# magisterka brudnopis

marekbauer07

July 2022

## 1 Cel pracy

## 2 Czym są typy ilorazowe

W algebrze abstrakcyjnej abstrakcyjnej zbiór pierwotny $T$, w którym utożsamiamy elementy zgodnie z relacją równoważności ($\sim$) nazywamy strukturą ilorazową. Oznaczmy ją symbolem $T/\sim$. Dobrym przykładem tego typu struktury jest grupa z modularną arytmetyką. Każdy z nas zaznajomiony z zasadami działania zegara i nikogo nie dziwni że po godzinie dwunastej następuje godzina pierwsza. O godzinach na traczy zegara możemy myśleć jako o operacjach w grupie $\mathbb{Z}/12\mathbb{Z}$, czyli takiej, w której utożsamiamy liczby których operacja dzielenia przez 12 daje taką samą resztę.

$$\cdots \equiv -11 \equiv 1 \equiv 13 \equiv 25 \equiv 37 \equiv \dots \pmod{12} \tag{1}$$

Tak jak podpowiada nam intuicja 1:00 i 13:00 to ta sama godzina w tej arytmetyce.

### 2.1 Relacja równoważności

Żeby sformalizować typy ilorazowe, będziemy usieli najpierw dokładnie zdefiniować jakie relacje są relacjami równoważności. Każda relacja równoważności spełnia trzy własności:

**Zwrotność** - każdy element musi być w relacji sam z sobą ($a \sim a$)

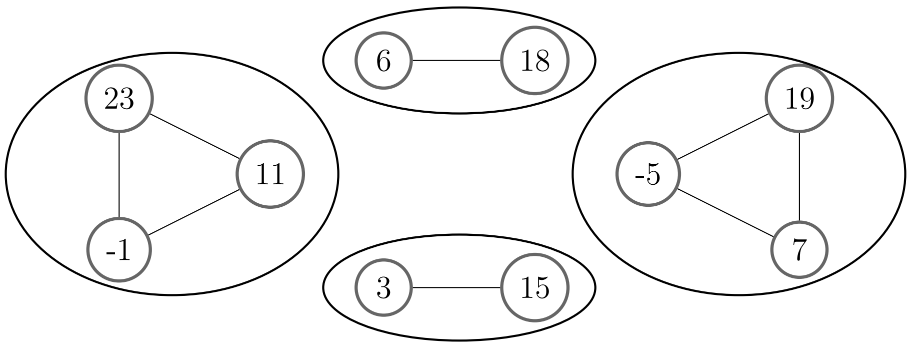
**Symetryczność** - jeśli $a$ jest w relacji z $b$ ($a \sim b$) to również relacja zachodzi w odwrotnej kolejności ($b \sim a$).

**Przechodniość** - jeśli istnieje punkt $b$ z którym zarówno $a$ i $c$ są w relacji (odpowiednio $a \sim b$ oraz $b \sim c$) to $a$ jest w relacji z $c$ ($a \sim c$).

Najlepszym przykładem takiej relacji jest równość, po chwili zastanowienie widać iż spełnia ona wszystkie trzy wymagane własności. O relacjach równoważności możemy myśleć jako o uogólnieniu pierwotnego pojęcia równości elementów. Pozwalają one na utożsamienie różnych elementów naszego pierwotnego zbioru z sobą np. godzinę 13:00 z 1:00. Innym przykładem może być utożsamienie różnych reprezentacji tej samej liczby $\frac{1}{2}$ z $\frac{2}{4}$. Z punktu widzenia teorii mnogości utożsamione z sobą elementy tworzą klasy abstrakcji. Tak naprawdę w tej teorii $T/\sim$ jest rodziną klas abstrakcji, inaczej mówiąc rodziną zbiorów elementów które zostały utożsamione relacją ($\sim$).

```
Class equivalance_relation {A: Type} (R: A -> A -> Prop) := {
  equiv_refl  : forall x: A, R x x;
  equiv_sym   : forall x y: A, R x y -> R y x;
  equiv_trans : forall x y z: A, R x y -> R y z -> R x z;
}.
```

Kod źródłowy 2.1: Klasa relacji równoważności zapisana w Coq




Rysunek 1: Przykład elementów utożsamionych z sobą w grupie $\mathbb{Z}/12\mathbb{Z}$, linie oznaczają elementy będące z sobą w relacji równoważności, a elipsy klasy abstrakcji wyznaczone przez tą relację równoważności

## 2.2 Spojrzenie teorii typów

Jako że praca skupia się na implementacji typów ilorazowych w Coqu, stąd też skupimy się na spojrzeniu teorii typów na ilorazy, w przeciwieństwie do spojrzenia teorio mnogościowego. W teorii typów $T/\sim$ będziemy nazywać typem ilorazowym generowanym przez relację równoważności $(\sim)$ n typie pierwotnym $T$. Będziemy oznaczać $a = b$ w typie $T/\sim$ wtedy i tylko wtedy gdy $a \sim b$. Widzimy zatem iż każdy element typ $T$ jest również elementem typu $T/\sim$. Natomiast nie wszystkie funkcje z typu $T$ są dobrze zdefiniowanymi funkcjami z typu $T/\sim$. Funkcja $f : T \to X$ jest dobrze zdefiniowaną funkcją $f : (T/\sim) \to X$ jeśli spełnia warunek, że $a \sim b$ implikuje $f(a) = f(b)$. Ten warunek jest konieczny, aby nie dało się rozróżnić utożsamionych wcześniej elementów poprzez zmapowanie ich do innego typu. Myśląc o wszystkich elementach będących z sobą w relacji $(\sim)$ jako o jednym elemencie brak tej zasady spowodowałby złamanie reguły monotoniczności aplikacji mówiącej, że $x = y \Rightarrow f(x) = f(y)$.

# 3 Relacje równoważności generowane przez funkcję normalizującą

Każda funkcja $h : T \to B$ generuje nam pewną relację równoważności $(\sim_h)$ zdefiniowaną poniżej:

$$\forall x, y \in T, x \sim_h y \iff h(x) = h(y) \tag{2}$$

Jak już wspominaliśmy równość jest relacją równoważności, stąd łatwo pokazać iż $(\sim_h)$ jest relacją równoważności. Dowolna funkcja $g : B \to X$ będzie teraz generować dobrze zdefiniowaną funkcję $f : T/\sim_h \to X$ w taki sposób, że $f = g \circ h$.

