Funkcje normalizujące dla typów ilorazowych w Coqu

(Canonizing functions for quotints types in Coq)

Marek Bauer

Praca magisterska

Promotor: dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30 marca 2023

Streszczenie

. . .

. . .

Spis treści

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 indykatywne które pozwalając na jednoznaczną 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 wykorzytsuje 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 [6]. 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 zadnich. 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 irrelewancję dowodów, czyli wszystkie dowody tego samego twierdzenie 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 udowodnienie. 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 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.

$$\cdots \equiv -11 \equiv 1 \equiv 13 \equiv 25 \equiv 37 \equiv \ldots \pmod{12} \tag{1.1}$$

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 ($a \sim a$)

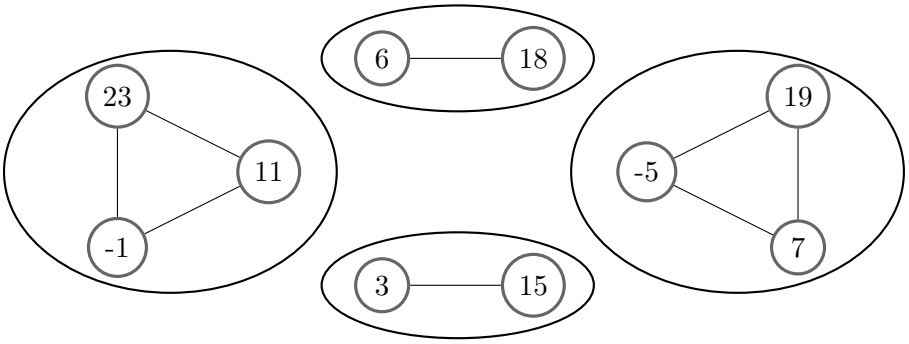
Symetryczność - jeśli a jest w relacji z b ($a \sim b$) to również relacja zachodzi w odwrotnej kolejności ($b \sim a$).

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 ($a \sim c$).

```
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 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 tą 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 teorio 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 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 $x = y \Rightarrow f(x) = f(y)$.

1.4. Relacje równoważności generowane przez funkcję normalizującą

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

$$\forall x, y \in T, x \sim_h y \iff h(x) = h(y) \tag{1.2}$$

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 $f = g \circ h$.

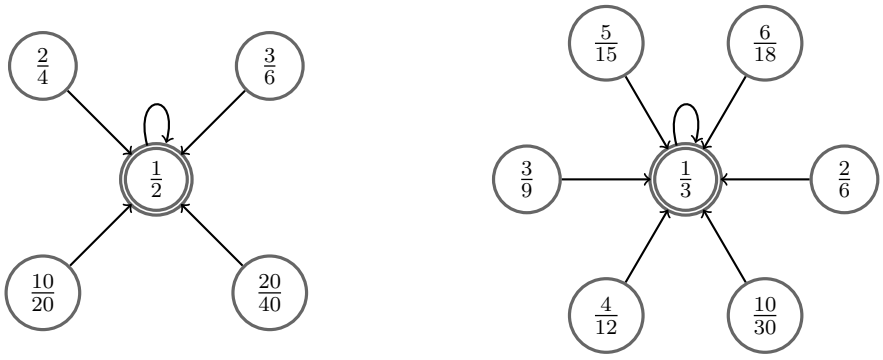
1.4.1. Funkcja normalizująca

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 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 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 zbór 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 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 $m - 1$.

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 $\{\square, \bigcirc\}$ jest w postaci normalnej, a może $\{\bigcirc, \square\}$. 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'eago

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

$$\textbf{isCauchy}(f) \coloneqq \forall m, n \in \mathbb{N}^+, n < m \to |f_n - f_m| < \frac{1}{n} \tag{1.3}$$

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_∞ 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 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

```
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 Coqa.

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 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

```
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.

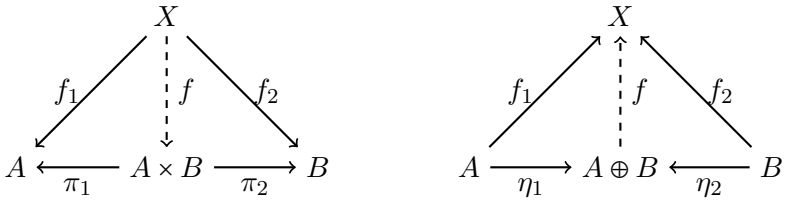
narzędzi, które dostarcza nam Coq.

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 σ jest konstrukcją w teorii kategorii to konstrukcję dualną do niej σ^{op} definiujemy poprzez:

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

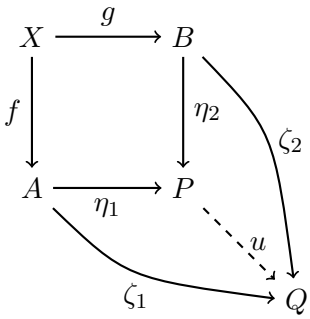
Mając to już za sobą możemy powiedzieć prostym językiem, że dualność polega na zmianie kierunku strzałek w naszej konstrukcji. Znając to pojęcie możemy zadać



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 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.

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 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 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 dojść 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.

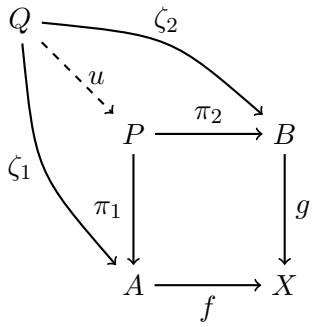
$$\begin{array}{ccc} \{0,1\} & \xrightarrow{id} & [0,1] \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle \eta_2} \\ [0,1] & \xrightarrow[\eta_1]{} & C \end{array}$$

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 w 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.

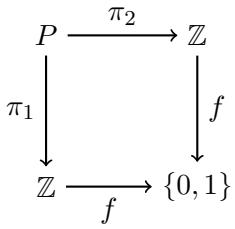
morfizmów $f : A \to X$ oraz $g : B \to X$, które tworzą strukturę przypominającą coprodukt. Z komutacji wiemy, że $f \circ \pi_1 = g \circ \pi_2$. Podobnie jak w przypadku pushoutu, tu również aby P był pullbackiem to musi być najlepszym obiektem, dla którego diagram komutuje. Oznacza to że dla każdego 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.

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. 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 ζ_1 oraz ζ_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 π_1 i π_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 $f(a) = g(b)$.

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



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

$f : \mathbb{Z} \to \{0,1\}$, 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.

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 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`, gdzie `sig` jest parą zależną 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 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 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 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 irrelewacji dowodów.

