Dowodzenie własności programów

Niniejsza notatka jest częścią materiałów pomocniczych do wykładu z Metod Programowania.[1] Zajmiemy się w niej zasadami nieformalnego systemu wnioskowania równościowego o programach w Rackecie, wprowadzonego na wykładzie.

1 Podstawy

Pracujemy w niesformalizowanym świecie matematycznym, w którym chcemy dowodzić *własności* programów nie precyzując czym te własności są — poza wprowadzeniem binarnej relacji *równoważności* wyrażeń Racketowych, którą zapisujemy $\equiv$.

Na dobry początek przyjmujemy że $\equiv$ jest relacją równoważności, tj., że jest zwrotna, symetryczna i przechodnia. Przyjmujemy też, że jest kongruencją ze względu na wszystkie konstrukcje języka, czyli że wyrażenia zbudowane z równoważnych elementów są równoważne. Przykładowa reguła dla formy specjalnej `if` mówiłaby że

$$(\texttt{if}\ e_1\ e_2\ e_3) \equiv (\texttt{if}\ e_1'\ e_2'\ e_3') \quad \text{gdy } e_1 \equiv e_1',\ e_2 \equiv e_2' \text{ i } e_3 \equiv e_3'.$$

Moglibyśmy sformułować analogiczne reguły dla pozostałych wyrażeń języka, w szczególności dla kombinacji i wszystkich form specjalnych, jednak pozostawimy je w domyśle. Powyższych reguł zazwyczaj nie stosujemy jawnie: to czego naprawdę chcemy to możliwość "przepisywania równości w kontekście", co robiliśmy w wielu dotychczasowych przykładach. W ten sposób reguły dotyczące kongruencji stanowią dla nas dobrą bazę, ale same nie wnoszą żadnych interesujących zasad.

1.1 Równoważność i obliczanie

Bardziej interesującą klasą reguł są reguły związane z przyjętym modelem obliczania wartości. Pokażemy trzy z nich, dotyczące reguł obliczania dla

[1] Notatka będzie powstawała w ramach tegorocznego lockdownu i powinna być traktowana jako *work in progress*. To znaczy, w szczególności, że mogą się pojawiać błędy czy nieścisłości: jeśli zauważycie coś podejrzanego, dajcie znać!

wyrażeń warunkowych i procedur:

$$(\texttt{if}\ e_1\ e_2\ e_3) \equiv e_2 \quad \text{gdy } e_1 \equiv \texttt{true}$$
$$(\texttt{if}\ e_1\ e_2\ e_3) \equiv e_3 \quad \text{gdy } e_1 \equiv \texttt{false}$$
$$((\texttt{lambda}\ (x_1 \ldots x_n)\ e)\ e_1 \ldots e_n) \equiv e[e_i/x_i]_{i=1\ldots n}$$

Pierwsze dwie reguły oddają zachowanie wyrażenia warunkowego: jeśli warunek jest równoważny prawdzie, całe wyrażenie jest równoważne e_2; jeśli fałszowi — e_3. Zwróćmy uwagę, że powyższe reguły to *schematy*: e_i reprezentują w nich *dowolne wyrażenia*. Ostatnia reguła nie ma dodatkowych warunków: procedurę zaaplikowaną do odpowiedniej liczby argumentów zawsze możemy zastąpić jej ciałem, w którym *wolne wystąpienia parametrów formalnych* (x_i) zastępujemy odpowiednim argumentem (e_i).[2] Ostatnia reguła, której nie podajemy formalnie, mówi że nazwy którym nadaliśmy wartość przy użyciu `define` są równe wyrażeniom które w ten sposób nazwaliśmy.

Przykład 1. Procedurę identycznościową zdefiniowaliśmy następująco:

```
(define (identity x) x)
```

Pokażemy, że dla dowolnego v mamy

$$(\texttt{identity}\ v) \equiv v.$$

Liczymy następująco:

$$\begin{aligned}(\texttt{identity}\ v) &\equiv && \text{(z definicji identity)}\\ ((\texttt{lambda}\ (\texttt{x})\ \texttt{x})\ v) &\equiv && \text{(z reguły dla procedur)}\\ v && \text{(gdyż x}[v/\texttt{x}]\text{ to po prostu } v)\end{aligned}$$

Zauważmy, że mimo prostoty przykładu pierwsza reguła niejawnie korzystała z własności kongruencji dla kombinacji (przepisaliśmy `identity` do odpowiedniej lambdy w kontekście aplikacji do v). Oczywiście w praktyce nie rozpisywalibyśmy *tych* dwóch kroków osobno: wygodniej jest użyć definicji i reguły dla lambdy jednocześnie. W praktyce to które kroki napisać jawnie, a które przemilczeć musi być dyktowane czytelnością rozumowania.