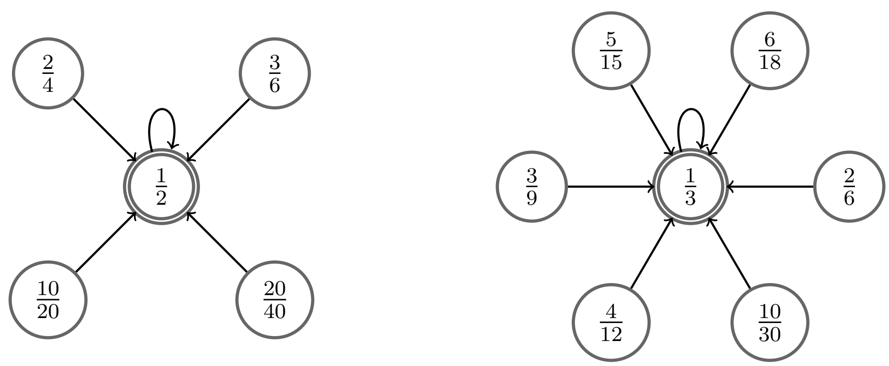
## 3.1 Funkcja normalizująca

W tej pracy skupimy się na szczególnym przypadku funkcji $h : T \to B$ w którym $B = T$. Dodatkowo będziemy wymagać aby funkcja $h$ była idempotentna. Tak zdefiniowaną funkcję h będziemy nazywać funkcją normalizującą. Zbudujmy jeszcze odrobinę intuicji wokół tak

```
Class normalizing_function {A: Type} (f: A -> A) :=
  idempotent : forall x: A, f (f x) = f x.
```

Kod źródłowy 3.1: Klasa funkcji normalizujących

zdefiniowanych funkcji normalizujących. Jak wiemy relacja równoważności łączny utożsamiane z sobą elementy w grupy, a mówiąc ściślej klasy abstrakcji. Celem funkcji normalizujących jest wyznaczenie reprezentanta każdej z klas abstrakcji. Obrazem tej funkcji będzie właśnie zbiór naszych reprezentantów lub inaczej elementów w postaci normalnej. Warunek idempotencji jest konieczny do tego aby obrazem elementu w postaci normalnej był on sam, gdyż nie wymaga on już normalizacji.




Rysunek 2: Przykład działania funkcji normalizującej dla reprezentacji liczby wymiernych w postaci $\mathbb{Z} \times \mathbb{N}$

## 3.2 Przykłady funkcji normalizujących

Wykorzystajmy tutaj już wcześniej przytoczone liczby wymierne. Jednym z sposobów na zdefiniowanie postaci normalnej liczby wymiernej rozumianej jako $\mathbb{Z} \times \mathbb{N}$ jest wymuszenie, aby licznik był względnie pierwszy z mianownikiem. Wprawne oko zapewne zauważy, iż 0 nie ma kanonicznej postawi w takiej definicji, ale możemy arbitralnie ustalić, że jego postacią normalną będzie $\frac{0}{1}$. W takim przypadku funkcja normalizująca powinna dzielić licznik i mianownik przez ich największy wspólny dzielnik. W przypadku par nieuporządkowanych, ale na których istnieje jakiś porządek liniowy postacią normalną może być para uporządkowana w której mniejszy element występuje na początku, a funkcją normalizującą będzie funkcja sortująca dwa elementy. Ostatnim, a zarazem najprostszym przykładem niech będzie postać normalna elementów w arytmetyce modularnej o podstawie $m$. Tutaj sama operacja będzie naszą funkcją normalizującą, a więc zostawimy jedynie elementy od zera do $m - 1$.

# 4 Ograniczenia wynikające z użycia normalizacji w typach ilorazowych

Wykorzystanie funkcji normalizujących w definicji typów ilorazowych jest bardzo wygodne z względu na fakt, że może posłużyć nam do ograniczenia liczby reprezentantów danej klasy abstrakcji do tylko jednego, tego który jest w postaci normalnej. Jest to szczególnie użyteczne w Coqu, gdyż ten język natywne nie wspiera typów ilorazowych. A utożsamianie elementów z sobą za pomocą aksjomatów z regały prowadzi do wprowadzenia sprzeczności do systemu. Takie podejście niesie jednak z sobą dojść poważne ograniczenie w postaci konieczności istnienia funkcji normalizującej dla danego typu ilorazowego, który chcemy zdefiniować. W teorii mnogości z aksjomatem wyboru, możemy zawsze wykorzystać ów aksjomat mówiący o tym że z każdej rodziny niepustych zbiorów możemy wybrać zbiór selektorów, w naszym przypadku zbiór elementów w postaci normalnej. Niestety nie jest on jednak obliczalny, co powoduje, że Coq nie możemy skorzystać z tego aksjomatu. Jedynie wyszczanie postaci normalnej dla typów z funkcją wyboru oraz rozstrzygalnej relacji równoważności możemy zautomatyzować.

```
Class choosable {A: Type} :=
  xchoose : forall P: A -> bool, (exists x: A, P x = true) -> A.
```

Kod źródłowy 4.1: Definicja typu z funkcją wyboru w Coq

Taka funkcja pozwala na łatwe wyznaczenie postaci kanoniczej, gdyż każde element jest sam z sobą w relacji, znalezienie świadka jest trywialne, co pozwala nam Nie we wszystkich przypadkach jest to jednak możliwe. Najprostszym przykładem jest wspominana wcześniej para nieuporządkowana. Jeśli na elementach tej pary istnieje jakiś porządek zupełny, wtedy jak już mówiliśmy możemy ją posortować zapominając jednocześnie oryginalny porządek elementów. Jeśli natomiast nie będziemy mieć do dyspozycji takiego porządku to nie sposób wybrać które z ustawiań jest bardziej normalne $\{\square, \bigcirc\}$ a może $\{\bigcirc, \square\}$.

TU COŚ JESZCZE TRZEBA DOPISAĆ

# 5 Czym jest podtypowanie

Koncept podtypowania jest dosyć intuicyjny. Wyobrazimy sobie na chwilę typy jako zbiory elementów należących do danego typu, wtedy podtyp to będzie podzbiorem naszego pierwotnego typu. Pozwala ono na wyrzucanie z typu wszystkich niepożądanych elementów z naszego typu pierwotnego. Dla przykładu wyobraźmy sobie, że piszemy funkcję wyszukiwania binarnego i jako jej argument chcielibyśmy otrzymać posortowaną listę. W tym celu właśnie możemy użyć podtypowania, wymuszając aby akceptowane były jedynie listy które spełniają predykat posortowania. Najbardziej ogólna definicja podtypowania 5.1 wymaga

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x:A, P x -> sig P.

