Wnioskowanie o programach

Niniejsza notatka dotyczy wnioskowania o programach w języku Plait. Przez *wnioskowanie o programach* rozumiemy traktowanie programów jako obiektów matematycznych i matematyczne dowodzenie przeróżnych własności tychże obiektów. Podobnie jak na kursie Logiki dla Informatyków do zapisywania własności i ich dowodów będziemy używać języka naturalnego, ale z pełną świadomością tego, że nasze twierdzenia i rozumowania można przepisać na bardziej formalny system (np. logika pierwszego rzędu).

1 Równoważność programów

Programy, a bardziej precyzyjniej wyrażenia i wartości[1] będziemy traktować czysto syntaktycznie, czyli jako drzewa dla których możemy definiować przeróżne relacje. Oczywiście można zdefiniować relacje mówiące np. o tym, że dana lista liczb jest posortowana, albo drzewo binarne jest drzewem przeszukiwań. Ale zamiast definiować wielu wyspecjalizowanych relacji, skupimy się na jednej bardzo podstawowej, czyli na *równoważności wyrażeń*. Będziemy zapisywać $e_1 \equiv e_2$ jeśli wyrażenie e_1 jest równoważne wyrażeniu e_2, o czym można myśleć tak, że jeśli w większym programie zamienimy wyrażenie e_1 na e_2 to nie zaobserwujemy żadnej różnicy w działaniu programu (może za wyjątkiem czasu wykonania). Na tym przedmiocie nie będziemy formalnie tej relacji definiować, ale za to ją zaksjomatyzujemy[2]. Oczekujemy, że relacja $\equiv$ porównuje wyrażenia tego samego typu i ma następujące własności.

- Jest relacją równoważności (jest zwrotna, symetryczna i przechodnia).
- Jest kongruencją, czyli jest zachowywana przez wszystkie konstrukcje

[1] Przez *wartości* rozumiemy wartości dla podstawieniowego modelu obliczeń, czyli wyrażenia których wykonanie nie powoduje błędu i nie dają się dalej uprościć.

[2] Istnienie relacji spełniającej nasze aksjomaty daleko wykracza poza ten wykład.

języka. Na przykład, jeśli $e_1 \equiv e'_1$, $e_2 \equiv e'_2$ oraz $e_3 \equiv e'_3$ to

$$\texttt{(if } e_1\ e_2\ e_3\texttt{)} \equiv \texttt{(if } e'_1\ e'_2\ e'_3\texttt{)}.$$

Własność kongruencji pozwala przepisywać równe na równe w większych wyrażeniach.

- Jest zgodna z modelem obliczeń, tzn. jeśli wyrażenie e_1 oblicza się do wyrażenia e_2 w podstawieniowym modelu obliczeń, to $e_1 \equiv e_2$.
- Rozróżnia rzeczy w oczywisty sposób różne, np. `#t` $\not\equiv$ `#f` albo `13` $\not\equiv$ `42`. Co ciekawe, można pokazać, że jeśli relacja jest kongruencją zgodną z modelem obliczeń oraz rozważany język jest dostatecznie ekspresywny (Plait taki jest), to z jednej takiej różności wynikają inne.

Do zestawu wymaganych aksjomatów można jeszcze dorzucić zasadę *ekstensjonalności funkcji*: jeśli dla dowolnych wartości $v_1 : \tau_1, \ldots, v_n : \tau_n$ zachodzi $\texttt{(}e_1\ v_1\ \texttt{...}\ v_n\texttt{)} \equiv \texttt{(}e_2\ v_1\ \texttt{...}\ v_n\texttt{)}$, (gdzie e_1 i e_2 mają odpowiedni typ funkcyjny), to $e_1 \equiv e_2$. Jednak w naszych zastosowaniach zasada ekstensjonalności nie będzie nigdy potrzebna.

Ciekawą cechą relacji równoważności jest to, że nie potrzebujemy teraz definiować innych relacji na programach. Na przykład jeśli chcemy wyrazić, że drzewo `t` jest binarnym drzewem przeszukiwań, to wystarczy powiedzieć `(bst? t)` $\equiv$ `#t`, gdzie `bst?` jest zwykłą funkcją zdefiniowaną w Plaicie.

2 Dowodzenie prostych własności programów

Przypomnijmy definicję bibliotecznej funkcji append:

```
(define (append xs ys)
  (if (empty? xs)
      ys
      (cons (first xs) (append (rest xs) ys))))
```

Pokażmy, że `empty` jest lewostronnym elementem neutralnym funkcji append.