[2] Zauważmy że powyższe reguły bardziej odpowiadają obliczeniom leniwym: jest to świadomy wybór z naszej strony. Analogiczny system wnioskowania dla obliczeń gorliwych byłby znacznie bardziej skomplikowany, a naszym celem jest nie tyle dowodzenie poprawności programów *per se* co uzyskanie głębszego zrozumienia natury programów, do czego ten system w zupełności wystarcza.

2 Dowodzenie własności dla liczb naturalnych

Ta sekcja zostanie uzupełniona na podstawie przykładów z wykładu w miarę możliwości czasowych.

3 Dowodzenie własności dla struktur danych

Mając za sobą podstawy naszego systemu dowodzenia i przykłady dla wyrażeń działających na liczbach naturalnych możemy zastanowić się jak wnioskować o strukturach danych zbudowanych z par. Będziemy do tego potrzebować nowych reguł wnioskowania, mówiących coś o tym jak zachowuje się konstruktor, predykat i selektory dla par, a także predykat dla listy pustej. Przyjmujemy następujące reguły związane z obliczeniami:

$$\begin{aligned}
(\texttt{pair?}\ e) &\equiv \mathit{true} && \text{gdy } e \equiv (\texttt{cons}\ e_1\ e_2)\\
(\texttt{pair?}\ e) &\equiv \mathit{false} && \text{w przeciwnym przypadku}\\
(\texttt{car}\ (\texttt{cons}\ e_1\ e_2)) &\equiv e_1\\
(\texttt{cdr}\ (\texttt{cons}\ e_1\ e_2)) &\equiv e_2\\
(\texttt{null?}\ e) &\equiv \mathit{true} && \text{gdy } e \equiv \texttt{null}\\
(\texttt{null?}\ e) &\equiv \mathit{false} && \text{w przeciwnym przypadku}
\end{aligned}$$

oraz (bez jawnego wypisywania ich) odpowiednie reguły kongruencji.

3.1 Listy i indukcja strukturalna

Listy, jak przystało na struktury danych w Scheme'ie, skonstruowane są z ułożonych w odpowiedni sposób par (i listy pustej). Konstrukcję listy możemy opisać następującym rekurencyjnym predykatem:

```
(define (list? x)
  (or (null? x)
      (and (pair? x)
           (list? (cdr x)))))
```

Predykat ten mówi że listą jest dowolna struktura która albo jest listą pustą, albo parą której drugi element jest listą — a o pierwszym (reprezentującym *element* listy) niczego konkretnego nie wiemy. Przyjmujemy ten predykat za *definicję* listy jako typu danych.

Podobnie jak w przypadku liczb naturalnych (dla których użyliśmy wbudowanego predykatu `natural?`) chcielibyśmy umieć dowodzić własności programów operujących na listach *ogólnie*: posłuży nam do tego odpowiednia zasada

indukcji, którą (podobnie jak dla liczb naturalnych) przyjmiemy bez dowodu. Brzmi ona następująco:

Zasada indukcji dla list. Dla dowolnej własności P, jeśli

- zachodzi $P(\texttt{null})$ i
- dla dowolnych x, xs, jeśli zachodzi $P(xs)$ to zachodzi $P((\texttt{cons}\ x\ xs))$,

to dla dowolnego xs, jeśli zachodzi $(\texttt{list?}\ xs)$ to zachodzi $P(xs)$.

Zasada wygląda bardzo podobnie do tej dla liczb naturalnych; podobnie jak poprzednio nie precyzujemy też czym może być własność P. Pewną różnicą jest to, że wiemy jak listy są zbudowane — i możemy zauważyć ścisłą odpowiedniość między sformułowaniem zasady indukcji a predykatem `list?`:

- dwa przypadki które musimy rozważyć („podstawa" i „krok" indukcji) odpowiadają dwóm przypadkom alternatywy w definicji predykatu;
- użycie odpowiednio konstruktorów `null` i `cons` w zasadzie indukcji odpowiada wymogom predykatów `null?` i `pair?` w gałęziach alternatywy w definicji predykatu;
- założenie indukcyjne $P(xs)$ odpowiada wymaganiu nałożonemu przez predykat `list?` że drugi element pary sam musi być listą.