Record sig' (A : Type) (P : A -> Prop) : Type := exist' {
    proj1' : A;
    proj2' : P proj1';
}
```

Kod źródłowy 5.1: Dwie równoważne definicje podtypowania w Coqu.

wykorzystania typów zależnych, które nie są dostępne w większości języków programowania, stąd też w większości języków programowania na programiście spoczywa obowiązek upewnienie się czy lista jest posortowana, gdyż ekspresywność języka jest zbyt uboga, aby zapisać takie wymagania. Coq w bibliotece standardowej `Coq.Init.Specif` posiada zdefiniowane `sig` wraz z notacją `{a : A | P a}`, która ma przypominać matematyczny zapis $\{x \in A : P(x)\}$. Ta sama biblioteka posiada identyczną konstrukcje, gdzie jednak `(P : A -> Prop)` zostało zamienione na `(P : A -> Type)` jest to uogólniona definicja podtypowania, nazywana sigma typem. Jest to para zależna w której drugi element pary zależy do pierwszego. Posiada ona

```
Inductive sigT (A:Type) (P:A -> Type) : Type :=
    existT : forall x:A, P x -> sigT P Q.
}
```

Kod źródłowy 5.2: Definicja definicja sigma typu z biblioteki standardowej Coqa.

również zdefiniowaną notację `{a : A & P}`, nawiązuje ona do tego `a` jest zarówno typu `A` jak i `P`.

## 5.1 Dlaczego w ogóle wspominamy o podtypowaniu?

Podtypowanie jest to dualna konstrukcja do typów ilorazowych o których mowa w tej pracy. Ten rozdział poświęcamy im z następującego powodu - Coq nie wspiera podtypowania. Nie możemy w żaden inny sposób niż aksjomatem wymusić równości dwóch różncyh elementów danego typu. Używanie aksjomatów jest jednak nie praktyczne, gdyż niszczy obliczalność dowodu, a na dodatek bardzo łatwo takimi aksjomatami doprowadzić do sprzeczności w logice Coqa. Dlatego zastąpimy tutaj koncept typów ilorazowych konceptem podtypowania, który będzie wymuszać istnienie jedynie normalnych postaci danej klasy abstrakcji w naszym typie ilorazowym. Pozwoli nam to na pracę jak na typach ilorazowych korzystając z ograniczonej liczby narzędzi, które dostarcza nam Coq.

# 6 Dualna? Co to oznacza?

W poprzedniej sekcji wspomnieliśmy, że podtypowanie jest pojęciem dualnym do ilorazów, tu wyjaśnimy co to oznacza. Dualizm jest pojęciem ze świata teorii kategorii, aby zrozumieć tą