```
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 2.3..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 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"[11]. My posłużymy się jego nieco zmodyfikowaną wersją (UIP - uniqueness of identity proofs), która lepiej oddaje konsekwencje jego użycia. Jest to nieco słabsza wersja aksjomatu irrelewancji, która mówi jedynie

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

Kod źródłowy 2.3..6: Aksjomat K w Coq

o irrelewanjci dowodów równości. Ma on pewną ciekawą konsekwencję, którą została opisana w [11], a mianowicie pozwala on na zanurzenie równości na parach zależnych w zwykłą równość Dowód tego twierdzenia pominiemy, lecz można go znaleźć

```
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 2.3..7: Twierdzenie o zanurzenie równości na parach zależnych w Coq

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 2.3..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.

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 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 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_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 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 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 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 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 2.3..14: Dowód, że ekstensjonalność dowodów oraz aksjomat K implikuje irrelewancję

2.3.2. 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 2.3..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 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_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 2.3..17: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta w `Squotient`

typach[8]. Homotopiczną interpretacją równości jest ω-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 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 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 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].

```
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 [7] 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 sprawdzalnymi 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.
    exfalso; 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 [11]. 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 elementów, 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:

$$\sum_{(X:Type)} \sum_{(H:\|X\simeq \mathsf{bool}\|)} A^X \tag{3.1}$$

Natomiast taka definicja również nie jest możliwa do zaimplementowania w Coqu z względu na brak obcięcia 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 pray, 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 sprawdzenie czy różne procesy produkują takie same elementy. Podobnie jak w przypadku pary nieuporządkowanej, tu również będziemy 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 definiowana 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). Jedyny problem z tą definicją jest fakt iż nie działa ona dla typów z nierozstrzygalna 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 tą samą listę. Oczywistym wyborem na taką funkcję jest oczywiście funkcja sortująca. Postuluję tutaj użycie funkcji sortującej wykorzystującą drzewo, które przechowuje 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 utożsami 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 utożsami 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 funkcje 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 tworzony 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_tre
```

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 funckji powinnien 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 zawiarające 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 programowanie 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?

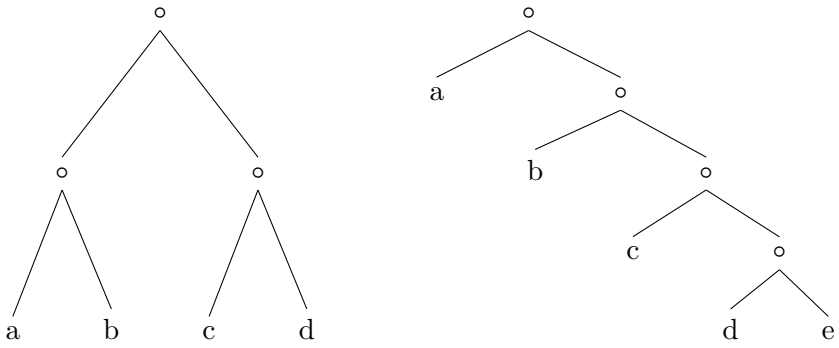
Monoid jest strukturą algebraiczną $(\mathbf{A}, \circ)$, gdzie $\mathbf{A}$ będziemy nazywać nośnikiem struktury, a $\circ$ jest działaniem w tej strukturze. O nośnikiem struktury mogą być na przykład liczby naturalne, lub jakiekolwiek inne obiekty na których możemy wykonywać działania. Działanie na strukturze jest natomiast funkcją binarną działającą na nośniku $\circ : \mathbf{A} \to \mathbf{A} \to \mathbf{A}$. 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 monoidtem 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 monalidu, 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:

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

Wprawne oko już powinno dostrzec w wyrażeniu powyżej znaną każdemu induktywną definicję listy4.1..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 Coqa.

