

# Funkcje normalizujące dla typów ilorazowych w Coqu

(Canonizing functions for quotient types in Coq)

Marek Bauer

Praca licencjacka

**Promotor:** dr Małgorzata Biernacka

Uniwersytet Wrocławski  
Wydział Matematyki i Informatyki  
Instytut Informatyki

27 lutego 2023



Streszczenie

...



...



# Spis treści

<b>1. Definicja problemu</b>	<b>9</b>
1.1. Cel pracy . . . . .	9
1.2. Czym jest Coq? . . . . .	9
1.3. Czym są typy ilorazowe . . . . .	10
1.3.1. Relacja równoważności . . . . .	10
1.3.2. Spojrzenie teorii typów . . . . .	11
1.4. Relacje równoważności generowane przez funkcję normalizującą . . . .	12
1.4.1. Funkcja normalizująca . . . . .	12
1.4.2. Przykłady funkcji normalizujących . . . . .	13
1.5. Typy ilorazowe nie posiadające funkcji normalizujących . . . . .	13
1.5.1. Para nieuporządkowana . . . . .	14
1.5.2. Liczby rzeczywiste definiowane za pomocą ciągów Cauchy'eago . . . .	14
1.5.3. Monada częściowych obliczeń . . . . .	14
<b>2. Pod-typowanie jako namiastka typów ilorazowych</b>	<b>17</b>
2.1. Czym jest podtypowanie . . . . .	17
2.1.1. Dlaczego w ogóle wspominamy o podtypowaniu? . . . . .	18
2.2. Dualna? Co to oznacza? . . . . .	18
2.2.1. Czym jest pushout? . . . . .	19
2.2.2. Przykład pushoutu w kategorii <i>Set</i> . . . . .	20
2.2.3. Sklejanie dwóch odcinków w okrąg, czyli pushout . . . . .	20
2.2.4. Czym jest pullback? . . . . .	21
2.2.5. Przykład pullbacku w kategorii <i>Set</i> . . . . .	21

2.2.6.	Pary liczb o tej samej parzystości, czyli pullback . . . . .	22
2.2.7.	Konkluzja . . . . .	22
2.3.	Unikatowość reprezentacji w podtypowaniu . . . . .	22
2.3.1.	Dodatkowe aksjomaty . . . . .	23
2.3.2.	Wykorzystując <b>SProp</b> . . . . .	27
2.3.3.	Homotopiczne podejście . . . . .	28
<b>3.</b>	<b>Typów ilorazowe z wykorzystujące pod-typowania</b>	<b>35</b>
3.1.	Pary nieuporządkowane . . . . .	35
3.1.1.	Relacja równoważności . . . . .	35
3.1.2.	Dowód jednoznaczności reprezentacji . . . . .	37
3.2.	Muti-zbiory skończone . . . . .	37
3.2.1.	Relacja równoważności . . . . .	37
3.2.2.	Funkcja normalizująca . . . . .	38
3.2.3.	Dowód jednoznaczności reprezentacji . . . . .	40
3.3.	Zbiory skończone . . . . .	40
3.3.1.	Relacja równoważności . . . . .	40
3.3.2.	Funkcja normalizująca . . . . .	41
3.3.3.	Dowód jednoznaczności reprezentacji . . . . .	42
<b>4.</b>	<b>Typy ilorazowe jako ślad funkcji normalizującej</b>	<b>43</b>
4.1.	Wolne monoidy . . . . .	43
4.1.1.	Czym jest wolny monoid? . . . . .	43
4.1.2.	Postać normalna wolnego monalidu, czyli lista . . . . .	44
4.2.	Liczby całkowite . . . . .	45
4.2.1.	Funkcja normalizująca dla liczb całkowitych . . . . .	46
4.2.2.	Indukcyjny typ liczb całkowitych z jednoznaczną reprezentacją	47
4.2.3.	Podstawowe operacje na jednoznacznych liczbach całkowitych	48
4.3.	Egzotyczne liczby całkowite . . . . .	50
4.3.1.	Inne spojrzenie na normalizację . . . . .	50

*SPIS TREŚCI*

7

**Bibliografia**

**51**





# Rozdział 1.

## Definicja problemu

### 1.1. Cel pracy

Celem pracy jest analiza możliwych sposobów definiowania typów ilorazowych w Coqu. Skupimy się jednak jedynie na typach ilorazowych, dla których istnieje funkcja normalizująca. Dodatkowo zostanie przedstawione dla których typów można zdefiniować taką funkcję, a dla których jest to nie możliwe. Koncentrować przy tym będziemy się na podejściu wykorzystującym zdefiniowany w bibliotece standardowej Coqa predykat równości, nie wykorzystując dodatkowych aksjomatów. Zostaną natomiast pokazane jakie możliwość dawałoby ich wykorzystanie w odpowiednich konstrukcjach. Gdyż język Coq nie posiada wsparcia dla typów ilorazowych będziemy musieli wykorzystać inne techniki pozwalające na pracę podobną do pracy z typami ilorazowymi. Głównym tematem zainteresowania tej pracy są typy indukcyjne które pozwalając na jednoznaczność reprezentację danego typu ilorazowego, oraz ich tworzenie na podstawie funkcji normalizujących tych że typów.

### 1.2. Czym jest Coq?

Coq jest darmowym asystentem dowodzenia na licencji GNU LGPL. Jego pierwsza wersja powstała w 1984 roku jako implementacja bazującego na teorii typów rachunku konstrukcji (Calculus of Constructions). Od 1991 roku Coq wykorzystuje bardziej zaawansowany rachunek indukcyjnych konstrukcji (Calculus of Inductive Constructions)[1], który pozwala zarówno na logikę wyższego rzędu, jak i na statycznie typowane funkcyjne programowanie, wszystko dzięki izomorfizmowi Curry’ego - Howarda [2]. Coq pozwala na proste obliczenia, lecz jest przygotowany pod ekstrakcję już gotowych programów do innych języków funkcyjnych, jakich jak OCaml. W Coqu każdy term ma swój typ, a każdy typ ma swoje uniwersum:

**Prop** - jest to uniwersum twierdzeń. Jest ono niepredykatatywne co pozwala na

tworzenie zdań, które coś mówią o innych zadaniach. To uniwersum jest usuwane podczas ekstrakcji kodu, przez co nie możliwym jest dopasowywanie do wzorca dowodów twierdzeń podczas konstrukcji typów, z wyjątkiem konstrukcji mających tylko jedno trywialne dopasowanie.

**SProp** - to samo co **Prop** zawierające jednak dodatkowo definicyjną irrelewantę dowodów, czyli wszystkie dowody tego samego twierdzenia są z definicji tym samym dowodem.

**Set** - jest to predykatywne uniwersum przeznaczone dla obliczeń. Jest ono zachowywane podczas ekstrakcji kodu.

**Type** - jest to nad uniwersum pozostałych, czyli **Prop** : **Type** itp. Tak naprawdę uniwersów **Type** jest nieskończenie wiele, gdyż każde uniwersum nie może zawierać same siebie, przez co **Type**(**i**) : **Type** (**i**+1).

Coq pozwala jedynie na definiowanie funkcji które terminują, co czyni z niego język nie równoważny maszynie Turinga, jednak w dalszej części pracy pokażemy jak modelować nie terminujące obliczenia w Coqu. Główną zaletą Coqa jest jednak możliwość pisania dowodów (jak i programów) za pomocą gotowych taktyk, oraz możliwość interaktywnego przyglądania się które części dowodu wymagają jeszcze udowodnienia. Pozwala to na dużo łatwiejsze dowodzenie niż w językach takich jak Idris czy Agda.