Jak zobaczymy dalej, nie jest to odpowiedniość przypadkowa. Na razie jednak spróbujmy wykorzystać indukcję do dowiedzenia kilku prostych własności.

Przykład 2. Niech procedura `length` będzie dana następującą definicją:

```
(define (length xs)
  (if (null? xs)
      0
      (+ 1 (length (cdr xs)))))
```

Dla dowolnego xs takiego że $(\texttt{list?}\ xs)$ pokażemy że $(\texttt{natural?}\ (\texttt{length}\ xs))$.

Przeprowadzimy dowód przez indukcję względem struktury xs, przyjmując za naszą własność

$$P(ys) = (\texttt{natural?}\ (\texttt{length}\ ys)).$$

Wystarczy zatem pokazać że dwa przypadki wymagane przez zasadę indukcji zachodzą.

Najpierw pokażmy że `(natural? (length null))`. W tym celu liczymy:

$$\begin{gathered}(\texttt{natural? } (\texttt{length null})) \equiv \text{z definicji } \texttt{length} \\ (\texttt{natural? } (\texttt{if } (\texttt{null? null}) \texttt{ 0} \ldots)) \equiv \text{z reguł dla } \texttt{if} \text{ i } \texttt{null?} \\ (\texttt{natural? 0})\end{gathered}$$

Na mocy naszej wiedzy o liczbach naturalnych pierwszy przypadek uznajemy za udowodniony.

Rozważmy teraz drugi przypadek: w tym celu ustalmy pewne x i xs, takie że zachodzi $P(xs)$, tj., że $(\texttt{natural? } (\texttt{length } xs))$. Pokażemy że zachodzi $P((\texttt{cons } x\ xs))$. Liczymy:

$$\begin{aligned}&(\texttt{natural? } (\texttt{length } (\texttt{cons } x\ xs))) \equiv \\ &\qquad \text{(z definicji } \texttt{length}\text{)} \\ &(\texttt{natural? } (\texttt{if } (\texttt{null? } (\texttt{cons } x\ xs))\ \texttt{0}\ (+\ \texttt{1}\ (\texttt{length } (\texttt{cdr } (\texttt{cons } x\ xs)))))) \equiv \\ &\qquad \text{(z reguł dla } \texttt{if} \text{ i } \texttt{null?}\text{)} \\ &(\texttt{natural? } (+\ \texttt{1}\ (\texttt{length } (\texttt{cdr } (\texttt{cons } x\ xs))))) \equiv \\ &\qquad \text{(z reguły dla } \texttt{cdr}\text{)} \\ &(\texttt{natural? } (+\ \texttt{1}\ (\texttt{length } xs)))\end{aligned}$$

Z naszej wiedzy o liczbach naturalnych[3] wynika że wystarczy żeby zachodziło $(\texttt{natural? } (\texttt{length } xs))$ — ale dokładnie tę własność założyliśmy jako $P(xs)$, co pozwala nam zakończyć dowód.

Dalsze przykłady pojawią się w miarę możliwości czasowych.

3.2 Indukcja strukturalna dla drzew binarnych

Przyjrzyjmy się teraz drzewom binarnym z etykietami w wierzchołkach wewnętrznych. Zdefiniowaliśmy je następującym predykatem:

```
(define (tree? x)
  (or (leaf? x)
      (and (node? x)
           (tree? (node-left x))
           (tree? (node-right x)))))
```

gdzie `leaf?` jest predykatem rozpoznawającym (puste) liście, `node?` rozpoznaje wierzchołki zewnętrzne, a `node-left` i `node-right` są odpowiednimi selektorami. Przyjmujemy że liść jest reprezentowany zmienną `leaf`, konstruktorem

[3] Konkretnie: z tego że suma liczb naturalnych jest liczbą naturalną.

wierzchołków wewnętrznych jest trzyargumentowa procedura `node`, a trzecim selektorem (wybierającym etykietę wierzchołka wewnętrznego) jest `node-elem`.

Zanim zastanowimy się nad zasadą indukcji odpowiednią dla naszego typu danych, zwróćmy uwagę że drzewa zdefiniowaliśmy w sposób bardziej *abstrakcyjny* niż listy: w istotnym sensie nie jest istotne jaka jest konkretna reprezentacja liści i wierzchołków wewnętrznych tak długo jak spełniają one pewne równości. Pytanie — jakie to równości?