chodzić po wszystkich wierzchołkach drzewa `FreeMonoid`4.1..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 suma4.2..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 `Int`4.2..1, będą one reprezentowane jako odpowiednio $(1, 1)$ oraz $(2, 2)$.

4.2.1. Funkcja normalizująca dla liczb całkowitych

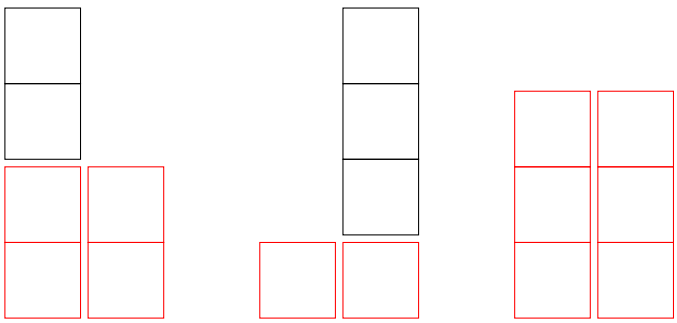
Aby pozbyć się problemu niejednoznaczności będziemy musieli wyznaczyć postać normalną dla liczb całkowitych. W naszym przypadku za normalną uznamy postać w której przynajmniej jednym z elementów pary jest liczba zero. Funkcją normalizującą będziemy nazywać taką, która będzie odejmować jedynkę z lewej 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 idę 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', O` - gdy wynikiem jest liczba ujemna o wartości `S x'`,

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

`O, O` - 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, na 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', O    => Neg x'
| O   , S y' => Pos y'
| O   , O    => 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 `Z` 4.2..4. Tak jak w przypadku liczb naturalnych warto zacząć od rozpocząć 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 `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 `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 `Z` 4.2..4.

W obu przypadkach definicja jest dojść prosta i nie pozostawia za dużo wątpliwości co do swojej poprawności. W obu przypadkach konieczny był specjalny 4 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
| O    => 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 zdefiować jako dodawane liczby przeciwnej. Możemy zatem zdefiniować ostania, 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.

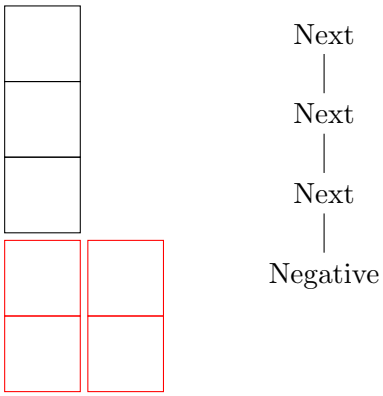
Definicja mnożenia wykorzystuje podobny mechanizm jak dodawanie, z tą różnicą, że tym razem nie dodajemy jedynki od wyniku, a całą liczbę przez którą mnożymy. Jest to dojść znana rekurencyjna definicja mnożenia, więc nie będziemy poświęcać zbyt dużo czasu na nią. Wszystkie przedstawione powyżej operacji i nie tylko, wraz z dowodami ich podstawowymi prawami, takimi jak łączność, przemienność, oraz rozdzielność 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

W poprzedniej sekcji patrzyliśmy na proces normalizacji "od dołu do góry", który polegał na usuwaniu następników z obu elementów pary, aż jedna będzie zerem. Możemy jednak odwrócić ten proces tworząc pewien proces pseudo-normalizacji, skupiając się na spojrzeniu "od góry do dołu". W tym procesie będziemy zliczać ile elementów jest powyżej, nie wiedząc jednak czy zliczamy następnik, czy poprzedniki. Dopiero po osiągnięciu punktu, w którym obie wartości są takie same, będziemy w stanie zweryfikować, czy liczba jest dodatnia czy ujemna.



Rysunek 4.3: Wizualizacja działania pseudo-normalizacji "od góry do dołu".

4.3.2. Następniko-poprzednik jako ślad pseudo-normalizacji

Możemy teraz stworzyć typ iniektywny bazujący na śladzie zaprezentowanej pseudo-normalizacji. Żeby jednak był on jednoznaczny poprzebujemy pozbyć się problemu zera. W zaprezentowanej powyżej intuicji nie wiadomo w jaki sposób powinno być one reprezentowane, gdyż sytuacja w której zaczynamy od równych wartości żadna wartość nie jest większa, a więc nie wiadomo którego konstruktora użyć. Ten problem rozwiążemy zaburzając symetrię między konstruktorami. Jeden z nich będzie reprezentować zero, i w jego kontekście `Next` będzie następnikiem, a więc

`Next Zero` będzie reprezentować jedynkę. Drugi natomiast będzie minus jedynką, a `Next` zaaplikowany na nim będzie oznaczać poprzednik.

```
Inductive Z' : Type :=
| Zero     : Z'
| MinusOne : Z'
| Next     : Z' -> Z'.
```

Kod źródłowy 4.3..1: Definicja liczby całkowitych z następniko-poprzednikiem.

4.3.3. Operacje na liczbach z następniko-poprzednikiem

Konstrukcja liczb z następniko-poprzednikiem pozwala na jednoznaczną reprezentację, nie jest jednak zbyt wygodna do przeprowadzania obliczeń. Podstawowe operacje takie jak następnik, czy poprzednik mają liniową złożoność obliczeniową względem wielkości liczby. Wszystko za sprawą faktu, że aby wiedzieć czy powinniśmy dodać następniko-poprzednik czy go usunąć musimy wiedzieć czym on tak naprawdę jest, a więc musimy sprawdzić ostatni konstruktor.

```
Function succ (k : Z') : Z' :=
match k with
| Zero => Next Zero
| MinusOne => Zero
| Next Zero => Next (Next Zero)
| Next MinusOne => MinusOne
| Next k' => Next (succ k')
end.
```

Kod źródłowy 4.3..2: Definicja następnika liczb całkowitych z następniko-poprzednikiem 4.3..1.