### 1.3. Czym są typy ilorazowe

W algebrze 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 modułarną arytmetyką. Każdy z nas zaznajomiony z zasadami działania zegara i nikogo nie dziwni że po godzinie dwunastej następuje godzina pierwsza. O godzinach na tracy 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.

$$\dots \equiv -11 \equiv 1 \equiv 13 \equiv 25 \equiv 37 \equiv \dots \pmod{12} \quad (1.1)$$

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

#### 1.3.1. 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 ( $a \sim a$ )

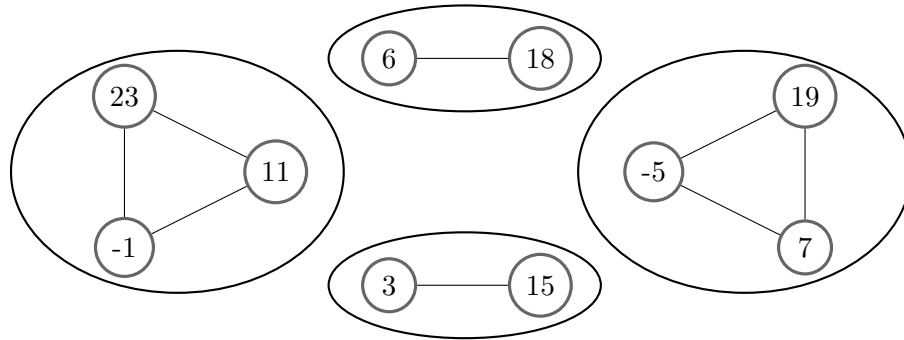
**Symetryczność** - jeśli  $a$  jest w relacji z  $b$  ( $a \sim b$ ) to również relacja zachodzi w odwrotnej kolejności ( $b \sim a$ ).

**Przechodność** - 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  $c$  ( $a \sim c$ ).

```
Class equivalence_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 1.3..1: Klasa relacji równoważności zapisana w Coq

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 ( $\sim$ ).



Rysunek 1.1: 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 tę relację równoważności

### 1.3.2. Spojrzenie teorii typów

Jako że praca skupia się na implementacji typów ilorazowych w Coqu, stąd też bardziej skupimy się na spojrzeniu teorii typów na ilorazy, w przeciwieństwie do spojrzenia teorii mnogościowego. W teorii typów  $T/\sim$  będziemy nazywać typem ilorazowym generowanym przez relację równoważności ( $\sim$ ) na typie pierwotnym  $T$ .

Będziemy oznaczać  $a = b$  w typie  $T/\sim$  wtedy i tylko wtedy gdy  $a \sim b$ . Widzimy zatem iż każdy element typu  $T$  jest również elementem typu  $T/\sim$ . Natomiast nie wszystkie funkcje z typu  $T$  są dobrze zdefiniowanymi funkcjami z typu  $T/\sim$ . Funkcja  $f : T \rightarrow X$  jest dobrze zdefiniowaną funkcją  $f : (T/\sim) \rightarrow X$  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  $x = y \Rightarrow f(x) = f(y)$ .

## 1.4. Relacje równoważności generowane przez funkcję normalizującą

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

$$\forall x, y \in T, x \sim_h y \iff h(x) = h(y) \quad (1.2)$$

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

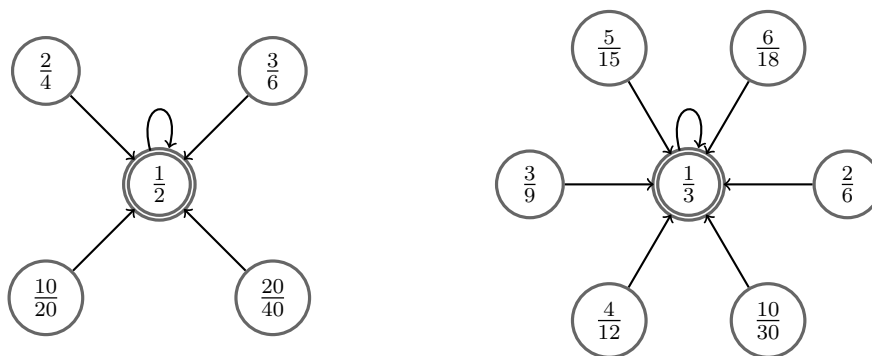
### 1.4.1. Funkcja normalizująca

W tej pracy skupimy się na szczególnym przypadku funkcji  $h : T \rightarrow B$  w którym  $B = T$ . Dodatkowo będziemy wymagać aby funkcja  $h$  była idempotentna. Tak zdefiniowaną funkcję  $h$  będziemy nazywać funkcją normalizującą. Zbudujemy jesz-

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

Kod źródłowy 1.4..1: Klasa funkcji normalizujących

cze odrobinę intuicji wokół tak zdefiniowanych funkcji normalizujących. Jak wiemy relacja równoważności łączy 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 1.2: Przykład działania funkcji normalizującej dla reprezentacji liczby wymiernych w postaci  $\mathbb{Z} \times \mathbb{N}$

### 1.4.2. Przykłady funkcji normalizujących

Wykorzystajmy tutaj już wcześniej przytoczony przykład liczb wymiernych. 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 postaci 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  $m - 1$ .

## 1.5. Typy ilorazowe nie posiadające funkcji normalizujących

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. Problem jest jednak to, iż nie każdy typ ilorazowy posiada taką funkcję. 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 możemy skorzystać z niego w Coqowym rachunku indykatywnych konstrukcji. Zobaczmy zatem jakie typy ilorazowe nie są definiowalne w ten sposób

### 1.5.1. Para nieuporządkowana

Jest to najprostszy typ którego nie można zdefiniować w ten sposób. W żaden sposób nie można określić czy para  $\{\square, \circ\}$  jest w postaci normalnej, a może  $\{\circ, \square\}$ . Oczywiście mając parę wartości na której istnieje pewien porządek zupełny możemy w zbudować parę wykorzystując go. Niestety w ogólności jest to niemożliwe. Niestety wraz z tym idzie iż zbiory, jak i multi-zbiory również nie mają w ogólności swoich postaci kanonicznych, a więc nie można ich zdefiniować dla typów nieposiadających w tym przypadku porządku liniowego.

### 1.5.2. Liczby rzeczywiste definiowane za pomocą ciągów Cauchy’ego

Kolejnym typem który w spodziewany sposób nie da się znormalizować są liczby rzeczywiste. W Coqu najlepiej je zdefiniować za pomocą ciągów Cauchy’ego liczb wymiernych.

$$\text{isCauchy}(f) := \forall m, n \in \mathbb{N}^+, n < m \rightarrow |f_n - f_m| < \frac{1}{n} \quad (1.3)$$

Granice ciągu będą wyznaczać, którą liczbę rzeczywistą reprezentuje dany ciąg. Chcielibyśmy zatem, aby ciągi o tej samej granicy były z sobą w relacji równoważności. Wyznaczenie granicy ciągu mając jedynie czarną skrzynkę, która może jedynie wyliczać kolejne jej elementy jest niemożliwe. Załóżmy, że jednak jest możliwe, niech  $s_k$  będzie ciągiem 1 do  $k$ -tego miejsca a później stałym ciągiem 0, natomiast  $s_\infty$  ciągiem samych 1. Problem sprawdzenia równości tych ciągów jest przeliczanie rekurencyjny (ciąg  $s_k$  może reprezentować zatrzymanie działania programu w  $k$ -tej sekundzie). Oba ciągi mają różne granice, a co za tym idzie powinny mieć różne postacie normalne, a więc istnieje jakiś element na którym się różnią. Gdyby istniała zatem taka obliczalna i całkowita funkcja normalizująca moglibyśmy za jej pomocą rozwiązać problem stopu w rekurencyjny sposób, sprawdzając jedną wartość ciągu w postaci normalnej. Zatem nie może istnieć taka funkcja.

### 1.5.3. Monada częściowych obliczeń

Kolejnym typem ilorazowym, dla którego nie może istnieć funkcja normalizująca jest typ częściowych obliczeń. Jak widzimy w definicji 1.5..1 jest to typ coinduktywny produkujący albo wynik działania funkcji, albo dalsze obliczenia, które mogą, ale nie muszą się kiedyś skończyć. W relacji równoważności chcielibyśmy aby były wszystkie