Okazuje się że dobrym przybliżeniem jest nasz zestaw równości dla par i listy pustej: wystarczy go dopasować do naszych konstruktorów, predykatów i selektorów. Mamy więc:

$$\begin{aligned}
(\texttt{node?}\ e) &\equiv \mathit{true} && \text{gdy } e \equiv (\texttt{node}\ x\ l\ r)\\
(\texttt{pair?}\ e) &\equiv \mathit{false} && \text{w przeciwnym przypadku}\\
(\texttt{node}-\texttt{elem}\ (\texttt{node}\ x\ l\ r)) &\equiv x\\
(\texttt{node}-\texttt{left}\ (\texttt{node}\ x\ l\ r)) &\equiv l\\
(\texttt{node}-\texttt{right}\ (\texttt{node}\ x\ l\ r)) &\equiv r\\
(\texttt{leaf?}\ e) &\equiv \mathit{true} && \text{gdy } e \equiv \texttt{leaf}\\
(\texttt{leaf?}\ e) &\equiv \mathit{false} && \text{w przeciwnym przypadku}
\end{aligned}$$

Powinniśmy teraz zauważyć kilka istotnych faktów. Po pierwsze, mamy bardzo wyraźny wzorzec dla „zestawów" konstruktor-predykat-selektory o *dowolnej* ustalonej liczbie argumentów. Jeśli konstruktor ma n argumentów, to mamy n selektorów o odpowiednich regułach opisujących „selekcję z konstruktora", a predykat jednoznacznie rozstrzyga czy jego argument mógł zostać skonstruowany odpowiednim konstruktorem.

Drugą istotną kwestią jest fakt że — w przeciwieństwie do par — naszych równań nie przyjmujemy za aksjomaty. W przeciwieństwie do przypadku par, nasze procedury nie są wbudowane ale *zdefiniowane*: zatem, żeby użyć tych równości w dowodach własności drzew musimy je wcześniej *udowodnić*. Ostatnia kwestia dotyczy samych dowodów: okazuje się że w standardowej implementacji którą rozważaliśmy (gdzie nasze dane są albo symbolami, albo tagowanymi krotkami, a selektory wyciągają z krotki element o odpowiednim indeksie) dowody te są trywialnymi obliczeniami używającymi aksjomatów dla par, które poznaliśmy wcześniej (i aksjomatem o porównywaniu symboli z notatki o cytowaniu). Ten wniosek również uogólnia się na dowolne konstruktory *tej postaci*,[4] o dowolnej liczbie argumentów.

[4]Ograniczenia związane z postacią definicji bywają istotne. W przypadku „sprytnego konstruktora" z zadania domowego o kopcach *sformułowanie* odpowiednich równań byłoby bardziej skomplikowane.

Możemy teraz wrócić do naszych drzew. Podobnie jak w przypadku list, konstrukcja zasady indukcji będzie odpowiadała konstrukcji predykatu definiującego strukturę danych. Żeby uprościć sformułowanie zasady, podobnie jak w przypadku list, bierzemy pod uwagę równania dotyczące poszczególnych konstruktorów.[5]

Zasada indukcji dla drzew binarnych. Dla dowolnej własności P, jeśli

- zachodzi $P(\texttt{leaf})$ i
- dla dowolnych x, l, r, jeśli zachodzą $P(l)$ i $P(r)$, to zachodzi $P((\texttt{node}\ x\ l\ r))$,

to dla dowolnego t, jeśli zachodzi `(tree? t)`, to zachodzi $P(t)$.

Zauważmy że (analogicznie jak w przypadku list) mamy tyle przypadków do rozpatrzenia co alternatyw w definicji predykatu, i tyle założeń indukcyjnych w każdym z nich ile rekurencyjnych wywołań predykatu w odpowiedniej gałęzi alternatywy.

Przykład 3 Przypomnijmy definicję procedury `mirror`, zwracającej „lustrzane odbicie" danego drzewa:

```
(define (mirror t)
  (if (leaf? t)
      leaf
      (node (node-elem t)
            (mirror (node-right t))
            (mirror (node-left t)))))
```

Udowodnijy że `mirror` dwukrotne odbicie lustrzane danego drzewa jest równoważne początkowemu drzewu, tj., że dla dowolnego t, jeśli `(tree? t)`, to `(mirror (mirror t))` $\equiv t$.