```
Function pred (k : Z') : Z' :=
match k with
| Zero => MinusOne
| MinusOne => Next MinusOne
| Next Zero => Zero
| Next MinusOne => Next (Next MinusOne)
| Next k' => Next (pred k')
end.
```

Kod źródłowy 4.3..3: Definicja poprzednika liczb całkowitych z następniko-poprzednikiem 4.3..1.

Oczywiście bardziej zaawansowane operacje jak dodawanie czy mnożenie można zdefiniować, lecz definicje te nie są szczególnie przyjemne. Pominiemy je zatem w tej pracy. Definicję dodawania wraz z dowodem na izomorfizm pomiędzy tą definicją liczb całkowitych, a tą nieco bardziej klasyczną zaprezentowaną w poprzedniej sekcji możemy znaleźć w dodatku integer_izo.v.

4.4. Dodatnie liczby wymierne

W przeciwieństwie do liczb całkowitych znalezienie jednoznacznej reprezentacji dla liczb wymiernych nie jest prostym zadanie. Ta sekcja jest operta na pracy Yves Bertot'a [4] w której przedstawił koncept reprezentowania liczb wymiernych za pomocą śladu algorytmu Euklidesa.

4.4.1. Normalizacja liczb wymiernych

Korzystając z podstawowej wiedzy na temat ułamków wiemy, że istnieje nieskończenie wiele reprezentacji tego samego ułamka. Zapisy takie jak $\frac{1}{2}$, $\frac{2}{4}$, czy $\frac{50}{25}$ wyrażają taką samą wartość: pół. Za postać normalną z reguły uznaje taką formę, w której nie można dokonać już operacji skrócenia ułamka, czyli podzielenia jego licznika i mianownika przez tą samą liczbę. Aby poznać przez jaką liczbę należy podzielić obie strony ułamka, aby uzyskać postać nieskracalną należy wyliczyć największy wspólny dzielnik. Jednym ze sposobów na jego wyznaczenie, jest właśnie postępowanie zgodnie z algorytmem Euklidesa 4.4..1. Algorytm ten bazuje na pra-

```
Function euclid (p q: nat) : nat :=
match compare p q with
| Eq => p
| Gt => euclid (p - q) q n'
| Lt => euclid p (q - p) n'
end.
```

Kod źródłowy 4.4..1: Definicja klasycznego algorytmu Euklidesa w pseudo Coqu.

wie $\mathrm{NWD}(a,b) = \mathrm{NWD}(a-b,b)$, jeśli $a > b$. Inną ciekawą obserwacją z której skorzystamy, jest to że ślad algorytmu nie zmieni się, jeśli przemnożymy obie liczby przez dowolną dodatnią liczbę naturalną, w przykładzie poniżej k.

$$\begin{aligned} 11k &> 5k \\ (11-5)k = 6k &> 5k \\ (6-5)k = k &< 4k \\ k < 3k &= (4-1)k \\ k < 2k &= (3-1)k \\ k = k &= (2-1)k \end{aligned} \tag{4.1}$$

Jak widzimy w przykładzie powyżej ślad, czyli to która liczba okazała się większa nie zależy od wybranego k. Jedyne co się zmieniło to wyliczony największy wspólny dzielnik został przemnożony przez stałą k. Dodatkowo znając ślad możemy odtworzyć argumenty, z dokładnością do największego wspólnego dzielnika. Możemy zatem użyć tych śladów do jednoznacznego reprezentowania liczb wymiernych. Dowód tego faktu, jak i innych związanych z śladami algorytmów Euklidesa możemy znaleźć w dodatku Qplus.v.

4.4.2. Typ liczb wymiernych dodatnich

Zdefiniujmy zatem typ śladów algorytmu Euklidesa 4.4..2. Jak mogliśmy się

```
Inductive Qplus : Type :=
| One : Qplus
| N   : Qplus -> Qplus
| D   : Qplus -> Qplus.
```

Kod źródłowy 4.4..2: Definicja typu śladów algorytmu Euklidesa w Coqu.

spodziewać składa się on z trzech konstruktorów: `N` kiedy licznik jest większy od mianowania, `D` gdy mianownik jest większy, oraz `One` gdy obie wartości były równe, co zakończyło proces normalizacji. Taka konstrukcja dodatkowo zapewnia nam przyjazną indukcję po wszystkich dodatnich liczbach wymiernych, w przeciwieństwie do naiwnej reprezentacji 4.4..3 wykorzystującej parę liczb naturalnych.

```
Definition Q : Type := (nat * nat).
```

Kod źródłowy 4.4..3: Definicja naiwnej reprezentacji liczb wymiernych w Coqu.

4.4.3. Funkcje przejścia między reprezentacjami

Wiedząc że `Qplus` 4.4..2 reprezentuje ślady algorytmu Euklidesa powinniśmy być w stanie w miarę łatwo napisać kod funkcji go generującej. Niestety rzeczywistość Coqa jest dojść brutalna i termination checker nie potrafi wyznaczyć dobrze

ufundowanego porządku na argumentach algorytmu Euklidesa. My wiemy że jest nim suma argumentów, lecz przekracza to jego automatyczne możliwości. Posłużymy się zatem konceptem zwanym paliwem. Polega on na dodaniu dodatkowego parametru funkcji, który będzie się zmniejszał z każdym wywołaniem rekurencyjnym funkcji, gwarantując terminację w oczach Coqa. Ustalając wartość tego parametru dostatecznie wysoko, będziemy mogli wyliczyć ślad zanim skończy nam się paliwo.

```
Function qplus_c' (p q n: nat) : Qplus :=
match n with
| O    => One
| S n' => match compare p q with
          | Eq => One
          | Gt => N (qplus_c' (p - q) q n')
          | Lt => D (qplus_c' p (q - p) n')
          end
end.

Function qplus_c (x: Q) : Qplus :=
  let (p, q) := x in qplus_c' p q ((p + q) / gcd p q).
```

Kod źródłowy 4.4..4: Definicja funkcji przekształcającą naiwną reprezentację w ilorazowy typ dodanych liczb wymiernych w Coqu.

Jak widzimy w definicji funkcji 4.4..5 paliwo zostało ustalone jako suma argumentów podzielona przez ich największy wspólny dzielnik. Wiemy, że w każdej iteracji algorytm Euklidesa zmniejsza sumę argumentów przynajmniej o ich największy wspólny dzielnik, mamy zatem pewność że zdążymy wyliczyć cały ślad tego algorytmu.

Funkcja odwrotna (z dokładnością do NWD) do zaprezentowanej powyżej ma równie oczywistą konstrukcję. Będziemy odwracać proces poprzez dodawanie wartości drugiego argumentu do tego który w trakcie normalizacji był większy. Wynika to z dojść oczywistego faktu, że $(p - q) + q = p$.