```
CoInductive delayed (A : Type) := Delayed {
  state : A + delayed A
}.
```

Kod źródłowy 1.5..1: Definicja monady częściowych obliczeń w Coqu.

obliczenia które ostatecznie zwrócą ten sam wynik, naturalnie będzie istnieć osobna klasa abstrakcji dla obliczeń, które nigdy się nie skończą. Podobnie jednak jak w przypadku powyżej pomimo iż możemy dla każdej klasy abstrakcji wyznaczyć jej reprezentanta w łatwy sposób (obliczenie które natychmiast zwraca wynik), to nie jest to funkcja całkowita rekurencyjna. Możemy zauważyć że za pomocą monady częściowych obliczeń możemy reprezentować dowolne obliczenia wykonywane na maszynie Turinga. Gdyby zatem potrafiliśmy w skończonym czasie wyznaczyć reprezentanta dla każdych obliczeń moglibyśmy w skończonym czasie stwierdzić czy maszyna Turinga się kiedyś zatrzyma, czy też nie. Oznaczało by to rozwiązanie problemu stopu, który jak wiem jest problemem nie rekurencyjnym.





## Rozdział 2.

# Pod-typowanie jako namiastka typów ilorazowych

### 2.1. 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 2.1..1 wymaga wykorzystania typów za-

```
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 2.1..1: Dwie równoważne definicje podtypowania w Coqu.

leżnych, 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 konstrukcję, gdzie

jednak  $(P : A \rightarrow \text{Prop})$  zostało zamienione na  $(P : A \rightarrow \text{Type})$  jest to uogólniona definicja podtypowania, nazywana sigma typem. Jest to para zależna w której

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

Kod źródłowy 2.1..2: Definicja definicja sigma typu z biblioteki standardowej Coq.

drugi element pary zależy do pierwszego. Posiada ona również zdefiniowaną notację  $\{a : A \ \& \ P\}$ , nawiązuje ona do tego  $a$  jest zarówno typu  $A$  jak i  $P$ .

### 2.1.1. 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 różnych 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 Coq. 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

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

Kod źródłowy 2.1..3: Definicja podtypu kanonicznych postaci względem funkcji normalizującej  $f$ .

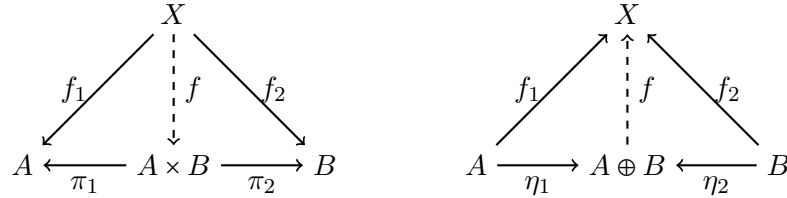
narzędzi, które dostarcza nam Coq.

## 2.2. 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 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 konstrukcją dualną do niej  $\sigma^{\text{op}}$  definiujemy poprzez:

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

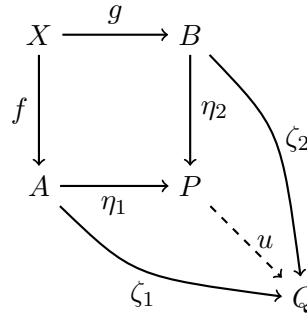
Mając to już za sobą możemy powiedzieć prostym językiem, że dualność polega na zmianie kierunku strzałek w naszej konstrukcji. Znając to pojęcie możemy zadać



Rysunek 2.1: Przykład dwóch dualnych konstrukcji. Po lewej stronie widzimy produkt, a po prawej co-produkt. Oba diagramy komutują.

sobie pytanie gdzie w ilorazach i podtypowaniu występują jakieś strzałki, których kierunki mielibyśmy zamieniać? Występują mianowicie odpowiednio w pushoutach oraz pullbackach.

### 2.2.1. Czym jest pushout?



Rysunek 2.2: Diagram definiujący pushout  $P$ . Diagram komutuje.

Na rysunku 2.2 możemy zobaczyć diagram definiujący pojęcie pushoutu. Widzimy, że powstaje on z pewnych dwóch morfizmów  $f : X \rightarrow A$  oraz  $g : X \rightarrow B$ . 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 tę własność. Najlepiej 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 morfizmów  $f : X \rightarrow A$  oraz  $g : X \rightarrow B$  istnieje pushout, jeśli jednak istnieje to jest unikatowy z dokładnością do unikatowego izomorfizmu.

### 2.2.2. 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 2.2 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 komutowanie 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 2.2 będzie komutował. Widzimy zatem, że pushout rzeczywiście ma jakiś związek z typami ilorazowymi, w których to  $X$  definiuje które elementy zostaną z sobą utożsamione.

### 2.2.3. Sklejanie dwóch odcinków w okrąg, czyli pushout

Rozważyliśmy bardziej przyziemny, lecz dość abstrakcyjny przykład w kategorii *Set*. Skonstruujmy nieco bardziej wizualny przykład, czyli w naszym przypadku okrąg na rysunku 2.3. Upewnijmy się, iż rzeczywiście  $C$  jest topologicznym okręgiem.

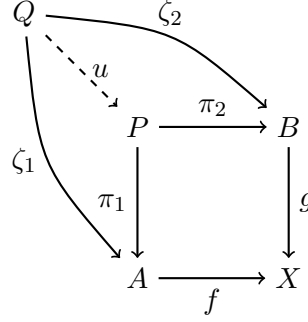
$$\begin{array}{ccc}
 \{0, 1\} & \xrightarrow{id} & [0, 1] \\
 id \downarrow & & \downarrow \eta_2 \\
 [0, 1] & \xrightarrow{\eta_1} & C
 \end{array}$$

Rysunek 2.3: Diagram definiujący okrąg  $C$  używając pushoutu

Wiemy, że aby diagram 2.3 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 wzdłuż kuli  $(n - 1)$ -wymiarowej otrzymamy kulę  $n$ -wymiarową.

### 2.2.4. 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 2.4. Każdy pullback jest definiowany za pomocą dwóch



Rysunek 2.4: Diagram definiujący pullback  $P$ . Diagram komutuje.

morfizmów  $f : A \rightarrow X$  oraz  $g : B \rightarrow X$ , 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 zewnętrzna 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  $f : A \rightarrow X$  oraz  $g : B \rightarrow X$  istnieje pushout, jeśli jednak istnieje to jest unikatowy z dokładnością do unikatowego izomorfizmu.

### 2.2.5. 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 2.4 struktura  $A, B$  oraz  $P$  przypomina coś na kształt produktu. Zaczniemy 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 warunku 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 2.4 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 komutowanie zewnętrznej części diagramu. Możemy na w tym miejscu zauważyć iż rzeczywiście pullback ma związek z podtypowaniem, poprzez wyrzucanie elementów które nie spełniają równości  $f(a) = g(b)$ .

### 2.2.6. 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 diagramie 2.3 widzimy definicję pullbacku  $P$  za pomocą morfizmu

$$\begin{array}{ccc} P & \xrightarrow{\pi_2} & \mathbb{Z} \\ \pi_1 \downarrow & & \downarrow f \\ \mathbb{Z} & \xrightarrow{f} & \{0, 1\} \end{array}$$

Rysunek 2.5: Diagram definiujący pary liczb całkowitych o tej samej parzystości  $P$  używając pullbacku

$f : \mathbb{Z} \rightarrow \{0, 1\}$ , 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 modulo 2, czyli mówiąc inaczej mają tę samą parzystość.