Jak widać, struktura naszego twierdzenia odpowiada konkluzji zasady indukcji dla drzew — a więc możemy jej użyć, przyjmując jako własność $P(x) =$ `(mirror (mirror x))` $\equiv x$. Ponieważ zasada indukcji ma dwa założenia, musimy sprawdzić czy obydwa zachodzą.

W pierwszym z przypadków musimy sprawdzić czy zachodzi $P(\texttt{leaf})$. Liczymy zatem:

$$
\begin{aligned}
(\texttt{mirror}\ (\texttt{mirror}\ \texttt{leaf})) &\equiv && \text{(z definicji mirror i własności leaf?)}\\
(\texttt{mirror}\ \texttt{leaf}) &\equiv && \text{(jak wyżej)}\\
\texttt{leaf} && && \text{co kończy ten przypadek}
\end{aligned}
$$

[5] Dzięki temu zasadę indukcji wyrażamy wyłącznie za pomocą konstruktorów — alternatywnie moglibyśmy sformułować ją używając predykatów i selektorów.

W drugim przypadku ustalmy dowolne x, l i r takie że $P(l)$ i $P(r)$ zachodzą. Musimy teraz pokazać że zachodzi `(mirror (mirror (node x l r)))` $\equiv$ `(node x l r)`. W tym celu liczymy:

$$\begin{aligned}
&(\texttt{mirror}\ (\texttt{mirror}\ (\texttt{node}\ x\ l\ r))) \equiv \\
&\text{(z definicji } \texttt{mirror}\text{, reguł obliczania i specyfikacji } \texttt{node}\text{)} \\
&(\texttt{mirror}\ (\texttt{node}\ x\ (\texttt{mirror}\ l)\ (\texttt{mirror}\ r))) \equiv \\
&\text{(jak wyżej)} \\
&(\texttt{node}\ x\ (\texttt{mirror}\ (\texttt{mirror}\ l))\ (\texttt{mirror}\ (\texttt{mirror}\ r))) \equiv \\
&\text{(w końcu możemy użyć naszych założeń indukcyjnych)} \\
&(\texttt{node}\ x\ l\ r)
\end{aligned}$$

Pokazaliśmy oczekiwaną równość, a więc zakończyliśmy dowód. Zwróćmy uwagę że wypisane kroki dowodu tym razem były znacznie większe niż poprzednio — dobrym ćwiczeniem jest rozpisanie takiego dowodu dokładnie, żeby rozumieć krok po kroku co dzieje się w każdym momencie dowodu.[6]

Przykład 4 Zajmiemy się teraz bardziej skomplikowanym przykładem: spróbujemy pokazać, że dwie definicje spłaszczania naszych drzew są równoważne. Prosta definicja procedury spłaszczającej drzewo może wyglądać następująco:

```
(define (flatten t)
  (if (leaf? t)
      null
      (append (flatten (node-left t))
              (cons (node-elem t)
                    (flatten (node-right t))))))
```

Definicja ta jest prosta i intuicyjna: odpowiada naszemu wyobrażeniu o zawartości drzewa w porządku in-order, ale jest bardzo nieefektywna. Powodem jest wykorzystanie procedury append: w każdym wywołaniu rekurencyjnym musimy skonstruować listę reprezentującą zawartość lewego poddrzewa a następnie *skopiować* jej strukturę aby dołączyć ją do elementów prawego poddrzewa. Bardziej efektywna wersja wygląda następująco:

[6] Oczekiwany stopień dokładności dowodu to stałe pytanie w kwestii kolokwiów i egzaminów. Na ustnym egzaminie można sobie wyobrazić odpowiedź „Indukcja strukturalna i proste obliczenia" — ale wtedy egzaminator może zapytać o konkretny przypadek gdzie podejrzewa istnienie jakichś trudności, albo zapytać o wybraną własność. W przypadku egzaminu pisemnego nie ma takiego luksusu, dlatego trzeba zawczasu zadbać żeby istotne kroki były przedstawione — i nie ma tu jednej precyzyjnej reguły. Na pewno nie ma sensu rozbijać rozwinięcia definicji na przywołanie definicji i aplikację lambdy czy pokazywać kroków w których mamy formy postaci `(if true . . . )` — ale też pomyłka przy kompresowaniu większych obliczeń do jednego kroku może być kosztowna, bo na papierze trudno czasem odróżnić ją od zwykłej ściemy.

```
(define (flatten-acc t)
  (flat-app t null))

(define (flat-app t xs)
  (if (leaf? t)
      xs
      (flat-app (node-left t)
                (cons (node-elem t)
                      (flat-app (node-right t) xs)))))
```