```
Record quotient {A: Type} {f: A -> A} (n: normalizing_function f) := {
  val: A;
  proof: val = f val
}.
```

Kod źródłowy 5.3: Definicja podtypu kanonicznych postaci względem funkcji normalizującej f.

sekcję wymagana jest bardzo podstawowa wiedza z tego zakresu. Jeśli ktoś jej natomiast nie posiada może spokojnie ją pominąć gdyż stanowi bardziej ciekawostkę niż integralną część tej pracy. Mówiąc formalnie jeśli $\sigma$ jest konstrukcją w teorii kategorii to konstrukcję dualną do niej $\sigma^{\mathrm{op}}$ definiujemy poprzez:

- zamianę pojęć elementu początkowego i końcowego nawzajem,
- zmianę kolejności składania morfinizmów.

Mając to już za sobą możemy powiedzieć prostym językiem, że dualność polega na zmianie kierunku strzałek w naszej konstrukcji. Znając to pojęcie możemy zadać sobie pytanie gdzie w

$$\begin{tikzcd} & X \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\ A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd} \qquad \begin{tikzcd} & X & \\ A \arrow[ur, "f_1"] \arrow[r, "\eta_1"'] & A \oplus B \arrow[u, dashed, "f"] & B \arrow[ul, "f_2"'] \arrow[l, "\eta_2"] \end{tikzcd}$$

Rysunek 3: Przykład dwóch dualnych konstrukcji. Po lewej stronie widzimy produkt, a po prawej co-produkt. Oba diagramy komutują.

ilorazach i podtypowaniu występują jakieś strzałki, których kierunki mielibyśmy zamieniać? Mianowicie występują w odpowiednio pushoutach oraz pullbackach.

## 6.1 Czym jest pushout?

Na rysunku 4 możemy zobaczyć diagram definiujący pojęcie pushoutu. Widzimy, że powstaje on z pewnych dwóch morfinizmów $f : X \to A$ oraz $g : X \to B$. Ponieważ diagram ten komutuje wiemy, że $\eta_1 \circ f = \eta_2 \circ g$. Nasz pushout $P$ jest najlepszym takim obiektem dla którego diagram zachowuje tą własność. Najlepszy definiujemy jako, dla każdego innego obiektu (na diagramie $Q$), dla którego zewnętrzna część $(X, A, B, Q)$ diagramu komutuje, istnieje unikatowy (dokładnie jeden) morfizm $u$ z $P$ do $Q$. Warto zaznaczyć, że nie dla każdych dwóch morfinizmów $f : X \to A$ oraz $g : X \to B$ istnieje pushout, jeśli jednak istnieje to jest unikatowy z dokładnością do unikatowego izomorfizmu.

$$
\begin{tikzcd}
X \arrow[r, "g"] \arrow[d, "f"'] & B \arrow[d, "\eta_2"] \arrow[ddr, bend left, "\zeta_2"] & \\
A \arrow[r, "\eta_1"'] \arrow[drr, bend right, "\zeta_1"'] & P \arrow[dr, dashed, "u"] & \\
 & & Q
\end{tikzcd}
$$

Rysunek 4: Diagram definiujący pushout $P$. Diagram komutuje.

## 6.2 Przykład pushoutu w kategorii *Set*

W definicji pushoutu łatwo możemy zobaczyć morfizmy (strzałki), natomiast trudno odnaleźć ilorazy o których jest ta praca. Dużo łatwiej wyrobić swoją intuicję na bardziej przyziemnym przykładzie. Przenieśmy się w tym celu do kategorii *Set*. Oznacza to, że nasze obiekty staną się zbiorami, a morfizmy (strzałki) funkcjami na zbiorach. Na rysunku 4 możemy zauważyć, że $A$, $B$ oraz $P$ tworzą coś na kształt co-produktu. Zatem warto zacząć definiowanie $P$ właśnie od $A \oplus B$. Wszystko byłoby wszystko dobrze gdyby nie to, że nasz diagram powinien komutować, a więc dla każdego $x \in X$ wiemy, że $\eta_1(f(x)) = \eta_2(g(x))$. Aby to zapewnić musimy utożsamić z sobą $f(x) \sim g(x)$, dla każdego $x \in X$. Dzięki podzieleniu przez tą relację zapewnimy sobie komutowane wewnętrznej części diagramu $(X, A, B, P)$. Nie możemy jednak wziąć dowolnej relacji $\sim$ spełniającej ten warunek, gdyż musimy zapewnić istnienie funkcji $u$ do każdego innego zbioru dla którego ten diagram będzie komutował, oznacza to że musi istnieć surjekcja ze zbioru $P$ do $Q$. Z uwagi na komutowanie funkcja $u$ musi spełniać $u \circ \eta_1 = \zeta_1$ oraz $u \circ \eta_2 = \zeta_2$. Nasza relacja równoważności musi zatem być tą najdrobniejszą, aby spełnić ten warunek. Jeśli nasz pushout $P$ jest równy $(A \oplus B)/\sim$ diagram 4 będzie komutował. Widzimy zatem, że pushout rzeczywiście ma jakiś związek z typami ilorazowymi, w których to $X$ definiuje które elementy zostaną z sobą utożsamione.

## 6.3 Sklejanie dwóch odcinków w okrąg, czyli pushout

Rozważyliśmy bardziej przyziemny, lecz dojść abstrakcyjny przykład w kategorii *Set*. Skonstruujmy nieco bardziej wizualny przykład, czyli w naszym przypadku okrąg na rysunku 5. Upewnijmy się, iż rzeczywiście $C$ jest topologicznym okręgiem. Wiemy, że aby diagram 5

$$
\begin{tikzcd}
\{0,1\} \arrow[r, "id"] \arrow[d, "id"'] & {[0,1]} \arrow[d, "\eta_2"] \\
{[0,1]} \arrow[r, "\eta_1"'] & C
\end{tikzcd}
$$

Rysunek 5: Diagram definiujący okrąg $C$ używając pushoutu komutował $\eta_1(0) = \eta_2(0)$ oraz $\eta_1(1) = \eta_2(1)$. Skleiliśmy zatem z sobą te dwa punkty. Ponieważ $C$ musi być najlepszym obiektem dla którego diagram komutuje, to żadne inne elementy nie mogą być z sobą sklejone, a więc dla każdego $x \in (0,1)$ wiemy, że $\eta_1(x) \neq \eta_2(x)$. Więc rzeczywiście stworzyliśmy okrąg, z dwóch odcinków oraz dwóch punktów sklejeń. Ta metoda uogólnia się do wyższych wymiarów. Mając dwie $n$-wymiarowe półkule, a następnie sklejając je w wzdłuż kuli $(n-1)$-wymiarowej otrzymamy kulę $n$-wymiarową.

## 6.4 Czym jest pullback?

Wiedząc jak wygląda pushout oraz wiedząc, że pullback jest pojęciem do niego dualnym każdy powinien być w stanie narysować diagram go definiujący, możemy się mu przyjrzeć na rysunku 6. Każdy pullback jest definiowany za pomocą dwóch morfizmów $f : A \to X$

$$\begin{tikzcd}
Q \arrow[dr, dashed, "u"] \arrow[drr, bend left, "\zeta_2"] \arrow[ddr, bend right, "\zeta_1"'] & & \\
& P \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & X
\end{tikzcd}$$

Rysunek 6: Diagram definiujący pullback $P$. Diagram komutuje.

oraz $g : B \to X$, które tworzą strukturę przypominającą co-produkt. Z komutacji wiemy, że $f \circ \pi_1 = g \circ \pi_2$. Podobnie jak w przypadku pushoutu, tu również aby $P$ był pullbackiem to musi być najlepszym obiektem, dla którego diagram komutuje. Oznacza to że dla każdego innego (na diagramie $Q$), dla którego zewnętrza część diagramu $(X, A, B, Q)$ komutuje to istnieje unikatowy morfizm $u$ z $Q$ do $P$, który oczywiście zachowuje własność komutacji diagramu. Ponownie nie dla każdych dwóch morfizmów $f : A \to X$ oraz $g : B \to X$ istnieje pushout, jeśli jednak istnieje to jest unikatowy z dokładnością do unikatowego izomorfizmu.

## 6.5 Przykład pullbacku w kategorii *Set*

Ponownie aby zobaczyć podtypowanie w zdefiniowanym powyżej pullbacku omawiane w tym rozdziale podtypowanie przeniesiemy się do prostszego świata kategorii $Set$, gdzie żyją zbiory oraz funkcje na zbiorach. Jak widzimy na diagramie 6 struktura $A$, $B$ oraz $P$ przypomina coś na kształt produktu. Zacznijmy więc definiowanie $P$ właśnie od $A \times B$. Wiemy jednak, że aby diagram komutował to dla każdego elementu $p \in P$ musi zachodzić własność $f(\pi_1(p)) = g(\pi_2(p))$. Ponieważ wstępnie $P = A \times B$ to dla każdej pary $(a,b) \in A \times B$ zachodzi $f(a) = g(b)$. Wystarczy teraz wyrzucić wszystkie elementy nie spełniające tego warunki otrzymując ostatecznie $P = \{(a,b) \in A \times B : f(a) = g(b)\}$. Upewnijmy się jeszcze iż rzeczywiście to jest najlepszy wybór. Weźmy dowolny inny zbiór $Q$ wraz z funkcjami $\zeta_1$ oraz $\zeta_2$ dla którego zewnętrzny diagram 6 komutuje i zdefiniujmy morfizm $u$ jako dla każdego $q \in Q$ mamy $u(q) =$

$(\zeta_1(q), \zeta_2(q))$. Ponieważ funkcje $\pi_1$ i $\pi_2$ są zwykłymi projekcjami to własności $\zeta_1 = \pi_1 \circ u$ oraz $\zeta_2 = \pi_2 \circ u$ mamy za darmo, a cała reszta diagramu komutuje z względu na na komutowanie zewnętrznej części diagramu. Możemy na w tym miejscu zauważyć iż rzeczywiście pullback ma związek z podtypowaniem, poprzez wyrzucanie elementów które nie spełniają równości $f(a) = g(b)$.

## 6.6 Pary liczb o tej samej parzystości, czyli pullback

Mamy już za sobą definicję, oraz przykład w kategorii *Set*. Możemy teraz przejść do stworzenia prostego podtypu, w tym przykładzie par liczb całkowitych o tej samej parzystości. Na

$$\begin{tikzcd} P \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & \mathbb{Z} \arrow[d, "f"] \\ \mathbb{Z} \arrow[r, "f"'] & \{0,1\} \end{tikzcd}$$

Rysunek 7: Diagram definiujący pary liczb całkowitych o tej samej parzystości $P$ używając pullbacku

diagramie 5 widzimy definicję pullbacku $P$ za pomocą morfizmu $f : \mathbb{Z} \to \{0, 1\}$, zdefiniowanej jako $f(n) = n \bmod 2$. Jak wiemy z poprzedniego przykładu $P = \{(n, m) \in \mathbb{Z} \times \mathbb{Z} : n \bmod 2 = m \bmod 2\}$. A więc jest to zbiór to pary dwóch liczb całkowitych, które przystają do siebie molulo 2, czyli mówiąc inaczej mają tą samą parzystość.

## 6.7 Konkluzja

Jak więc zobaczyliśmy na poprzednich przykładach w teorii kategorii pushouty pozwalają nam na wyznaczenie obiektów, które utożsamiają z sobą pewne elementy względem pewnej relacji równoważności, generowanej przez morfizmy. Czyniąc je odpowiednikiem typów ilorazowych. Natomiast pullbacki pozwalają nam stworzyć obiekty z elementami które spełniają pewien zadany przez morfizmy warunek, czyniąc z nich odpowiedniki podtypów. Ponieważ te pojęcia są dualne co możemy zobaczyć na diagramach 4 oraz 6, to możemy mówić o tych pojęciach jako dualnych do siebie nawzajem.

# 7 Unikatowość reprezentacji w podtypowaniu

Podtypowanie w Coqu nie dostarcza nam jednak niestety tak przyjemnego interfejsu, jak moglibyśmy się spodziewać po teorio zbiorowym podejściu do tego konceptu. W teorii zbiorów jesteśmy przyzwyczajeni, że zapisów w stylu $8 \in \{x \in \mathbb{N} : \text{even}(x)\}$, w Coq natomiast zapis `8 : {x : nat | even x}` powoduje konflikt typów, gdyż 8 jest typu `nat`, a nie typu `{x : nat | even x}`. Wynika to z definicji `sig`, jest to para zależna w związku z tym, aby skonstruować element tego typu potrzebujemy dwóch składników, wartości oraz dowodu, że ta wartość spełnia spełnia wymagany przez podtypowanie predykat, w tym przypadku `even`.