### 2.2.7. 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 2.2 oraz 2.4, to możemy mówić o tych pojęciach jako dualnych do siebie nawzajem.

## 2.3. Unikatowość reprezentacji w podtypowaniu

Podtypowanie w Coqu nie dostarcza nam jednak niestety tak przyjemnego interfejsu, jak moglibyśmy się spodziewać po matematycznym podejściu do tego conceptu. 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`, gdzie `sig` jest parą zależną w związku z tym, aby skonstruować element tego typu potrzebujemy dwóch składników, wartości oraz dowodu, że ta wartość 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 2.3..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 2.3..3

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

Kod źródłowy 2.3..2: Twierdzenie mówiące o unikalności reprezentacji w podtypowaniu

nie można udowodnić w Coq bez dodatkowych aksjomatów, natomiast wersja tego twierdzenia dla `sigT` jest po prostu fałszywa. Pozostaje nam zatem zredukować oczekiwania, lub dodać dodatkowe założenia.

### 2.3.1. Dodatkowe aksjomaty

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

#### 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

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

Kod źródłowy 2.3..3: Definicja irrelewancji w Coqu

trudu możemy udowodnić twierdzenie 2.3..3. Dodatkowo możemy udowodnić, że nasze unikatowość reprezentacji jest tak naprawę równoważna aksjomatowi irrelewancji dowodów.

```

Theorem irrelevance_uniques : 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 2.3.4: Dowód unikalności reprezentacji używając irrelewancji w Coq

```

Theorem uniques_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 2.3.5: Dowód, że unikalności reprezentacji implikuje irrelewancję w Coq



### Aksjomat K

Aksjomat ten został wymyślony przez Habil Streicher w swojej pracy "Investigations Into Intensional Type Theory" [5]. My posłużymy się jego nieco zmodyfikowaną wersją (UIP - uniqueness of identity proofs), która lepiej oddaje konsekwencje jego użycia. Jest to nieco słabsza wersja aksjomatu irrelevancji, która mówi jedynie o

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

Kod źródłowy 2.3..6: Aksjomat K w Coq

irrelevancji dowodów równości. Ma on pewną ciekawą konsekwencję, którą została opisana w [5], a mianowicie pozwala on na zanurzenie równości na parach zależnych w zwykłą równość. Dowód tego twierdzenia pominiemy, lecz można go znaleźć

**Theorem** `sig_injectivity : 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 2.3..7: Twierdzenie o zanurzeniu równości na parach zależnych w Coq

w dodatku Stricher.v. Aksjomat K nie jest równoważny aksjomatowi irrelevancji [?] 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 2.3..8: Definicja podtypu postaci kanonicznych generowanych przez funkcję normalizującą f, oraz projekcji dla niego w Coq

tego typu można z łatwością udowodnić wykorzystując aksjomat K.

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 1.3..1 `norm_equiv`. Mając już definicję jak wygląda ta relacja równoważności możemy przejść do właściwego dowodu.

```
Theorem unqiues_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 2.3..9: Dowód unikalności reprezentacji dla podtypu postaci kanonicznych generowanych przez funkcję normalizującą  $f$  w Coq

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

```
Theorem norm_equiv_is_equivalence_relation (A: Type) (f: A -> A)
  (N:normalizing_function f) : equivalence_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 2.3..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 2.3..11: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta używając aksjomatu  $K$

### Związek między tymi aksjomatami

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

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

Kod źródłowy 2.3..12: Predykat ekstensjonalności dowodów

**Theorem** `Irrelevance_K : Irrelevance -> K.`

**Proof.**

```
intros Irr A x y. apply Irr.
Qed.
```

Kod źródłowy 2.3..13: Dowód że irrelevancja 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 2.3..14: Dowód, że ekstensjonalność dowodów oraz aksjomat K implikuje irrelevancję

### 2.3.2. Wykorzystując `SProp`

`SProp` jest to uniwersum predykatów z definicyjną irrelevancją. Oznacza to że występuje w nim wbudowany aksjomat irrelevancji 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**. Również pozwala się wydostać z świata **SProp**, za pomocą reguły eliminacji,

**Ssig** - jest to odpowiednik **sig**. Ponieważ w tym świecie występuje definicyjna irrelevancja 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 2.3..15: Typ induktywny równości w **SProp**

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

Kod źródłowy 2.3..16: Typ postaci normalnych w **SProp**

Mając już podstawowe definicje możemy przejść do właściwego dowodu.

Korzystanie z **SProp** niesie z sobą jednak poważny problem jakim jest próba przeniesienia predykatu do **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**, co nie gwarantuje możliwości wyprowadzenia analogicznego dowodu w **Prop**.

### 2.3.3. 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

```

Theorem only_one_representant {A: Type} (f: A -> A) (N: normalization 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 2.3..17: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta w `Squotient`

typach[4]. Homotopieczną 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.

### *N*-typy

Wprowadza ona różne poziomy uniwersów w których żyją typy w zależności od dowodów równości między nimi. Ich indeksowanie zaczynamy nie intuicyjnie od -2. Opiszmy istotne dla 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 2.3..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 2.3..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 2.3..20: Klasa typów żyjących w uniwersum HSet.

Nie są to jedyne uniwersa. Na kolejnym poziomie żyją 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 2.3..21. Jak wi-

```
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 2.3..21: Klasa typów żyjących w  $n$ -tym uniwersum.

dzimy 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 2.3..23 każde kolejne uniwersum zawiera 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'a [3].

```

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 2.3..22: Dowód, że typu żyjące w  $n$ -tym uniwersum, żyją też w  $(n+1)$ -pierwszym uniwersum.

### 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 rozstrzygalnego typu jest `bool`. Natomiast

```

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

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

```

Kod źródłowy 2.3..23: Definicja rozstrzygalności, oraz rozstrzygalnej równości.

rozstrzygalnej równości nie ma na przykład typ funkcji `nat -> nat`. Wspomniane wcześniej twierdzenie Hedberg’ga [3] 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 2.3..25. Szczególnym przypadkiem są więc typy z rozstrzygalną równością, które mają sprowadzalne dowody równości (ścieżki) 2.3..26. Mając już zdefiniowane sprowadzalne ścieżki, potrzebujemy jeszcze szybkiego dowodu, na temat pętli ścieżek 2.3..27. Mając to już za sobą możemy przejść do właściwego dowodu, że dowolny typ z sprowadzalnymi ścieżkami jest *HSet*’em 2.3..28. Jak więc widzimy każdy typ z rozstrzygalną równością ma tylko jeden dowód równości między dowolną parą równych sobie ele-

```

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

```

Kod źródłowy 2.3..24: Definicja sprowadzalności.

```

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.
    exfalse; apply f; assumption.
Qed.

```

Kod źródłowy 2.3..25: Dowód, że każdy typ rozstrzygalny jest sprowadzalny.

```

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 2.3..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 2.3..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 2.3..28: Dowód, że każdy typ z sprowadzalnymi ścieżkami jest HSet'em.

mentów. 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.

### 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 jednak spróbujemy napisać to samo dla par zależnych otrzymamy

**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 2.3..29: Charakterystyka równości dla par.

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 [5]. Aby zrozumieć równość par zależnych musimy najpierw zdefiniować transport. Transport pozwala na przeniesienie typu  $q : P \ x$  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 2.3..31 równość par zależnych składa się z równości na pierwszych elementach, oraz na równości drugich elementów przetransportowanej wzdłuż pierwszej równości.

```

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 2.3..30: Definicja transportu.

