwóch relacji niż indukcji-rekursji. Chodzi o problem z wymazywaniem tych predykatów i zakodowanych wywołaniach rekurencyjnych w dowodzie. Uzyskany kod odpowiada temu jak zaprogramowałem funkcję w Coqu, a więc gdybym zdecydował się użyć CPSu by wpływać na kolejność obliczeń to mogę to zrobić - o ile poradzę sobie z robieniem dowodów dla obligacji. Mam wpływ na to jak wygląda i jak działa ekstraktowany kod. Wierzę, że to jak funkcja liczy te argumenty nie ma wpływu na to jak wygląda `EvalGraph` oraz `EvalDom`. W przypadku indukcji-rekursji, zakładając że można robić ekstrakcję z wymazywaniem, musiałbym dla każdego ewaluatora robić od nowa predykaty - bo w nich są zakodowane wywołania konkretnej funkcji. (Oraz problem że wywołania rekurencyjne są w dowodzie, więc mając konkretny mechanizm ekstrakcji, który by działał, bylibyśmy skazani na to, gdzie on te wywołania z dowodu przeniesienie).

Czy można prościej uzyskać ekstrakcję funkcji z wykresu? Szukając informacji o funkcjach częściowych znalazłem także prace nad rozszerzeniem mechanizmu ekstrakcji w Coqu do bezpośredniej obsługi wykresów funkcji [5]. Mechanizm prawdopodobnie zostanie włączony do kolejnego wydania systemu Coq. Jestem w posadaniu developerskiej wersji, lecz jeszcze nie porównywałem jak wygląda ekstraktowany kod. Zamierzam dopiero to zrobić gdy będę ekstraktował coś większego niż przykładowy ewaluator, dopiero wtedy spodziewam się możliwych różnic.

Natomiast jeżeli chodzi o wykorzystanie to skoro ekstrakcja będzie bezpośrednio z wykresu do języka, to nie będzie możliwe przygotowanie takiej certyfikowanej funkcji jak ta z innego paragrafu.

Inne pytanie jakie można zadać, to to czy można użyć innej machinerii Coqa do zdefiniowania tych funkcji. Oprócz rekursji strukturalnej możliwe jest podanie innych miar dla poleceń `Program Fixpoint` czy `Function`. Wersja `{measure ...}` wymaga abym podał miarę z danego typu w funkcje naturalne, ponieważ chcę przy ekstrakcji wymazać argument po którym robię rekursję to musi on być w `Prop` a ponieważ na nim obliczenia są zabrobione to takiej miary nie można zrobić. Wersja `{wf ...}` wymaga abym zamiast `EvalDom` posłużył się dobrze ufundowaną relacją. Nawet gdyby udało mi się zrobić dobrze ufundowaną relację, lub udowodniłbym że obecna taka jest, to nie wiem jak użyć `wf` dla przeplatających się funkcji. Funkcja ewaluacji i aplikacji muszą się przeplatać z powodu defunkcjonalizacji, a przy robieniu różnych eksperymentów Coq dawał komunikat:

```
Cannot use mutual definition with well-founded recursion or measure
```

Literatura

[1] Per Martin-Löf, Intuitionistic type theory, 1980
`http://www.csie.ntu.edu.tw/~b94087/ITT.pdf`

[2] The Agda Wiki
`http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.RulesForTheStandardSetFormers`

[3] Y.Bertot, V. Capretta, K. D. Barman - Type-Theoretic Functional Semantics, 2002
`http://www.cs.ru.nl/~venanzio/publications/Functional_Semantics_TPHOLs_2002.pdf`

[4] A.Abel - Towards Normalization by Evaluation for the Calculus of Constructions, 2009
`http://www2.tcs.ifi.lmu.de/~abel/flops10long.pdf`

[5] D. Delahaye, C. Dubois, J. Étienne, Extracting Purely Functional Contents from Logical Inductive Types, 2007
`http://cedric.cnam.fr/~delahaye/papers/pred-exec%20(TPHOLs'07).pdf`