```
Function qplus_i (x : Qplus) : Q :=
match x with
| One  => (1, 1)
| N x' => let (p, q) := qplus_i x' in (p + q, q)
| D x' => let (p, q) := qplus_i x' in (p, p + q)
end.
```

Kod źródłowy 4.4..5: Definicja funkcji przekształcającą ilorazowy typ dodanych liczb wymiernych do naiwnej reprezentacji w Coqu.

4.4.4. Rozszerzenie do ciała liczb wymiernych

Same dodatnie liczby wymierne nie są aż tak użyteczne jak całe ciało liczb wymiernych. Na szczęście istnieje prosty sposób na rozszerzenie tego konceptu, do wszystkich liczb wymiernych. Skorzystamy tutaj z konstrukcji podobnej do tej zaprezentowanej przy liczbach, a więc dodamy typ z trzema konstruktorami: dla liczb dodatnich, dla ujemnych oraz dla zera.

```
Inductive FullQ :=
| Pos  : Qplus -> FullQ
| Zero : FullQ
| Neg  : Qplus -> FullQ.
```

Kod źródłowy 4.4..6: Definicja ilorazowego typu liczb wymiernych w Coqu.

4.4.5. Operacje na liczbach wymiernych

TODO

4.5. Wolne grupy

W tym rozdziale poznaliśmy już wolne monoidy, teraz przyszedł czas na ich bardziej skomplikowaną wersję, czyli grupy. Ma ona trzy podstawowe prawa: istnienie elementu neutralnego, łączność działania oraz istnienie elementu przeciwnego.

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

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

Element odwrotny: $\forall x \in \mathbf{A}, \exists x^{-1} \in \mathbf{A}, x \circ x^{-1} = x^{-1} \circ x = e$

Jak więc możemy zaobserwować każda grupa jest również monoidem, a jedyne co się zmienia to nowa operacja odwrotności. Jak więc możemy się domyślać wolną grupą będą listy z elementami oraz ich odwrotnościami. Naiwną implementację możemy zatem zapisać jako listę par elementu z boolem, który będzie przechowywać informację czy to ten element czy element do niego odwrotny.

```
Definition CanonFreeGroup (A: Type) := list (bool*A).
```

Kod źródłowy 4.5..1: Naiwna implementacja wolnej grupy w Coqu.

4.5.1. Normalizacja wolnej grupy

Od normalizowanej wolnej grupy będziemy wymagać, aby dwa wzajemnie odwrotne elementy nie były swoimi sąsiadami, gdyż jak wiemy z praw grupy takie elementy się "anihilują"i są równoważne elementowi neutralnemu. Predykat bycia

```
Inductive Normalized {A: Type} `{Group A} : CanonFreeGroup A -> Prop :=
| NNil   : Normalized []
| NSingl : forall (b : bool) (v : A), Normalized [(b, v)]
| NCons  : forall (b b': bool) (v v' : A) (x: CanonFreeGroup A),
            v <> v' \/ b = b' -> Normalized ((b', v') :: x) ->
            Normalized ((b, v) :: (b', v') :: x).
```

Kod źródłowy 4.5..2: Predykat bycia w postaci normalnej dla wolnej grupy w Coqu.

w postaci normalnej możemy wyrazić za pomocą typu induktywnego z trzema konstruktorami dla pustej grupy, singletona, oraz dla następnika 4.5..2. Funkcję normalizującą4.5..4 również bez większych problemów da się wyrazić, jednak, żeby to zrobić nasz typ bazowy będzie musiał posiadać zdefiniowaną rozstrzygalną równość na naszym typie bazowym, w postaci funkcji `eqf`.

```
Class Group (A : Type) := GroupDef {
  zero      : A;
  op        : A -> A -> A;
  inv       : A -> A;
  eqf       : A -> A -> bool;
  left_id   : forall x: A, op zero x = x;
  right_id  : forall x: A, op x zero = x;
  left_inv  : forall x: A, op (inv x) x = zero;
  right_inv : forall x: A, op x (inv x) = zero;
  op_assoc  : forall x y z: A, op (op x y) z = op x (op y z);
  eqf_eq    : forall x y, reflect (x = y) (eqf x y)
}.
```

Kod źródłowy 4.5..3: Klasa grup z rozstrzygalną równością w Coqu.

4.5.2. Jednoznaczny typ dla wolnych grup w Coqu

Wiedząc już czym charakteryzują się wolne grupy znormalizowane, możemy przejść do stworzenia typu induktywnego dla nich bazując na funkcji normalizacji. Pierwszą ważną obserwacją jest to, że funkcja zaczyna swoją normalizację od końca, wyznaczając element początkowy. Pod drugie wyrażenie `eqf v v'` jest równoważne przyrównaniu do elementu neutralnego, różnicy `v` oraz `v'`, czyli `eqf (op v (inv v')) zero`.

```
Fixpoint normalize {A: Type} `{Group A} (x: CanonFreeGroup A) :=
match x with
| [] => []
| (b, v) :: x' =>
    match normalize x' with
    | [] => [(b, v)]
    | (b', v') :: x'' => if andb (eqf v v') (xorb b b')
                         then x''
                         else (b, v) :: (b', v') :: x''
    end
end.
```

Kod źródłowy 4.5..4: Funkcja normalizująca wolną grupą w Coqu.

A więc tak naprawdę interesują nasz różnice między elementami. Po trzecie podobnie jak w przypadku elementów, tu również interesują nasz różnice między kolejnymi znakami. Nasz typ ilorazowy będzie zatem odwrotną listą różnicową elementów i ich znaków, czyli taką w której różnice odnoszą się do poprzedniego elementu, z dodatkowym warunkiem zabezpieczającym przed postawieniem dwóch tych samych elementów z innym znakiem obok siebie.