Lemat 1. *Dla dowolnej listy xs mamy* $\texttt{(append empty } xs\texttt{)} \equiv xs$.

Zanim udowodnimy ten lemat, zwróćmy uwagę na dwie rzeczy. Po pierwsze pracujemy w dobrze typowanym języku, więc lemat ma sens tylko wtedy gdy lista xs jest typu `(Listof` τ`)` dla pewnego typu τ. Przyjmijmy więc konwencję, że niejawnie przyjmujemy założenia, potrzebne do tego by pisane przez nas wyrażenia były dobrze typowane. Po drugie Plait jest językiem gorliwym, więc by policzyć `(append empty` xs`)` najpierw trzeba obliczyć xs do wartości, co może prowadzić do pewnych niezręczności w dowodach. By tego uniknąć, przyjmijmy konwencję, że o ile nie zaznaczymy inaczej, to wszystkie zmienne w naszych sformułowaniach oznaczają pewne wartości. Przyjmując powyższe konwencje, sformułowanie lematu należy rozumieć następująco:

Dla dowolnego typu τ oraz wartości xs typu `(Listof` τ`)` *zachodzi* `(append empty` xs`)` $\equiv$ xs.

Dowód. Relacja $\equiv$ jest relacją równoważności, więc dowód zapiszemy jako ciąg wyrażeń, z których każde kolejne dwa są równoważne.

`(append empty` xs`)` $\equiv_{\text{(wykonując krok obliczeń)}}$

`(`**`if`**` (empty? empty)`xs` (...))` $\equiv_{\text{(wykonując krok obliczeń)}}$

`(`**`if`**` #t `xs` (...))` $\equiv_{\text{(wykonując krok obliczeń)}}$

xs □

Powyższy dowód nie jest specjalnie ciekawy i polega wyłącznie na przepisywaniu programów korzystając z tego, że relacja $\equiv$ jest zgodna z modelem obliczeń. Od tego momentu nie będziemy rozpisywać wszystkich przekształceń w tak szczegółowy sposób, tylko wolimy powiedzieć, że ta równość wynika wprost z definicji funkcji `append`.

3 Indukcja strukturalna dla list

Pokażmy, że `empty` jest prawostronnym elementem neutralnym funkcji `append`. Niestety próbując obliczyć wyrażenie `(append` xs `empty)` będziemy musieli obliczyć `(empty?` xs`)`, którego wartość zależy od tego, czym jest lista xs. Można by rozważyć dwa przypadki (jakie?), ale jeśli lista xs jest postaci `(cons` y ys`)`, to zawołamy funkcję `append` rekurencyjnie i będziemy musieli policzyć `(empty?` ys`)`. Oczywiście można by teraz rozważyć kolejne przypadki jak wygląda lista ys, ale kontynuując w ten sposób okaże się, że musimy rozważyć nieskończenie wiele przypadków. Potrzebujemy więc metody by móc wnioskować o nieskończenie wielu listach w skończony sposób.

Widzieliśmy już technikę wnioskowania o wszystkich liczbach naturalnych. Była to zasada indukcji, która dla liczb naturalnych wygląda następująco.

Zasada indukcji (dla liczb naturalnych). *Niech P będzie własnością liczb naturalnych, taką, że:*

(i) $P(0)$,

(ii) dla dowolnego $n \in \mathbb{N}$ *jeśli* $P(n)$ *to również* $P(n+1)$.

Wówczas dla każdego $n \in \mathbb{N}$ *zachodzi* $P(n)$.

Zauważmy, że zasada indukcji dla liczb naturalnych jest konsekwencją tego, że każda liczba naturalna to albo 0, albo $n+1$ dla jakiejś liczby naturalnej n. A dokładniej tego, że zbiór liczb naturalnych to najmniejszy zbiór do którego należy zero i jest zamknięty na dodawanie jedynki. Listy mają podobną strukturę: każda lista to albo lista pusta (`empty`), albo element dołączony do innej listy za pomocą konstruktora `cons`. Przez analogię możemy sformułować zasadę indukcji dla list.

Zasada indukcji (dla list). *Niech P będzie własnością list, taką, że:*

(i) $P(\texttt{empty})$,

(ii) dla dowolnego elementu x *oraz listy* xs*, jeśli* $P(xs)$ *to* $P(\texttt{(cons}\ x\ xs\texttt{)})$.

Wówczas dla dowolnej listy xs *zachodzi* $P(xs)$.

Teraz mamy wszystkie potrzebne narzędzia, by pokazać, że `empty` jest prawostronnym elementem neutralnym funkcji `append`.