```

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 2.3..31: Charakterystyka równości dla par zależnych.

## Rozdział 3.

# Typów ilorazowe z wykorzystujące pod-typowania

W poprzedni rozdziale poznaliśmy czym jest pod-typowanie, oraz w jaki sposób możemy je wykorzystać w implementacji typów którymi możemy się posługiwać prawie jak typami ilorazowymi. W tym natomiast poznamy jakie typy ilorazowe możemy zaimplementować w ten sposób. Skupimy się tutaj na typach z uniwersum zerowego, w związku z tym będziemy pracować z domyślną równością oraz w uniwersum [Prop](#).

### 3.1. Pary nieuporządkowane

Wcześniej wspomnieliśmy, że nie da się zdefiniować pary nieuporządkowanej dla dowolnego typu bazowego, gdyż nie wiadomo który element powinien być pierwszy. jako ciekawostka homotopicznej teorii typów taką parę można zdefiniować jako:

$$\sum_{(X:Type)} \sum_{(H:\|X \approx \text{bool}\|)} A^X \quad (3.1)$$

Natomiast taka definicja również nie jest możliwa do zaimplementowania w Coqu z względu na brak obciążenia oraz na brak możliwości rozbijania dowodów z [SProp](#) w uniwersum [Type](#). Organicznym się zatem jedynie do typów bazowych z rozstrzygalnym pełnym porządkiem na nich 3.1..1, słaba asymetria jest potrzebna do udowodnienia unikalności, a pełność do zbudowania pary dla każdych dwóch elementów. Takie pary zdefiniujemy jak trójka elementów: pierwszy element pary, drugi element pary, oraz dowód dobrego posortowania 3.1..2.

#### 3.1.1. Relacja równoważności

Będziemy chcieli utożsamić ze sobą takie pary które zawierają takie same dwa elementy, a więc są w relacji `sim` 3.1..3.

```

Class FullOrd (A: Type) := {
  ord  : A -> A -> bool;
  asym : forall x y: A, ord x y = true -> x = y;
  full : forall x y: A, ord x y = true \/ ord y x = true;
}.

```

Kod źródłowy 3.1..1: Definicja rozstrzygalnego pełnego porządku dla typu A w Coqu.

```

Record UPair (A: Type) `{FullOrd A} := {
  first  : A;
  second : A;
  sorted : ord first second = true;
}.

```

Kod źródłowy 3.1..2: Definicja pary nieuporządkowanej w Coqu.

```

Definition contains {A: Type} `{FullOrd A} (x y: A) (p: UPair A) :=
  (first p = x /\ second p = y) \/ (first p = y /\ second p = x).

```

```

Definition sim {A: Type} `{FullOrd A} (p q: UPair A) :=
  forall x y: A, contains x y p <-> contains x y q.

```

Kod źródłowy 3.1..3: Relacja równoważności par nieuporządkowanych.

### 3.1.2. Dowód jednoznaczności reprezentacji

Mając te definicje możemy w łatwy sposób udowodnić iż rzeczywiście jeśli dwie pary nieuporządkowane zawierają takie same elementy to są sobie równe w Coqu. Oczywiście będziemy musieli tu skorzystać z dowodu `bool_is_hset`, ale wiedząc że `bool` w trywialny sposób ma rozstrzygalną równość oraz wiedzę z poprzedniego rozdziału, w łatwy sposób można udowodnić, że żyje on rzeczywiście w uniwersum `HSet`. Można by sądzić iż wymaganie istnienia rozstrzygalnego pełnego porządku na

**Theorem** `UPair_uniq (A: Type) {FullOrd A} (p q : UPair A) (x y : A) :`  
`contains x y p -> contains x y q -> p = q.`

**Proof.**

```
intros c1 c2. case_eq (ord x y); intro o;
destruct p, q; unfold contains in *; cbn in *;
destruct c1, c2, H0, H1; subst;
try assert (x = y) by (apply asym; assumption);
subst; try f_equal; try apply bool_is_hset.
```

**Qed.**

Kod źródłowy 3.1.4: Dowód, że pary uporządkowane zawierające te same elementy są tą samą parą.

typie bazowym znacząco ogranicza użyteczność takiej konstrukcji. Oczywiście nie będzie można zdefiniować pary dowolnych funkcji, lecz w praktycznych zastosowaniach programistycznych obiekty są reprezentowane za pomocą ciągów bitów, na których w prosty sposób można zdefiniować rozstrzygalny pełny porządek.

## 3.2. Muti-zbiory skończone

Kolejnym klasycznym przykładem typu ilorazowego, których chcielibyśmy mieć w Coqu jest multi-zbiór skończony. Jest to struktura danych która może przechowywać wiele tych samych elementów, natomiast kolejności nie ma w niej znaczenia. Można o nich myśleć jako nieuporządkowanych listach. Tak jak one są również monadami. Mogą one zostać wykorzystane na przykład do sprawdzenia czy różne procesy produkują takie same elementy. Podobnie jak w przypadku pary nieuporządkowanej, tu również będziemy musieli dodać pewnie ograniczenia na typ bazowy w postaci rozstrzygalnego porządku liniowego 3.2.1.

### 3.2.1. Relacja równoważności

Chcemy aby równoważne sobie zbiory miały takie same elementy. Inaczej aby mówiąc dwa multi-zbiory były sobie równoważne muszą być swoimi własnymi permu-

```

Class LinearOrder {A: Type} := {
  ord      : A -> A -> bool;
  anti_sym : forall x y: A, ord x y = true -> ord y x = true -> x = y;
  trans    : forall x y z: A, ord x y = true -> ord y z = true -> ord x z = true;
  full     : forall x y: A, ord x y = true \/ ord y x = true;
}.

```

Kod źródłowy 3.2..1: Definicja rozstrzygalnego porządku liniowego dla typu A w Coqu.

tacjami. Postulujemy w tym miejscu użycie nieco sprytniejszej definicji permutacji, która jest równoważna dla typów z rozstrzygalną równością 3.2..2. Nie wymaga ona definiowania wszystkich praw permutacji, które później mogą być trudne w użyciu, natomiast opiera się na idei iż dwie listy które są permutacjami muszą zawierać takie same elementy z dokładnością do ich ilości. A takie sformułowanie jest równoważne temu, że dla każdego predykatu liczba elementów na obu listach jest taka sama (w szczególności dla predykatu bycia jakimś konkretnym elementem typu). Jedyne problem z tą definicją jest fakt iż nie działa ona dla typów z nierozstrzygalną równością, ale rozstrzygalny linowy porządek implikuje rozstrzygalną równość na typie. Dowód równoważności tych definicji można znaleźć w `permutations.v`.

```

Fixpoint count {A: Type} (p: A -> bool) (l: list A): nat :=
  match l with
  | nil => 0
  | 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 3.2..2: Definicja permutacji dla list z rozstrzygalną równością.

### 3.2.2. Funkcja normalizująca

Nie trudno zgadnąć jaka funkcja, jaką idempotentną funkcję możemy użyć do wyznaczenia postaci normalnej dla list, takiej że wszystkie permutacje są rzutowane na tę samą listę. Oczywiście wyborem na taką funkcję jest oczywiście funkcja sortująca. Postuluję tutaj użycie funkcji sortującej wykorzystującą drzewo, które przechwytuje elementy tylko w liściach 3.2..3. To podejście pozwala nam na łatwe dzielenie listy na dwie równe części, a sama sortowanie przez scalania łatwo zaimplementować w funkcyjnym języku programowania. Oczywiście dowolna inna funkcja sortująca zadziałałaby równie dobrze.

```

Inductive BT(A : Type) : Type :=
  | leaf : A -> (BT A)
  | node : (BT A)->(BT A)->(BT A).