```
Inductive NonEmptyFreeGroup (A: Type) `{Group A} :=
| Singl  : bool -> A -> NonEmptyFreeGroup A
| Switch : forall x: A, Squash (x <> zero) -> NonEmptyFreeGroup A -> NonEmptyFreeGro
| Stay   : A -> NonEmptyFreeGroup A -> NonEmptyFreeGroup A.

Definition FreeGroup (A: Type) `{Group A} := option (NonEmptyFreeGroup A).
```

Kod źródłowy 4.5..5: Ilorazowy typ wolnej grupy w Coqu.

Jak widzimy w definicji 4.5..5 mamy tak naprawdę dwa typy, pustych oraz nie pustych list wolnych. Wynika to z problemu niejednoznaczności dla elementu neutralnego (pustej grupy). Odwrotnością elementu neutralnego jest on sam, a więc zgodnie z prawami nie powinna istnieć pusta grupa "dodatnia"oraz "ujemna", stąd też potrzeba zdefiniowania niepustej wolnej grupy. Sama jej definicja składa się z trzech konstruktorów. Singleton jest dojść prostym konstruktorem bez niespodzianek. Drugi konstruktor natomiast reprezentuje zmianę znaku w stosunku do poprzedniego elementu, w związku z tym wymaga on aby różnica elementów była różna od 0, gdyż w przeciwnym wypadku obok siebie stałyby te same elementy z przeciwnymi znakami. Żeby nie martwić się o jednoznaczność dowodu został on opakowany w `Squash` a więc żyje w świecie z definicyjną irrelewancją dowodów. Ostatni z konstruktorów reprezentuje różnicę bez zmiany znaków, w takim wypadku nie są potrzebne dodatkowe dowody, a więc jest on dojść prosty. Niestety, wymaganie

odnośnie wyliczania różnicy wymusza, aby typ bazowy był sam w sobie grupą z operacją odwrotności.

4.5.3. Funkcje przejścia między reprezentacjami

Mając już zdefiniowany typ ilorazowy dla wolnych list powinniśmy zdefiniować funkcję która pozwoli nam przekonwertować naszą nawianą reprezentację do typu ilorazowego. Jej definicja nie jest zupełnie trywialne ze względu na zastosowany typ zależny w drugim konstruktorze.

```
Fixpoint to_uniq' {A: Type} `{Group A} (b : bool) (v: A) (x: CanonFreeGroup A)
: NonEmptyFreeGroup A :=
match x with
| []             => Singl b v
| (b', v') :: x' =>
    if eqb b b'
    then Stay (sub v v') (to_uniq' b' v' x')
    else match eqf_eq (sub v v') zero with
         | ReflectF _ p => Switch (sub v v') (squash p) (to_uniq' b' v' x')
         | _  => (Singl b v) (* invalid *)
         end
end.

Definition to_uniq {A: Type} `{Group A} (x: CanonFreeGroup A) : FreeGroup A :=
match x with
| [] => None
| (b, v) :: x' => Some (to_uniq' b v x')
end.
```

Kod źródłowy 4.5..6: Definicja funkcji przekształcającej naiwną reprezentacje w ilorazową w Coqu.

Jak widzimy w definicji funkcji `to_uniq` 4.5..6, w przypadku w którym zmieniamy znak musimy wyciągnąć dowód, że dwa kolejne elementy są od siebie różne. Na szczęście użycie `reflect` w definicji grupy pozwala nam w łatwy sposób wyciągnąć taki dowód, jeśli funkcja `eqf` zwraca `false`. W przeciwnym zaś przypadku funkcja zwraca cokolwiek. Wprawne oko pozwala wypatrzeć, że funkcja `to_uniq` działa poprawnie tylko dla znormalizowanych wolnych grup, jeśli chcemy pozbyć się tego ograniczenia należy ją złożyć z funkcją normalizującą `normalize` 4.5..4.

Funkcja odwrotna `to_canon` 4.5..7 ma nieco prostszy kod, gdyż nie wymaga ona tworzenia typów zależnych. Cała jej idea skupia się dodawania zapisanej różnicy do poprzedniego elementu i dzięki temu wyliczanie kolejnych wartości wolnej grupy.

```
Fixpoint to_canon' {A: Type} `{Group A} (x: NonEmptyFreeGroup A) : CanonFreeGroup A
match x with
| Singl b v     => [(b, v)]
| Stay v x'     => match to_canon' x' with
                   | [] => [(true, v)] (* invalid *)
                   | (b, v') :: y => (b, op v v') :: (b, v') :: y
                   end
| Switch v _ x' => match to_canon' x' with
                   | [] => [(true, v)] (* invalid *)
                   | (b, v') :: y => (negb b, op v v') :: (b, v') :: y
                   end
end.

Definition to_canon {A: Type} `{Group A} (x: FreeGroup A) : CanonFreeGroup A :=
match x with
| None => []
| Some x' => to_canon' x'
end.
```

Kod źródłowy 4.5..7: Definicja funkcji przekształcającej ilorazową reprezentacje w naiwną w Coqu.

4.5.4. Wolna grupa jako grupa

4.5.5. Wolna grupa jako monada

4.5.6. Zastosowania wolnych grup

4.6. Listy różnicowe jak multi-zbiory

W poprzednim rozdziale pokazaliśmy jak można zdefiniować zbiry i multi-zbiory w Coqu używając pod-typowania. Jest to skuteczna strategia, której niestety nie da się przenieść do języków nie posiadających typów zależnych takich jak Haskell. Nasuwa się zatem pytanie czy istnieją typy bazowe dla których można zdefiniować multi-zbiory bez wykorzystania pod-typowania. W tym celu wykorzystamy listę różnic pomiędzy kolejnymi elementami multi-zbioru.

4.6.1. Funkcja normalizująca

Jak można się domyślać w celu normalizacji listy elementów w multi-zbiór posłużymy się funkcją sortującą, podobnie jak w poprzednim rozdziale. Tutaj w jednak przyjrzymy się pewnemu zmodyfikowanemu algorytmowi sortowanie przez wybiera-

```
Fixpoint min (h: nat) (l: list nat) : nat :=
match l with
| [] => h
| (h' :: l') => if h' <=? h then min h' l' else min h l'
end.