```
Definition even (x: nat) : Prop := exists (t : nat), t + t = x.

Lemma eight_is_even : even 8.
Proof. red. exists 4. cbn. reflexivity. Qed.

Check (exist _ 8 eight_is_even) : {x : nat | even x}.
```

Kod źródłowy 7.1: Przykład elementu typu naturalnej liczby parzystej w Coqu

Takie zdefiniowane podtypowanie rodzi pytanie o unikatowość reprezentacji. Cała koncepcja używania podtypowania do reprezentacji typów ilorazowych opiera się na tym, że będzie istnieć jedynie jeden element w postaci normalnej dla każdej klasy abstrakcji. Istnienie wielu takich elementów różniących się jedynie dowodem uniemożliwiłoby zastosowanie podtypowania do tego celu. Niestety twierdzenia 7.3 nie można udowodnić w Coq bez dodatkowych

```
Theorem uniqnes_of_representation : forall (A : Type) (P : A -> Prop)
  (x y  : {a : A | P a}), proj1_sig x = proj1_sig y -> x = y.
```

Kod źródłowy 7.2: Twierdzenie mówiące o unikalności reprezentacji w podtypowaniu

aksjomatów, natomiast wersja tego twierdzenia dla `sigT` jest po prostu fałszywa. Pozostaje nam zatem zredukować oczekiwania, lub dodać dodatkowe założenia.

## 7.1 Dodatkowe aksjomaty

Pomimo iż w tej pracy unikamy używania dodatkowych założeń spoza Coq warto rozważyć jakie rezultaty dało by ich zastosowanie.

### 7.1.1 Aksjomat irrelewancji

Jest to aksjomat mówiący o tym, że nie ma różnicy między dowodami tego samego twierdzenia. Jak możemy się domyśleć mając tak potężne narzędzie bez trudu możemy udowodnić

```
Definition Irrelevance := forall (P: Prop) (x y: P), x = y.
```

Kod źródłowy 7.3: Definicja irrelewancji w Coqu

twierdzenie 7.3. Dodatkowo możemy udowodnić, że nasze unikatowość reprezentacji jest tak naprawę równoważna aksjomatowi irrelewacji dowodów.

### 7.1.2 Aksjomat K

Aksjomat ten został wymyślony przez Habil Streicher w swojej pracy "Investigations Into Intensional Type Theory" [?]. My posłużymy się jego nieco zmodyfikowaną wersją (UIP -

```
Theorem irrelevance_uniqnes : Irrelevance -> forall (A: Type) (P: A -> Prop)
  (x y: {z: A| P z}), proj1_sig x = proj1_sig y -> x = y.
Proof.
  intros Irr A P [x_v x_p] [y_v y_p] H.
  cbn in H; subst.
  apply eq_dep_eq_sig.
  specialize (Irr (P y_v) x_p y_p); subst.
  constructor.
Qed.
```

Kod źródłowy 7.4: Dowód unikalności reprezentacji używając irrelewancji w Coq

```
Theorem uniqnes_irrelevance : (forall (A: Type) (P: A -> Prop)
  (x y: {z: A| P z}), proj1_sig x = proj1_sig y -> x = y) -> Irrelevance.
Proof.
  intros Uniq P x y.
  specialize (Uniq unit (fun _ => P) (exist _ tt x) (exist _ tt y) eq_refl).
  refine (eq_dep_eq_dec (A := unit) _ _).
  - intros. left. destruct x0, y0. reflexivity.
  - apply eq_sig_eq_dep. apply Uniq.
Qed.
```

Kod źródłowy 7.5: Dowód, że unikalności reprezentacji implikuje irrelewancję w Coq

```
Definition K := forall (A: Type) (x y: A) (p q: x = y), p = q.
```

Kod źródłowy 7.6: Aksjomat K w Coq

uniqueness of identity proofs), która lepiej oddaje konsekwencje jego użycia. Jest to nieco słabsza wersja aksjomatu irrelewancji, która mówi jedynie o irrelewanjci dowodów równości. Ma on pewną ciekawą konsekwencję, którą została opisana w [?], a mianowicie pozwala on na zanurzenie równości na parach zależnych w zwykłą równość Dowód tego twierdzenia

```
Theorem sig_jnjectivity : K -> forall (A : Type) (P : A -> Prop)
  (a : A) (p q : P a), exist P a p = exist P a q -> p = q.
```

Kod źródłowy 7.7: Twierdzenie o zanurzenie równości na parach zależnych w Coq

pominiemy, lecz można go znaleźć w dodatku Stricher.v. Aksjomat K nie jest równoważny aksjomatowi irrelewacji [?] to nie można za jego pomocą udowodnić unikalności reprezentacji w ogólności. W przypadku jednak typów ilorazowych generowanych przez funkcję normalizującą, będziemy potrzebować jedynie predykatów równości. Unikatowość reprezentacji dla

```
Inductive quotient {A: Type} {f: A -> A} (N: normalizing_function f) : Type :=
| existQ : forall x: A, x = f x -> quotient N.