Fixpoint BTInsert{A : Type}(x : A)(tree : BT A) :=
  match tree with
  | leaf y => node (leaf x)(leaf y)
  | node l r => node r (BTInsert x l)
  end.

Fixpoint listToBT{A : Type}(x : A)(list : list A): BT A :=
  match list with
  | nil => leaf x
  | cons y list' => BTInsert x (listToBT y list')
  end.

Fixpoint merge{A : Type}(ord : A -> A -> bool)(l1 : list A): (list A) -> list A :=
  match l1 with
  | [] => fun (l2 : list A) => l2
  | h1::t1 => fix anc (l2 : list A) : list A :=
    match l2 with
    | [] => l1
    | h2::t2 => if ord h1 h2
      then h1::(merge ord t1) l2
      else h2::anc t2
    end
  end.

Fixpoint BTSort {A : Type}(ord : A -> A -> bool)(t : BT A): list A :=
  match t with
  | leaf x => [x]
  | node l r => merge ord (BTSort ord l) (BTSort ord r)
  end.

Definition mergeSort{A: Type}(ord : A -> A -> bool)(l: list A): list A :=
  match l with
  | [] => []
  | x::l' => BTSort ord (listToBT x l')
  end.

```

Kod źródłowy 3.2.3: Sortowanie przez scalanie z wykorzystaniem drzewa przechowującym wartości w liściach.

### 3.2.3. Dowód jednoznaczności reprezentacji

Każda funkcja sortująca musi spełniać dwa kryteria. Po pierwsze wynik jej działania musi być listą posortowaną, a po drugie wynik musi być permutacją listy wejściowej. Drugie kryterium sprawia że sortownie nie utożsamia ze sobą list które nie były ze sobą w relacji permutacji. Pierwsze kryterium gwarantuje nam natomiast impotencję sortowania, gdyż jak wiemy, sortowanie nie zmienia już posortowanej listy. Dowód tego faktu jak i tego, że zaprezentowana powyżej funkcja `mergeSort` 3.2..3 rzeczywiście sortuje można znaleźć w dodatku `sorted_lists.v`.

## 3.3. Zbiory skończone

Kolejną użytecznym typem ilorazowym są zbiory skończone. Różnią się one od zdefiniowanych powyżej multi-zbiorów tym, że każdy element znajduje się na nich co najwyżej raz. A więc dodanie elementu do zbioru który już się w nim znajduje daje identyczny zbiór jak przed tą operacją. Podobnie jak w przypadku multi-zbiorów tu również będziemy wymagać aby typ bazowy miał porządek liniowy 3.2..1.

### 3.3.1. Relacja równoważności

W przypadku zbiorów chcemy utożsamiać ze sobą listy które zawierają takie same elementy, nie patrząc na ich liczbę. Możemy zatem nieco zmodyfikować definicję permutacji ze multi-zbiorów tak, żeby rozróżniała jedynie czy wynik jest zerowy czy nie 3.3..1. Wystarczy zatem zamienić funkcję zliczającą na funkcję która sprawdza czy na liście istnieje element spełniający predykat.

```
Fixpoint any {A: Type} (p : A -> bool) (l: list A) : bool :=
  match l with
  | [] => false
  | (x::l') => if p x then true else any p l'
  end.
```

```
Definition Elem_eq {A: Type} (l l' : list A) : Prop :=
  forall p : A -> bool, any p l = any p l'.
```

Kod źródłowy 3.3..1: Definicja relacji zawierania tych samych elementów przez dwie listy w Coq.



### 3.3.2. Funkcja normalizująca

W przypadku zbiorów do normalizacji listy będziemy potrzebować funkcji pseudo sortującej, która usuwa kolejne wystąpienia tego samego elementu. Podobnie jak w przypadku sortowania tu również zalecamy użycie funkcji opartej na drzewie, natomiast tym razem na klasycznym drzewie binarnym. Funkcja `DSort` ?? działa poprzez wkładanie kolejnych elementów do drzewa binarnego, pomijając element jeśli taki znajduje się już w drzewie, a na końcu spłaszcza drzewo i tworzy z niego listę.

```
Inductive tree (A: Type) : Type :=
| leaf : tree A
| node : A -> tree A -> tree A -> tree A.
```

```
Definition comp {A: Type} `{LinearOrder A} (x y: A) :=
  if ord x y then (if ord y x then Eq else Gt) else Lt.
```

```
Fixpoint add_tree {A: Type} `{LinearOrder A} (x: A) (t : tree A) : tree A :=
match t with
| leaf => node x leaf leaf
| node v l r => match comp x v with
                  | Lt => node v (add_tree x l) r
                  | Eq => node v l r
                  | Gt => node v l (add_tree x r)
                end
end.
```

```
Fixpoint to_tree {A: Type} `{LinearOrder A} (l : list A) : tree A :=
  match l with
  | [] => leaf
  | (x::l') => add_tree x (to_tree l')
  end.
```

```
Fixpoint to_list {A: Type} (l : tree A) : list A :=
  match l with
  | leaf => []
  | node x l r => to_list l ++ [x] ++ to_list r
  end.
```