Dzięki zastosowaniu akumulatora, procedura ta jest znacznie bardziej efektywna: każdy element jest dodawany do listy wynikowej dokładnie raz: przy przetwarzaniu poddrzewa którego jest on korzeniem, a więc nie tworzymy komórek pamięci które nie są nam potrzebne w wyniku — tzw. *nieużytków*. Nie jest jednak oczywiste czy te dwie definicje są równoważne: spróbujmy to udowodnić.

Łatwo sformułować twierdzenie, którego potrzebujemy: dla dowolnego t, jeśli `(tree? t)` to `(flatten t)` $\equiv$ `(flatten-acc t)`. Jednak powinniśmy patrzeć na to sformułowanie nieco podejrzliwie: ostatecznie, `flatten-acc` nie jest zdefiniowana rekurencyjnie względem struktury drzewa, czyli próba bezpośredniego dowodu indukcyjnego prawdopodobnie się zatnie. W tym miejscu warto (zanim pójdziemy dalej) zastanowić się jaką własnością możemy powiązać dwie procedury które *są* zdefiniowane rekurencyjnie względem struktury drzewa: `flatten` i `flat-app`.

Po zastanowieniu się czas na sformułowanie lematu. Będzie on brzmiał następująco: dla dowolnego t i xs, jeśli `(tree? t)` to `(append (flatten t) xs)` $\equiv$ `(flat-app t xs)`. Żeby udowodnić ten lemat, zauważmy że ma on postać odpowiadającą naszej zasadzie indukcji, musimy tylko ustalić własność P. Tym razem musi ona być odpowiednio ogólna (zauważmy że wywołania rekurencyjne `flat-app` używają *różnych* list jako drugiego argumentu):

$$P(x) = \text{dla dowolnego } xs\text{, } \texttt{(append (flatten } x\texttt{) } xs\texttt{)} \equiv \texttt{(flat-app } x\ xs\texttt{)}$$

Podobnie jak w poprzednim przykładzie, musimy rozważyć dwa przypadki. W pierwszym musimy pokazać że zachodzi $P(\texttt{leaf})$. Ustalmy więc pewne xs i liczmy:

$$\begin{aligned}
(\texttt{append (flatten leaf}) \ xs) &\equiv && \text{(licząc z definicji } \texttt{flatten}\text{)} \\
(\texttt{append leaf } xs) &\equiv && \text{(licząc z definicji } \texttt{append}\text{)} \\
xs &\equiv && \text{(licząc z definicji } \texttt{flat-app}\text{)} \\
\texttt{(flat-app leaf } xs\texttt{)} && && \text{co kończy ten przypadek}
\end{aligned}$$

Pozostaje przypadek dla wierzchołków wewnętrznych. Weźmy zatem dowolne x, l i r takie że $P(l)$ i $P(r)$ zachodzą i dowolne xs. Mamy pokazać że `(append (flatten (node x l r)) xs)` $\equiv$ `(flat-app (node x l r) xs)`. Liczmy zatem:

$$\begin{aligned}
&\texttt{(append (flatten (node } x\ l\ r\texttt{)) } xs\texttt{)} \equiv \\
&\qquad \text{(licząc z definicji } \texttt{flatten}\text{)} \\
&\texttt{(append (append (flatten } l\texttt{) (cons } x\ \texttt{(flatten } r\texttt{))) } xs\texttt{)} \equiv \\
&\qquad \text{(z łączności append)} \\
&\texttt{(append (flatten } l\texttt{) (append (cons } x\ \texttt{(flatten } r\texttt{)) } xs\texttt{))} \equiv \\
&\qquad \text{(licząc z definicji append)} \\
&\texttt{(append (flatten } l\texttt{) (cons } x\ \texttt{(append (flatten } r\texttt{) } xs\texttt{)))} \equiv \\
&\qquad \text{(z założenia indukcyjnego dla } r\text{)} \\
&\texttt{(append (flatten } l\texttt{) (cons } x\ \texttt{(flat-app } r\ xs\texttt{)))} \equiv \\
&\qquad \text{(z założenia indukcyjnego dla } l\text{)} \\
&\texttt{(flat-app } l\ \texttt{(cons } x\ \texttt{(flat-app } r\ xs\texttt{)))} \equiv \\
&\qquad \text{(licząc z definicji } \texttt{flat-app}\text{)} \\
&\texttt{(flat-app (node } x\ l\ r\texttt{) } xs\texttt{)}
\end{aligned}$$