Lemat 2. *Dla dowolnej listy* xs *mamy* $\texttt{(append}\ xs\ \texttt{empty)} \equiv xs$.

Dowód. Przez indukcję względem listy xs. To znaczy, stosujemy zasadę indukcji dla list podstawiając

$$P(xs) \quad := \quad \texttt{(append}\ xs\ \texttt{empty)} \equiv xs.$$

Trzeba pokazać, że P spełnia założenia zasady indukcji.

(i) $P($`empty`$)$ zachodzi na mocy Lematu 1.

(ii) Weźmy dowolne x i xs takie, że $P(xs)$, czyli (append xs empty) $\equiv$ xs. Założenie $P(xs)$ będziemy nazywać *założeniem indukcyjnym* (w skrócie *z.i.*). Pokażmy, że (append (cons x xs) empty) $\equiv$ (cons x xs).

(append (cons x xs) empty) $\equiv_{\text{(z def. append)}}$
(cons x (append xs empty)) $\equiv_{\text{(z z.i. oraz własności kongruencji)}}$
(cons x xs)

Zatem na mocy zasady indukcji $P(xs)$ zachodzi dla dowolnej listy xs, co należało pokazać. □

Spróbujmy udowodnić odrobinę bardziej skomplikowany lemat o łączności funkcji append.

Lemat 3. *Dla dowolnych list xs, ys oraz zs zachodzi*
(append (append xs ys) zs) $\equiv$ (append xs (append ys zs)).

Dowód oczywiście będzie przebiegał przez indukcję. Teraz jednak mamy aż trzy listy: xs, ys oraz zs. Względem której należy przeprowadzić indukcję? Jeśli wybierzemy listę zs, to będziemy musieli pokazać między innymi, że (append (append xs ys) empty) $\equiv$ (append xs (append ys empty)). Niewiele nam to pomoże, bo funkcja append w ogóle nie patrzy na swój ostatni argument i podstawienie za zs czegokolwiek nie pozwoli nam w żaden sposób bardziej uprościć rozważanych wyrażeń.

Dowód Lematu 2 przebiegł bezboleśnie, bo nałożyły się na siebie dwie rzeczy: po pierwsze funkcja append woła się rekurencyjnie tylko na liście, która jest ogonem pierwszego argumentu (mówimy, że funkcja append jest strukturalnie rekurencyjna względem swojego pierwszego argumentu). A po drugie, indukcję przeprowadziliśmy względem właśnie pierwszego argumentu funkcji append. Spowodowało to, że zarówno rekursja, jak i indukcja podążały za strukturą tej samej listy, więc wszystko do siebie pasowało. Jest to też jeden z powodów dla których lubimy rekursję strukturalną: indukcja jest świetnym narzędziem do wnioskowania o funkcjach strukturalnie rekurencyjnych. W końcu indukcja to nic innego jak rekursja strukturalna na poziomie dowodów matematycznych. Zatem dowód należy przeprowadzić względem tej listy, względem której podąża rekursja.

Dowód. Przez indukcję względem listy xs.

(i) Pokażmy, że (append (append empty ys) zs) $\equiv$ (append empty (append ys zs)). Będziemy upraszczać zarówno lewą stronę (L) jak i prawą (P).

L $\equiv$ (append (append empty ys) zs) $\equiv_{(\text{z lem. 1})}$ (append ys zs)

Można ulec złudzeniu, że prawą stronę również można uprościć z Lematu 1. Nie jest to prawda, bo przyjęliśmy konwencję, że wszystkie zmienne reprezentują wartości, a wyrażenie (append ys zs) wartością nie jest. Co więcej, Plait jest gorliwym językiem więc nie możemy rozwinąć ciała zewnętrznej funkcji append dopóki argumenty nie są policzone.[3] Sytuacja nie jest zupełnie beznadziejna, bo funkcja append zawsze się zatrzymuje, więc istnieje wartość v taka, że (append ys zs) $\equiv$ v $(*)$, co pokażemy na ćwiczeniach. A zatem:

P $\equiv$ (append empty (append ys zs)) $\equiv_{(*)}$
(append empty v) $\equiv_{(\text{z lem. 1})}$ v $\equiv_{(*)}$ (append ys zs) $\equiv$ L.

(ii) Załóżmy, że (append (append xs ys) zs) $\equiv$ (append xs (append ys zs)) i pokażmy, że (append (append (cons x xs) ys) zs) $\equiv$ (append (cons x xs) (append ys zs)). Podobnie jak w poprzednim przypadku, będzie trzeba uprościć wywołanie funkcji append z argumentem który nie jest wartością, ale oblicza się do pewnej wartości. Dla czytelności dowodu pominiemy kroki pośrednie, tylko powiemy, że równość wynika z własności terminacji dla funkcji append.