```
Definition DSort {A: Type} `{LinearOrder A} (l : list A) : list A := to_list (to_tree l)
```

Kod źródłowy 3.3..2: Definicja funkcji sortująco deduplikującej w Coq.

### 3.3.3. Dowód jednoznaczności reprezentacji

Podobnie jak w przypadku sortowania każda funkcja sortująco deduplikująca powinna spełniać dwa kryteria. Po pierwsze wynik takiej funkcji powinien być listą która nie zawiera powtarzających się elementów a po drugie elementy te są posortowane zgodnie z porządkiem liniowym. Wiemy, że dwie zdeduplikowane listy, które zawierają takie same elementy są swoimi permutacjami. Dodatkowo wiemy, że dla każdej permutacji listy istnieje dokładnie jedna taka która jest posortowana zgodnie z porządkiem liniowym. Wynika z tego że funkcja sortująco deduplikująca jest idempotentna. Po drugie lista wejściowa i wyjściowa powinna być w relacji zawierania tych samych elementów, co sprawia że listy zawierające różne elementy nie zostaną ze sobą utożsamione. Dowód faktu, że nasza funkcja `DSort 3.3..2` spełnia kryteria funkcji sortująco deduplikującej możemy znaleźć w dodatku `Deduplicated.v`.

## Rozdział 4.

# Typy ilorazowe jako ślad funkcji normalizującej

W tym rozdziale omówimy jakie jakie typy ilorazowe można zdefiniować za pomocą szeroko pojętego śladu funkcji normalizującej. Zostaną przedstawione przykłady typów indukcyjnych, których konstrukcja jest silnie inspirowana procesem ich normalizacji. Część z nich to będą już znane od dłuższego czasu typy, a inne to wymyślone na potrzeby tej pracy przykłady o różnym poziomie użyteczności.

### 4.1. Wolne monoidy

Osoby zajmujące się programowaniem funkcyjnym czasem nazywają listy wolnymi monoidami. Nie jest to jednak oczywiste dla każdego czym są wolne monoidy a tym bardziej skąd wzięła się ta analogia.

#### 4.1.1. Czym jest wolny monoid?

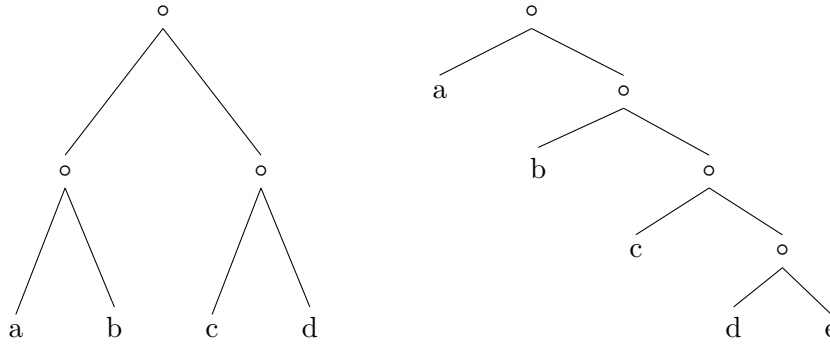
Monoid jest strukturą algebraiczną  $(\mathbf{A}, \circ)$ , gdzie  $\mathbf{A}$  będziemy nazywać nośnikiem struktury, a  $\circ$  jest działaniem w tej strukturze. O nośniku struktury mogą być na przykład liczby naturalne, lub jakiegokolwiek inne obiekty na których możemy wykonywać działania. Działanie na strukturze jest natomiast funkcją binarną działającą na nośniku  $\circ : \mathbf{A} \rightarrow \mathbf{A} \rightarrow \mathbf{A}$ . Każdy monoid musi spełniać następujące prawa:

**Element neutralny:**  $\exists e \in \mathbf{A}, \forall x \in \mathbf{A}, e \circ x = x = x \circ e$

**Łączność działania:**  $\forall xyz \in \mathbf{A}, x \circ (y \circ z) = (x \circ y) \circ z$

Dobrym przykładem struktury która jest monoidem są macierze z operacją mnożenia, lub w mniej oczywisty sposób funkcje jednoargumentowe z operacją ich składania. Wolność struktury algebraicznej natomiast objawia się w tym, że ograni-

czymy się jedynie do do praw wynikających z jej własności, natomiast nie będziemy uwzględniać praw wynikających z jej nośnika. Zatem dla wolnego monoidu z liczbami naturalnymi jako nośnikiem nie będziemy utożsamiać z sobą wyrażeń 2 oraz  $1 + 1$ , gdyż nie wynikają one z praw monoidu. Natomiast wyrażenie takie jak  $(1 + 2) + 3$ , będą tym samym co  $1 + (2 + 3)$ , gdyż łączność działania jest jednym z praw monoidu. Zatem dla dowolnego nośnika możemy myśleć o wolnym monoidzie jako drzewie operacji.



Rysunek 4.1: Dwa równoważne drzewa dla wyrażenia  $a \circ b \circ c \circ d$

#### 4.1.2. Postać normalna wolnego monoidu, czyli lista

Jak możemy zobaczyć na rysunku 4.1 to samo wyrażenie możemy zapisać na wiele równoważnych sposobów w wolnym monoidzie, wynika to z faktu, iż nawiasy nie mają znaczenia ze względu na łączność działania, a ponad to można było dopisać dowolną ilość elementów neutralnych, które nie zmieniają wartości wyrażenia. W naiwny sposób moglibyśmy zatem zaimplementować wolny monoid korzystając z 3 konstruktorów 4.1.1: elementu neutralnego, dowolnej wartości, oraz łączącego ich operatora. Taka naiwna definicja, tak jak nasze drzewa 4.1, powoduje problem

```
Inductive FreeMonoid (A: Type) :=
| leaf : FreeMonoid A
| var   : A -> FreeMonoid A
| op    : FreeMonoid A -> FreeMonoid A -> FreeMonoid A.
```

Kod źródłowy 4.1.1: Naiwna definicja wolnego monoidu w Coqu.

niejednoznaczności reprezentacji. Aby rozwiązać ten problem musimy ustalić jaką postać tego wyrażenia będziemy traktować jako normalną. Zastosujemy tutaj klasyczne rozwiązanie, w którym zawsze występuje dokładnie jeden element neutralny na końcu oraz operatory wiążące mocniej w lewo:

$$(a_1 \circ (a_2 \circ (a_3 \circ \dots \circ (a_n \circ e) \dots)))$$

Wprawne oko już powinno dostrzec w wyrażeniu powyżej znaną każdemu induktywną definicję listy 4.1..2, gdzie elementem neutralnym jest `nil`, a kolejne elementy są połączone konstruktorem `cons`. Nasza funkcja normalizująca powinna zatem prze-

```
Inductive list (A: Type) :=
| nil   : list A
| cons  : A -> list A -> list A.
```

Kod źródłowy 4.1..2: Definicja listy z biblioteki standardowej Coq.

chodzić po wszystkich wierzchołkach drzewa `FreeMonoid4.1..1` od lewej do prawej i przekształcać drzewo do ustalonej postaci normalnej. Ślad tej funkcji w postaci listy napotkanych elementów z pominięciem tych neutralnych jest typem ilorazowym dla wolnych monoidów. Możemy w łatwy sposób zdefiniować również funkcję która przekształci wolny monoid od razu do jego normalnej postaci w formie listy 4.1..3.

```
Fixpoint free_to_list {A: Type} (m: FreeMonoid A) : list A :=
match m with
| leaf   => []
| var x   => [x]
| op x y => to_list x ++ to_list y
end.
```

Kod źródłowy 4.1..3: Definicja funkcji normalizującej wolny monoid do postaci list w Coqu.

## 4.2. Liczby całkowite

Klasyczny przykładem wykorzystania typów ilorazowych są liczby całkowite. Są one naturalnym rozszerzeniem liczb naturalnych do grupy addytywnej, a więc takie w której każdy element ma swój element przeciwny. Możemy je zaimplementować w naiwny sposób za pomocą pary liczb naturalnych 4.2..1. Liczba po prawej stronie będzie reprezentować z ilu poprzedników, a liczba po prawej stronie z ilu następników składa się definiowana liczba. Taka reprezentacja jest bardzo wygodna

```
Definition Int : Type := nat * nat.
```

Kod źródłowy 4.2..1: Naiwna reprezentacja liczb całkowitych w Coqu.

w implementacji, takich operacji jak suma 4.2..2, następnik, czy poprzednik. Wynika to z faktu, że możemy w tym celu wykorzystać już istniejące operacje na liczbach naturalnych. Niestety taka definicja pozostawia problem niejednoznaczności reprezentacji. Żeby się na niego natknąć wystarczy porównać wyniki  $1 + (-1) = 0$  oraz

```

Definition int_add (n: Int) (m: Int) : Int :=
  let (a, b) := n in let (c, d) := m in (a + c, b + d).

```

Kod źródłowy 4.2..2: Dodawanie naiwnie zdefiniowanych liczb całkowitych w Coqu.

$2 + (-2) = 0$ . Pomimo iż oba te wyrażenia mają ten sam wynik to w naiwnej postaci Int4.2..1, będą one reprezentowane jako odpowiednio  $(1, 1)$  oraz  $(2, 2)$ .

#### 4.2.1. Funkcja normalizująca dla liczb całkowitych

Aby pozbyć się problemu niejednoznaczności będziemy musieli wyznaczyć postać normalną dla liczb całkowitych. W naszym przypadku za normalną uznamy postać w której przynajmniej jednym z elementów pary jest liczba zero. Funkcją normalizującą będziemy nazywać taką, która będzie odejmować jedynkę z lewej i prawej strony, tak długo, aż nie będzie to już możliwe 4.2..3.