Definition proj1Q {A: Type} {f: A -> A} {N: normalizing_function f}
(x : quotient N) : A := let (a, _) := x in a.
```

Kod źródłowy 7.8: Definicja podtypu postaci kanoniczych generowanych przez funkcję normalizującą f, oraz projekcji dla niego w Coq

tego typu można z łatwością udowodnić wykorzystując aksjomat K.

```
Theorem uniqnes_quotient {A: Type} (f: A -> A) (N: normalizing_function f)
    (q q': quotient N) : K -> (proj1Q q) = (proj1Q q') -> q = q'.
Proof.
  intros K H.
  destruct q, q'.
  cbn in *. subst.
  destruct (K A x0 (f x0) e e0).
  reflexivity.
Qed.
```

Kod źródłowy 7.9: Dowód unikalności reprezentacji dla podtypu postaci kanoniczych generowanych przez funkcję normalizującą f w Coq

Możemy w tym miejscu pójść nawet o krok dalej i zdefiniować, że wszystkie elementy będące w tej samej klasie abstrakcji mają unikalnego reprezentanta, przy założeniu aksjomatu K. Zaczniemy od dowodu że funkcja f rzeczywiście generuje relację równoważności 2.1 `norm_equiv`. Mając już definicję jak wygląda ta relacja równoważności możemy przejść do właściwego dowodu.

```
Definition norm_equiv {A: Type} (f: A -> A) (N: normalizing_function f)
  (x y: A) : Prop := f x = f y.

Theorem norm_equiv_is_equivalance_relation (A: Type) (f: A -> A)
  (N:normalizing_function f) : equivalance_relation (norm_equiv f N).
Proof.
  unfold norm_equiv. apply equiv_proof.
  - intro x. reflexivity.
  - intros x y H. symmetry. assumption.
  - intros x y z H H0. destruct H, H0. reflexivity.
Qed.
```

Kod źródłowy 7.10: Definicja relacji równoważności generowanej przez funkcję normalizującą w Coq

```
Theorem norm_equiv_quotient {A: Type} (f: A -> A) (N: normalizing_function f)
  (q q': quotient N) : K -> norm_equiv f N (proj1Q q) (proj1Q q') -> q = q'.
Proof.
  intros K H. destruct q, q'.
  cbn in *. unfold norm_equiv in H.
  assert (x = x0).
  - rewrite e, H, <- e0. reflexivity.
  - subst. destruct (K A x0 (f x0) e e0).
    reflexivity.
Qed.
```

Kod źródłowy 7.11: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta używając aksjomatu K

### 7.1.3 Związek między tymi aksjomatami

Jak już wspominaliśmy w ogólności aksjomat K jest szczególnym przypadkiem aksjomatu irrelewancji. Oznacza to że nie są one równoważne, jak ciekawostkę możemy powiedzić, że w świecie z ekstensjonalnością dowodów aksjomat K jest równoważny aksjomatowi irrelewancji.

```
Definition Prop_ex : Prop := forall (P Q : Prop), (P <-> Q) -> P = Q.
```

Kod źródłowy 7.12: Predykat ekstensjonalności dowodów

```
Theorem Irrelevance_K : Irrelevance -> K.
Proof.
  intros Irr A x y. apply Irr.
Qed.
```

Kod źródłowy 7.13: Dowód że irrelewancja implikuje aksjomat K

```
Theorem K_Irrelevance : Prop_ex -> K -> Irrelevance.
Proof.
  unfold Prop_ex, K, Irrelevance.
  intros Prop_ex K P x y.
  assert (P = (P = True)).
  - apply Prop_ex. split.
    + intros z. rewrite (Prop_ex P True); trivial. split.
      * trivial.
      * intros _. assumption.
    + intros []. assumption.
  - revert x y. rewrite H. apply K.
Qed.
```

Kod źródłowy 7.14: Dowód, że ekstensjonalność dowodów oraz aksjomat K implikuje irrelewancję

## 7.2 Wykorzystując `SProp`

`SProp` jest to uniwersum predykatów z definicyjną irrelewancją. Oznacza to że występuje w nim wbudowany aksjomat irrelewacji i wszystkie dowody tego samego typu można w nim przepisywać, bez dodatkowych założeń. Niestety jest to wciąż eksperymentalna funkcjonalność w Coqu i posiada bardzo ubogą bibliotekę standardową, która nie posiada nawet wbudowanej równości. Posiada natomiast kilka użytecznych konstrukcji:

`Box` - jest to rekord, który pozwala opakować dowolne wyrażenie w `SProp` i przenieść je do świata `Prop`, `Squash` - jest to typ induktywny, które jest indeksowany wyrażeniem w `Prop`. Pozwala na przeniesienie dowolnego predykatu do świata `SProp`, co z uwagi na mają ilość konstrukcji w bibliotece standardowej jest bardzo użyteczne,

`sEmpty` - jest to odpowiednik `False : Prop`. Posiada on regułę eliminacji z której wynika fałsz,

`sUnit` - jest to odpowiednik `True : Prop`. Rrównież pozwala się wydostać z świata `SProp`, za pomocą reguły eliminacji,

`Ssig` - jest to odpowiednik `sig`. Ponieważ w tym świecie występuje definicyjna irrelewancja to w bibliotece standardowej wraz z nim otrzymujemy twierdzenie `Spr1_inj`, które mówi o unikalności reprezentacji dla tego typu.

Aby pokazać, że używając podtypowania z `Ssig` również mamy jednego reprezentanta dla klasy abstrakcji musimy zdefiniować najpierw równość oraz nasz typ postaci normalnych.

```
Inductive Seq {A: Type} : A -> A -> SProp :=
| srefl : forall x: A, s_eq x x.
```

Kod źródłowy 7.15: Typ induktywny równości w `SProp`

```
Definition Squotient {A: Type} {f: A->A} (N: normalzation f) : Type :=
  Ssig (fun x : A => Seq x (f x)).
```

Kod źródłowy 7.16: Typ postaci normalnych w `SProp`

Mając już podstawowe definicje możemy przejść do właściwego dowodu.