L $\equiv$ (append (append (cons x xs) ys) zs) $\equiv_{(\text{z def. append})}$
(append (cons x (append xs ys)) zs) $\equiv_{(\text{z def. i term. append})}$
(cons x (append (append xs ys) zs)) $\equiv_{(\text{z z.i.})}$
(cons x (append xs (append ys zs))) $\equiv_{(\text{z def. i term. append})}$
(append (cons x xs) (append ys zs)) $\equiv$ P

□

[3] Cała ta dyskusja wynika z technicznych niezręczności, jakie nie występują w językach leniwych. Jeśli ktoś te problemy zignoruje, to uznamy to za niewielki grzech.

Spróbujmy teraz pokazać, że poniższe dwie definicje funkcji obracającej listę są równoważne.

```
(define (rev1 xs)
  (if (empty? xs)
      empty
      (append (rev1 (rest xs)) (cons (first xs) empty))))

(define (revapp xs ys)
  (if (empty? xs)
      ys
      (revapp (rest xs) (cons (first xs) ys))))

(define (rev2 xs)
  (revapp xs empty))
```

Lemat 4. *Dla dowolnej listy xs mamy* `(rev1` xs`)` $\equiv$ `(rev2` xs`)`.

Pierwsza próba dowodu. Oczywiście przez indukcję względem listy xs.

(i) Przypadek dla listy pustej jest prosty: oba wyrażenia obliczają się do listy pustej.

(ii) Załóżmy, że `(rev1` xs`)` $\equiv$ `(rev2` xs`)` i pokażmy, że `(rev1 (cons` x xs`))` $\equiv$ `(rev2 (cons` x xs`))`. Zacznijmy upraszczać prawe wyrażenie.

P $\equiv$ `(rev2 (cons` x xs`))` $\equiv_{\text{(z def. rev2)}}$
`(revapp (cons` x xs`) empty)` $\equiv_{\text{(z def. revapp)}}$
`(revapp` xs `(cons` x `empty))`

Otrzymanego wyrażenia nie ma jak dalej uprościć. Co więcej, założenie indukcyjne jest tutaj bezużyteczne, bo mówi o funkcji `rev2`, a my mamy do czynienia z funkcją `revapp`. Tego dowodu nie da się dokończyć w elegancki sposób i dlatego go porzucimy. Oczywiście można go dokończyć wprowadzając odpowiednie lematy pomocnicze, które następnie udowodnimy przez indukcję, ale trzeba się przy tym nameczyć. Niepotrzebnie — zrobimy to prościej.

Nasza próba dowodu się nie udała z bardzo prostej przyczyny — indukcja jest świetnym narzędziem do dowodzenie własności funkcji strukturalnie rekurencyjnych, a funkcja `rev2` rekurencyjna nie jest. Całe szczęście funkcja `rev2` jest zdefiniowana za pomocą funkcji `revapp`, która już jest strukturalnie rekurencyjna. Trzeba więc znaleźć własność, która powiąże ze sobą funkcje `rev1` i `revapp`, którą będziemy w stanie udowodnić przez indukcję. Naturalnym kandydatem mogłaby być równość

$$\texttt{(rev1 } xs\texttt{)} \equiv \texttt{(revapp } xs \texttt{ empty)},$$

ale to nam nic nie da. W końcu prawa strona tej równości to nic innego jak rozwinięcie definicji `rev2`. Zauważmy, że drugi argument funkcji `revapp` zmienia się przy każdym wywołaniu rekurencyjnym, więc poszukujemy własności w której drugi argument może być dowolną listą, niekoniecznie pustą. Równość

$$\texttt{(rev1 } xs\texttt{)} \equiv \texttt{(revapp } xs \; ys\texttt{)}$$

też nie jest dobrym pomysłem, bo w ogólności nie zachodzi. Lista ys powinna występować również po lewej stronie równania. Tu z pomocą przychodzi intuicja dotycząca funkcji `revapp`: funkcja `revapp` obraca jedną listę i dołącza do drugiej. Intuicję tę można łatwo przekuć w sformułowanie następującego lematu.

Lemat 5. *Dla dowolnych list xs oraz ys zachodzi*
$\texttt{(append (rev1 } xs\texttt{) } ys\texttt{)} \equiv \texttt{(revapp } xs \; ys\texttt{)}$.

…