```

Function int_norm' (x y : nat) : (nat * nat) :=
  match x, y with
  | S x', S y' => int_norm' x' y'
  | _, _      => (x, y)
end.

```

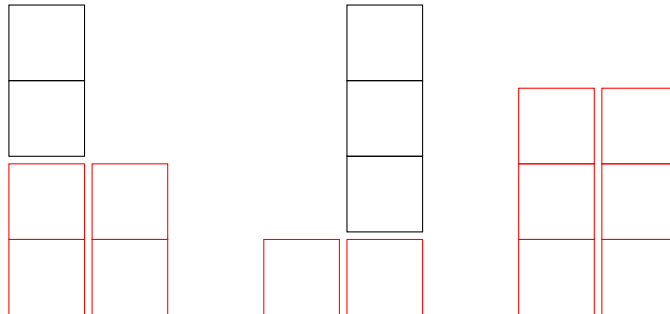
```

Function int_norm (p: nat * nat) : (nat * nat) :=
  let (x, y) := p in int_norm' x y.

```

Kod źródłowy 4.2..3: Definicja funkcji normalizującej liczby całkowite w Coqu.

Po zakończeniu działania tej funkcji pozycja na której pozostanie liczba niezerowa będzie wyznaczała znak liczby całkowitej, natomiast jej wartość będzie wyznaczać wartość bezwzględną całej liczby całkowitej.



Rysunek 4.2: Wizualizacja działania funkcji normalizującej dla liczb całkowitych (odpowiednio  $-2, 3$  i  $0$ ), na czerwono zaznaczone są elementy które zostaną usunięte przez działanie tej funkcji.

### 4.2.2. Indukcyjny typ liczb całkowitych z jednoznaczną reprezentacją

Mając już do dyspozycji funkcję normalizującą możemy wykorzystać ideę jej działania do stworzenia typu indukcyjnego, który będzie charakteryzował się jednoznacznością reprezentacji. Sam ślad funkcji nie jest szczególnie ciekawy w tym przypadku. Możemy jednak dużo nauczyć się z ostatniego kroku procesu normalizacji. W kodzie 4.2..3 jest on uproszczony do `_`, `_`, jednak pod tym wzorcem możemy wyróżnić trzy przypadki:

`S x'`, `0` - gdy wynikiem jest liczba ujemna o wartości `S x'`,

`0`, `S y'` - gdy wynikiem jest liczba dodatnia o wartości `S y'`,

`0`, `0` - gdy wynikiem jest zero.

Możemy zatem stworzyć typ indukcyjny z konstruktorami dla każdego z tych trzech przypadków.

```
Inductive Z : Type :=
| Pos  : nat -> Z
| Zero : Z
| Neg  : nat -> Z.
```

Kod źródłowy 4.2..4: Definicja typu liczb całkowitych z jednoznaczną reprezentacją w Coqu.

Możemy w bardzo łatwy sposób zmodyfikować funkcję normalizującą 4.2..3, a taką która przekształca naszą naiwną reprezentację na nową jednoznaczną.

```
Function int_to_Z (x y : nat) : Z :=
match x, y with
| S x', S y' => norm x' y'
| S x', 0    => Pos x'
| 0, S y'    => Neg y'
| 0, 0       => Zero
end.
```

Kod źródłowy 4.2..5: Definicja funkcji przekształcającej naiwną reprezentację w jednoznaczną Coqu.

Taka definicja nie pozostawia wątpliwości co do swojej jednoznaczności, o ile oczywiście funkcja normalizująca jest poprawna.

### 4.2.3. Podstawowe operacje na jednoznacznych liczbach całkowitych

Możemy teraz przejść do zdefiniowania kilku przykładowych funkcji dla liczb całkowitych, na zdefiniowanym powyżej typie  $\mathbb{Z}$  4.2.4. Tak jak w przypadku liczb naturalnych warto zacząć od rozpoczęcia od zdefiniowania następnika, a z racji posiadania liczb ujemnych również poprzednika.

```
Definition succ (n: Z) : Z :=
match n with
| Pos k => Pos (S k)
| Zero => Pos 0
| Neg 0 => Zero
| Neg (S n) => Neg n
end.
```

Kod źródłowy 4.2.6: Definicja następnika dla liczb całkowitych  $\mathbb{Z}$  4.2.4.

```
Definition pred (n: Z) : Z :=
match n with
| Pos (S n) => Pos n
| Pos 0 => Zero
| Zero => Neg 0
| Neg n => Neg (S n)
end.
```

Kod źródłowy 4.2.7: Definicja poprzednika dla liczb całkowitych  $\mathbb{Z}$  4.2.4.

```
Definition neg (n: Z) : Z :=
match n with
| Pos k => Neg k
| Zero => Zero
| Neg k => Pos k
end.
```

Kod źródłowy 4.2.8: Definicja poprzednika dla liczb całkowitych  $\mathbb{Z}$  4.2.4.

W obu przypadkach definicja jest dość prosta i nie pozostawia za dużo wątpliwości co do swojej poprawności. W obu przypadkach konieczny był specjalny przypadek, który odpowiada za przejście do zero. Definicja negacji jest wyjątkowo trywialna w przypadku tej reprezentacji, więc nie wymaga omówienia. Przejdziemy zatem to najważniejszej operacji na liczbach całkowitych, jaką jest dodawanie.



```

Fixpoint map_n {A: Type} (n: nat) (f: A -> A) (x: A) : A :=
match n with
| 0    => x
| S n' => f (map_n n' f x)
end.

Definition add (a b : Z) : Z :=
match a with
| Pos n => map_n (S n) succ b
| Zero  => b
| Neg n => map_n (S n) pred b
end.

```

Kod źródłowy 4.2..9: Definicja dodawania dla liczb całkowitych Z 4.2..4.

W przedstawionej w tej pracy definicji dodawania 4.2..9 została wykorzystana funkcja pomocnicza `map_n` 4.2..9. Pozwala ona na nałożenie  $n$  operacji na daną wartość. Z uwagi iż nasze liczby całkowite wykorzystują liczby naturalne oczywistym jest zdefiniowanie dodawania jako wykonanie  $n$  operacji następnika, lub poprzednika jeśli liczba którą dodajemy była ujemna. Odejmowanie można w łatwy sposób zdefiniować jako dodawanie liczby przeciwnej. Możemy zatem zdefiniować odejmowanie, lecz równie ważną operację mnożenia.

```

Definition mul (a b: Z) : Z :=
match a with
| Pos n => map_n (S n) (add b) Zero
| Zero  => Zero
| Neg n => neg (map_n (S n) (add b) Zero)
end.

```

Kod źródłowy 4.2..10: Definicja mnożenia dla liczb całkowitych Z 4.2..4.

Definicja mnożenia wykorzystuje podobny mechanizm jak dodawanie, z tą różnicą, że tym razem nie dodajemy jedynki od wyniku, a całą liczbę przez którą mnożymy. Jest to dość znana rekurencyjna definicja mnożenia, więc nie będziemy poświęcać zbyt dużo czasu na nią. Wszystkie przedstawione powyżej operacje i nie tylko, wraz z dowodami ich podstawowymi prawami, takimi jak łączność, przemienność, oraz rozdzielnosc mnożenia względem dodawania można znaleźć w dodatku `better_integer.v`.

### 4.3. Egzotyczne liczby całkowite

Poprzednia definicja liczb całkowitych wydaje się być naturalnym kandydatem, z uwagi na powiązanie z funkcją normalizującą, a dodatkowo pozwala na łatwe obliczenia. Większość konkurencyjnych definicji jest bardzo podobna, czasami zamiast symetrycznych trzech konstruktorów wykorzystują one dwa i jeden z nich jest obciążony zerem. Niesmak może jednak pozostawiać fakt, iż wykorzystują one w swojej definicji liczby naturalne. Nasuwa się zatem pytanie czy można zdefiniować liczby całkowite bez odwoływania się do liczb naturalnych?

#### 4.3.1. Inne spojrzenie na normalizację

# Bibliografia

- [1] Calculus of inductive constructions.
- [2] P. D. Groote. *The Curry-Howard Isomorphism*. Academia, 1995.
- [3] M. HEDBERG. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.
- [4] N. Institute for Advanced Study (Princeton, U. F. Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations Program, 2013.
- [5] T. Streicher. Investigations into intensional type theory. *Habilitation Thesis, Ludwig Maximilian Universität*, 1993.