```
Theorem only_one_repersentant {A: Type} (f: A -> A) (N: normalzation f)
  (q q': Squotient N) : norm_equiv f N (Spr1 q) (Spr1 q') -> Seq q q'.
Proof.
  intro H.
  destruct q, q'. cbn in *.
  assert (E: Seq Spr1 Spr0).
  - unfold norm_equiv in H. destruct Spr2, Spr3.
    subst. constructor.
  - destruct E. constructor.
Qed.
```

Kod źródłowy 7.17: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta w `Squotient`

Korzystanie z `SProp` niesie z sobą jednak poważny problem jakim jest próba powiedzenia predykatu w `Prop`. Gdybyśmy zamienili `Seq` na zwykłą równość (`=`), takim wypadku nie dałoby się udowadniać tego twierdzenia. Dowody w Coqu domyślnie są w uniwersum `Prop` i to w nim chcielibyśmy mieć dowody dla naszych typów ilorazowych. Z uniwersum `SProp` możemy się jedynie wydostać eliminując `sEmpty` lub `sUnit`, nie zawsze jednak da się to zrobić.

## 7.3 Homotopiczne podejście

Co jeśli jednak nie chcemy używać dodatkowych aksjomatów i pracować w `Prop`? W takiej sytuacji z ratunkiem przychodzi homotopiczna teoria typów. Jest to relatywnie nowa gałąź matematyki, która zajmuje się dowodami równości w różnych typach[**?**]. Homotopiczną interpretacją równości jest $\omega$-graf w którym punkty reprezentują elementy typów, a ścieżki dowody równości, ścieżki między ścieżkami dowody równości dowodów równości i tak dalej.

### 7.3.1 $N$-typy

Wprowadza ona różne poziomy uniwersów w których żyją typu w zależności od dowodów równości między nimi. Ich indeksowanie zaczynamy nie intuicyjnie od -2. Opiszmy istotne dla nas nas uniwersa:

Contr - jest to najniższe uniwersum, na poziomie minus dwa. Żyjące w nim typy mają dokładnie jeden element. Przykładem takiego typu jest `unit`.

```
Class isContr (A: Type) := ContrBuilder {
  center : A;
  contr  : forall x: A, x = center
}.
```

Kod źródłowy 7.18: Klasa typów żyjących w uniwersum Contr.

HProp - nie mylić z Coqowym `Prop`. Dla typów z tego uniwersum wszystkie elementy są sobie równe. Przykładem mieszkańca tego uniwersum jest `Empty`. Ponieważ nie ma on żadnych elementów, to w trywialny sposób wszystkie jego elementy są równe, ale brak elementów wyklucza bycie w Contr.

```
Class isHProp (P : Type) :=
  hProp : forall p q : P, p = q.
```

Kod źródłowy 7.19: Klasa typów żyjących w uniwersum HProp.

HSet - tu również nie ma związku z Coqowym `Set`. Jest to poziom zerowy hierarchii uniwersów. Typy żyjące w tym uniwersum charakteryzują się tym, że jeśli dwa elementy są sobie równe to istnieje tylko jeden dowód tego faktu. Można o tym myśleć jako, że dla tych typów prawdziwy jest aksjomat K. Przykładem takiego typu jest `bool`. Dlaczego jednak ma on unikatowe dowody równości powiemy później.

```
Class isHSet (X : Type) :=
  hSet : forall (x y : X) (p q : x = y), p = q.
```

Kod źródłowy 7.20: Klasa typów żyjących w uniwersum HSet.

```
Inductive universe_level : Type :=
| minus_two  : universe_level
| S_universe : universe_level -> universe_level.

Fixpoint isNType (n : universe_level) (A : Type) : Type :=
match n with
| minus_two => isContr A
| S_universe n' => forall x y: A, isNType n' (x = y)
end.
```

Kod źródłowy 7.21: Klasa typów żyjących w $n$-tym uniwersum.

Nie są to jedyne uniwersa. Na kolejnym poziomie typy, dla których dowody równości, między dowodami równości są zawsze tym samym dowodem i tak dalej i tak dalej. Definicję dowolnego uniwersum możemy przyjrzeć się w 7.21. Jak widzimy typy dowodów równości między elementami typu żyjącego w $(n+1)$-wszym uniwersum żyją na w $n$-tym uniwersum. Aby nabrać nieco więcej intuicji na temat poziomów uniwersów pozwolimy sobie na udowodnienie twierdzenia dotyczącego zawierania się uniwersów. Jak widzimy w twierdzeniu 7.23 każde

```
Lemma contr_bottom : forall A : Type, isContr A ->
  forall x y : A, isContr (x = y).

Theorem NType_inclusion : forall A: Type, forall n : universe_level,
  isNType n A -> isNType (S_universe n) A.
Proof.
  intros A n; revert A.
  induction n; intros A H.
  - cbn in *; intros x y.
    apply contr_bottom; assumption.
  - simpl in *; intros x y.
    apply IHn.
    apply H.
Qed.
```

Kod źródłowy 7.22: Dowód, że typu żyjące w $n$-tym uniwersum, żyją też w $(n+1)$-pierwszym uniwersum.

kolejne uniwersum zwiera w sobie poprzednie. Dowód twierdzenia `contr_bottom` pominiemy tutaj, lecz można się z nim zapoznać w dodatku HoTT.v. Wracając jednak to naszego podtypowania widzimy, że jak długo będziemy się zajmować typami, które żyją w uniwersum

HSet, nie będziemy musieli się martwić o dowody równości między elementami tego typu. A więc doskonale nadają się one do bycia pierwotnymi typami dla naszych typów ilorazowych. Pozostaje tylko ustalić które typy należą do tego uniwersum, tu również z pomocą przychodzi homotopiczna teoria typów oraz twierdzenie Hedberg'ga [?].

### 7.3.2 Typy z rozstrzygalną równością

Na początku warto definiować czym jest rozstrzygalna równość. Dla każdego typu z rozstrzygalną równością istnieje obliczalna (taka która można napisać w Coqu) funkcja która określa czy dwa elementy danego typu są tym samym elementem, czy też nie. Dobrym przykładem

```
Class Decidable (A : Type) :=
  dec : A + (A -> False).

Class DecidableEq (A : Type) :=
  dec_eq : forall x y: A, Decidable (x = y).
```

Kod źródłowy 7.23: Definicja rozstrzygalności, oraz rozstrzygalnej równości.

rozstrzygalnego typu jest `bool`. Natomiast rozstrzygalnej równości nie ma na przykład typ funkcji `nat -> nat`. Wspomniane wcześniej twierdzenie Hedberg'ga [?] mówi o tym, że każdy typ z rozstrzygalną równością żyje w uniwersum *HSet*. Dowód zaczniemy od zdefiniowania klasy typów sprowadzalnych. Pokażemy, że każdy typ rozstrzygalny jest sprowadzalny 7.25.