Fixpoint sub_list (s: nat) (l: list nat) : list nat :=
match l with
| [] => []
| (h :: l') => (h - s) :: sub_list s l'
end.

Fixpoint rm_one_zero (l: list nat) : list nat :=
match l with
| [] => []
| (h :: l') => if h =? 0 then l' else h :: rm_one_zero l'
end.

Fixpoint select_sort' (f: nat) (prev: nat) (l : list nat) : list nat :=
match f with
| 0 => []
| S f' =>
  match l with
  | [] => []
  | (h :: l') => let m := min h l' in
                 let p := prev + m in
                 p :: (select_sort' f' p (rm_one_zero (sub_list m l)))
  end
end.

Definition select_sort (l : list nat) : list nat :=
  select_sort' (length l) 0 l.
```

Kod źródłowy 4.6..1: Definicja sortowania przez wybieranie dla liczb naturalnych w Coqu.

nie dla liczb naturalnych 4.6..1. Modyfikacja ta polega na zoptymalizowaniu kolejnych operacji porównań poprzez odjęcie od każdej liczby znalezione minimum. Porównywanie liczb naturalnych w systemie unarnym jest liniowe względem mniejszej z liczb, w związku z tym im mniejsze liczby na liście tym szybszy proces szukania minimum. Możemy się teraz przyjrzeć śladowi tej funkcji, a konkretnie jak będą wyglądać kolejne minima na sortowanej liście. Z uwagi na to iż w każdym wywołaniu rekurencyjnym wartości są pomniejszane o minimum to kolejne wyliczane minima będą stanowić różnicę pomiędzy kolejnymi (zgodnie z porządkiem na liczbach naturalnych) elementami.

4.6.2. Typ ilorazowy dla multi-zbiorów

Zanim przystąpimy do implementacji typu ilorazowego postaramy się uogólnić przedstawiony powyżej schemat na więcej typów. Po pierwsze potrzebujemy, aby na typie istniał porządek liniowy, gdyż istniały dwie różne minimalne wartości nie moglibyśmy zawsze wyznaczyć unikalnego minimum. Po drugie musi istnieć jakaś operacja różnicy dwóch elementów, oraz operacja dodawania różnicy do wartości, aby odzyskać oryginalny multi-zbiór. W tej pracy została ostatecznie wykorzystana

```
Class Diff (A: Type) `{E: EqDec A} := {
  D: Type;
  diff: A -> A -> D;
  add: A -> D -> A;

  diff_comm: forall x y, diff x y = diff y x;
  diff_def: forall x d, d = diff x (add x d);
  recover: forall x y, y = add x (diff x y) \/ x = add y (diff x y);
  diff_anti_sym : forall x d, x = add (add x d) d -> d = diff x x;
  diff_trans: forall x y z,
    z = add (add x (diff x y)) (diff y z) -> z = add x (diff x z);
}.
```

Kod źródłowy 4.6..2: Definicja klasy typów dla której istnieją unikatowe listy różnicowe w Coqu.

definicja4.6..2 w która swoją konstrukcją implikuje istnienie porządku liniowego, w związku z tym nie wymaga ona jego istnienia jako przesłanki. Wymaga ona za to istnienie typu różnic symetrycznych, funkcji która dla dwóch elementów wylicza takową różnicę oraz funkcja dodająca wskazaną różnicę do elementu. Dodatkowo będziemy wymagać spełnienia pięciu praw. Po pierwsze wymusza symetryczność różnicy, poprzez wymaganie przemienności operacji. Drugie definiuje związek pomiędzy operacją różnicy symetrycznej a dodawaniem. Trzecie prawo wymaga aby z różnicy symetrycznej dało się odzyskać przynajmniej jeden z elementów. Czwarte mówi o

tym, że jeśli dwa razy dodaliśmy jakąś różnicę i wynik się nie zmienił to znaczy że ta różnica jest neutralna, prawo to wraz z trzecim implikuje anty-symetryczność generowanego porządku. Piąte prawo wymaga aby nasze różnice symetryczne były przechodnie, a więc jeśli da się odzyskać jakąś wartość dodając dwie różnice o wspólnym końcu to można ją odzyskać z bezpośredniej różnicy. Mając te prawa możemy zdefiniować porządek liniowy, dwa elementy będą z sobą w relacji wtedy i tylko wtedy gdy dodanie do pierwszej ich wspólnej różnicy daje nam drugą. W dodatku $DifferentialLists.v$ znajduje się dowód tego faktu. Mając zdefiniowana klasę możemy napisać typ list różnicowych4.6..3. Typ ten jest zdefiniowany za pomocą opcji,

```
Definition DiffList (A: Type) `{Diff A} :=
  option (A * list D).
```

Kod źródłowy 4.6..3: Definicja list różnicowych w Coqu.

która umożliwia istnienie pustych multi-zbiorów. W środku opcji mamy głowę listy typu bazowego, wraz z listą różnic.

4.6.3. Funkcje przejścia między reprezentacjami

Po zdefiniowaniu kształtu typu multi-zbiorów z wykorzystaniem list różnicowych, możemy napisać funkcję które pozwolą na przejście między reprezentacją listową a ilorazową.