Zauważmy że w dwóch miejscach użyliśmy założeń indukcyjnych z *różnymi* listami jako drugim argumentem. W przypadku drzewa r była to lista xs (ta sama, z którą zaczynaliśmy), w przypadku drzewa l — lista `(cons x (flat-app r xs))`. Było to możliwe dzięki temu że wybraliśmy odpowiednio ogólną własność P. Tym samym zakończyliśmy dowód lematu.

Pozostaje jeszcze pokazać że z naszego lematu wynika oczekiwane twierdzenie. W tym celu liczymy:

$$\begin{aligned}
\texttt{(flatten } t\texttt{)} &\equiv && \texttt{null} \text{ jest elementem neutralnym appenda} \\
\texttt{(append (flatten } t\texttt{) null)} &\equiv && \text{z lematu} \\
\texttt{(flat-app } t\ \texttt{null)} &\equiv && \text{z definicji } \texttt{flatten-acc} \\
\texttt{(flatten-acc } t\ \texttt{null)} & && \text{co kończy dowód}
\end{aligned}$$

Widzimy zatem, że formułując odpowiednie lematy (w tym przypadku musieliśmy użyć dodatkowo dwóch lematów z ćwiczeń!) możemy zgrabnie udowodnić równoważność naszych dwóch definicji. Oczywiście istotna część trudności w dowodzeniu to sformułowanie właściwych lematów: jest to umiejętność którą trzeba wyćwiczyć.

3.3 Indukcja dla struktur drzewiastych

Nie będziemy tutaj dawać formalnego „przepisu" na zasadę indukcji, tylko zauważymy że zwłaszcza dla drzewiastych struktur danych jest ona naturalnym uogólnieniem tych rozważanych powyżej. Żeby zobaczyć mniej klasyczny przypadek, rozważmy drzewa które mają wierzchołki wewnętrzne dwóch rodzajów — z jednym lub trojgiem dzieci — i dwa różne rodzaje liści, z których drugi etykietowany. Dla wierzchołków pierwszego rodzaju przyjmiemy konstruktor `node1`, selektor `node1-child` i predykat `node1?`; dla wierzchołków drugiego rodzaju — konstruktor `node3`, selektory odpowiednio `node3-child1/2/3` i predykat `node3?`; dla liścia pierwszego rodzaju zmienną `leafa` i predykat `leafa?` i dla liścia drugiego rodzaju konstruktor `leafb`, selektor `leafb-label` i predykat `leafb?`. Wszystkie powyższe struktury danych powinny spełniać równości analogiczne do opisanych wcześniej.

Możemy teraz zdefiniować predykat opisujący te dziwne drzewa:

```
(define (weird-tree? x)
  (or (leafa? x)
      (leafb? x)
      (and (node1? x)
           (weird-tree? (node1-child x)))
      (and (node3? x)
           (weird-tree? (node3-child1 x))
           (weird-tree? (node3-child2 x))
           (weird-tree? (node3-child3 x)))))
```

Jak powinna wyglądać zasada indukcji dla dziwnych drzew? Na pewno musi mieć cztery założenia, dla czterech różnych konstrukcji. Pierwsze dwa przypadki nie będą miały założeń indukcyjnych, trzeci będzie miał jedno, a czwarty — trzy. Całość będzie wyglądać następująco.

Zasada indukcji dla dziwnych drzew Dla dowolnej własności P, jeśli

- zachodzi $P($`leafa`$)$;
- dla dowolnego l zachodzi $P($`(leafb` l`)`$)$;
- dla dowolnego t, jeśli zachodzi $P(t)$ to zachodzi $P($`(node1` t`)`$)$ i
- dla dowolnych t_1, t_2, t_3, jeśli zachodzą $P(t_1), P(t_2)$ i $P(t_3)$ to zachodzi również $P($`(node3` t_1 t_2 t_3`)`$)$,

to dla dowolnego t, jeśli zachodzi `(weird-tree?` t`)`, to zachodzi $P(xs)$.