```
Class Collapsible (A : Type) :={
  collapse        : A -> A ;
  wconst_collapse : forall x y: A, collapse x = collapse y;
}.
```

Kod źródłowy 7.24: Definicja sprowadzalności.

Szczególnym przypadkiem są więc typy z rozstrzygalną równością, które mają sprowadzalne

```
Theorem dec_is_collaps : forall A : Type, Decidable A -> Collapsible A.
Proof.
  intros A eq. destruct eq.
  - exists (fun x => a). intros x y. reflexivity.
  - exists (fun x => x); intros x y.
    exfalso; apply f; assumption.
Qed.
```

Kod źródłowy 7.25: Dowód, że każdy typ rozstrzygalny jest sprowadzalny.

dowody równości (ścieżki) 7.26. Mając już zdefiniowane sprowadzalne ścieżki, potrzebujemy jeszcze szybkiego dowodu, na temat pętli ścieżek 7.27. Mając to już za sobą możemy przejść

```
Class PathCollapsible (A : Type) :=
  path_coll : forall (x y : A), Collapsible (x = y).

Theorem eq_dec_is_path_collaps : forall A : Type, DecidableEq A -> PathCollapsible A.
Proof.
  intros A dec x y. apply dec_is_collaps. apply dec.
Qed.
```

Kod źródłowy 7.26: Definicja wraz z dowodem, że każdy typ z rozstrzygalną równością ma sprowadzalne ścieżki.

```
Lemma loop_eq : forall A: Type, forall x y: A, forall p: x = y,
  eq_refl = eq_trans (eq_sym p) p.
Proof.
  intros A x y []. cbn. reflexivity.
Qed.
```

Kod źródłowy 7.27: Dowód, że każda pętla z ścieżek jest `eq_refl`.

```
Theorem path_collaps_is_hset (A : Type) : PathCollapsible A -> isHSet A.
Proof.
  unfold isHSet, PathCollapsible; intros C x y.
  cut (forall e: x=y, e = eq_trans (eq_sym(collapse(eq_refl x))) (collapse e)).
  - intros H p q.
    rewrite (H q), (H p), (wconst_collapse p q).
    reflexivity.
  - intros []. apply loop_eq.
Qed.
```

Kod źródłowy 7.28: Dowód, że każdy typ z sprowadzalnymi ścieżkami jest HSet'em.

do właściwego dowodu, że dowolny typ z sprawdzalnymi ścieżkami jest *HSet*'em 7.28. Jak więc widzimy każdy typ z rozstrzygalną równością ma tylko jeden dowód równości między dowolną parą równych sobie elementów. Oznacza to, że bez żadnych dodatkowych aksjomatów możemy udowodnić unikalność reprezentacji dla naszych typów ilorazowych, które mają rozstrzygalny typ pierwotny. Z uwagi na to, iż zdefiniowanie nie trywialnej funkcji normalizującej na typie bez rozstrzygalnej równości jest prawie nie możliwe, typy z rozstrzygalną równością wystarczą nam w tym rozdziale.

### 7.3.3 Równość między parami zależnymi

Podtypowanie w Coqu opiera się na parach zależnych, warto się przyjrzeć w jaki sposób wygląda równość między takimi parami. W przypadku zwykłych par sprawa jest prosta. Jeśli

```
Theorem pair_eq : forall (A B: Type) (a x : A) (b y : B),
  (a, b) = (x, y) -> a = x /\ b = y.
Proof.
  intros. inversion H. split; trivial.
Qed.
```

Kod źródłowy 7.29: Charakterystyka równości dla par.

jednak spróbujemy napisać to samo dla par zależnych otrzymamy błąd wynikający z niezgodności typów dowodów, nawet jeśli pozbędziemy się go ustalając wspólną pierwszą pozycję w parze to nie uda nam się udowodnić iż równość pary implikuje równość drugich elementów tych par, gdyż taka zależność implikowałaby aksjomat K [?]. Aby zrozumieć równość par zależnych musimy najpierw zdefiniować transport. Transport pozwala na przeniesienie typu

```
Definition transport {A: Type} {x y: A} {P: A -> Type} (path: x = y)
  (q : P x) : P y :=
match path with
| eq_refl => q
end.
```

Kod źródłowy 7.30: Definicja transportu.

`q : P x` wzdłuż ścieżki `path : x = y` do nowego typu `P y`. Pozwala on pozbyć się problemu niezgodności typów w charakterystyce równości na parach zależnych. Jak więc wynika z twierdzenia 8.1 równość par zależnych składa się z równości na pierwszych elementów, oraz na równości drugich elementów przetransportowanej wzdłuż pierwszej równości.

# 8 Przykłady typów ilorazowych zdefiniowanych przy użyciu podtypowania

Rozważywszy problemy wynikające z wykorzystania podtypowania do zdefiniowania typów ilorazowych w Coqu możemy przejść do przedstawienia kilku typów ilorazowych zaimple-

```
Theorem dep_pair_eq : forall (A: Type) (P: A->Type) (x y: A) (p: P x) (q: P y),
  existT P x p = existT P y q -> exists e: x = y, transport e p = q.
Proof.
  intros A P x y p q H. inversion H.
  exists eq_refl. cbn. trivial.
Qed.
```

Kod źródłowy 7.31: Charakterystyka równości dla par zależnych.

mentowanych w Coqu wykorzystując tą metodę. Wszystkie one posiadają rozstrzygalne typy pierwotne, a więc nie będziemy wykorzystywać dodatkowych aksjomatów.

## 8.1 Muti-zbiory dla typów z porządkiem liniowym

Jednym z wspominanych już zastosowań dla podtypowania jest stworzenie typu posortowanych list. Sortowanie listy zachowuje elementy, a więc posortowana lista jest permutacją listy pierwotnej. Dodatkowo możemy pokazać, że w przypadku list posortowanych porządkiem liniowym jeśli dwie listy są swoimi permutacjami, oraz obie są posortowane to oznacza, że są to te same listy. Jak można się spodziewać naszą funkcją normalizującą będzie funk-

```
Fixpoint count {A: Type} (p: A -> bool) (l: list A): nat :=
  match l with
  | nil => O
  | cons h t => if p h then S (count p t) else count p t
  end.

Definition permutation {A: Type} (a b : list A) :=
  forall p : A -> bool, count p a = count p b.
```

Kod źródłowy 8.1: Definicja permutacji dla list z rozstrzygalną równością.

cja sortująca listy. Jest ona idempotentna, czym spełnia nasze wymagania odnośnie funkcji normalizujących.