```
Fixpoint from_diff' {A: Type} `{Diff A} (x: A) (l: list D) :=
  match l with
  | []        => [x]
  | (h :: l') => x :: from_diff' (add x h) l'
  end.

Definition from_diff {A: Type} `{Diff A} (x: DiffList A) : list A :=
  match x with
  | None        => []
  | Some (a, l) => (from_diff' a l)
  end.
```

Kod źródłowy 4.6..4: Definicja funkcji przejścia z listy różnicowej do listowej postaci multi$_z$$bioruwCoqu$.

Funkcja `from_diff` 4.6..4 pozwana nam wygenerować listę elementów z listy różnic. Zasada działania jest bardzo prosta. Po pierwsze dla pustej listy różnicowej zwraca listę pustą. Dla listy różnicowej z samą głową zwracamy singleton. W przypadku dłuższych listy kolejne wartości będą stanowić sumy elementu poprzedzającego wraz z koleją różnicą.

```
Fixpoint to_diff_sorted' {A: Type} `{Diff A} (p: A) (l: list A) : list D :=
  match l with
  | [] => []
  | (h :: t) => diff p h :: to_diff_sorted' h t
  end.

Definition to_diff_sorted {A: Type} `{Diff A} (l: list A) : DiffList A :=
  match l with
  | [] => None
  | (h :: t) => Some (h, to_diff_sorted' h t)
  end.

Definition to_diff {A: Type} `{Diff A} (l: list A) : DiffList A :=
  to_diff_sorted (mergeSort ord l).
```

Kod źródłowy 4.6..5: Definicja funkcji przejścia z listowej postaci multi-zbioru do listy różnicowej w Coqu.

Funkcja `to_diff` 4.6..4 to złożenie dwóch funkcji: sortującej z funkcją generującą listy różnicowe. Zasada jej działania jest również bardzo prosta i polega na wyliczaniu kolejnych różnic pomiędzy elementami listy. Naturalnie pierwszy element listy zostaje zapisany jako głowa listy różnicowej. Obie te funkcje są dojść proste. Można udowodnić, że są one swoimi odwrotnościami z dokładnością do permutacji elementów. Dodatkowo można za ich pomocą potwierdzić, że nasz typ multi-zbioru jest rzeczywiście ilorazowy poprzez wykazanie, że jeśli dwie listy są swoimi permutacjami, to po przejściu do postaci list różnicowych ich reprezentacje będą sobie równe. Dowód tych dwóch faktów można znaleźć w dodatku $DifferentialLists.v$.

4.6.4. Operacje na listach różnicowych

Mając zdefiniowane funkcje przejścia między listami różnicowymi możemy w łatwy sposób wykorzystać wszystkie funkcje zaimplementowane dla list. Oznacza to że nasza implementacja jest zarówno funktorem jak i monadą, oczywiście przyjmując że typ wynikowy również implementuje klasę `Diff ??`. Przejście między reprezentacjami jest jednak kosztowne i ma liniową złożoność obliczeniową względem długości listy. Zatem operacje takie jak dodanie elementu lepiej jest zaimplementować bezpośrednio na liście różnicowej. Przyjrzyjmy się implementacji funkcji 4.6..6 dodającej x. Przechodzi ona przez listę aż do momentu napotkania pozycji na której znajduje się wartość od niej większa. Po jej napotkaniu tworzy dwie różnice pomiędzy poprzednim elementem a x oraz pomiędzy x a następnym elementem (czyli sumie poprzedniego i kolejnej różnicy na liście). Możemy udowodnić że po zaaplikowaniu takiej funkcji na liście znajduje się element x, żaden element nie został usunięty, oraz

```
Fixpoint add_to_diff' {A: Type} `{Diff A} (x: A) (h: A) (l: list D) : list D :=
match l with
| [] => [diff h x]
| (d :: l') => if ord x (add h d)
               then diff h x :: diff x (add h d) :: l'
               else d :: add_to_diff' x (add h d) l'
end.

Definition add_to_diff {A: Type} `{Diff A} (x: A) (l: DiffList A) : DiffList A :=
match l with
| None => Some (x, [])
| Some (h, l) => if ord x h
                 then Some (x, diff x h :: l)
                 else Some (h, add_to_diff' x h l)
end.
```

Kod źródłowy 4.6..6: Definicja funkcji dodającej element do listy różnicowej w Coqu.

to że długość listy zwiększyła się o 1. Dowody tych faktów można znaleźć również w dodatku $DifferentialLists.v$.

Rozdział 5.

Funkcje jako typy ilorazowe

W tym rozdziale zostanie przedstawione w jaki sposób możemy wykorzystać typ funkcji do zdefiniowania pewnych typów ilorazowych. Niestety przestawione poniżej przykłady tworzą więcej problemów niż rozwiązują, w związku z tym są praktycznie bezużyteczne w realnych aplikacjach, nie mniej jednak stanowią ciekawy przykład niekonwencjonalnego podejścia do problemu. Dlatego zostały zamieszczone w tym krótkim rozdziale.

5.1. Jak funkcje utożsamiają elementy

Na początku pracy wspomnieliśmy, iż nie można zdefiniować typu zbiorów oraz multi-zbiorów na dowolnego typu bazowego. Jest to prawda w przypadku typów induktywnych, w których kolejność konstruktorów ma znaczenie, możemy natomiast wykorzystać wbudowany w praktycznie każdy język programowania typ funkcji. Naturalnie, aby móc w rozsądny sposób rozumować o równościach na funkcjach będziemy musieli zapostulować aksjomat ekstensjonalności funkcji5.1..1, nie wprowadza on sprzeczności do systemu Coqa, zatem możemy go bezpiecznie dodać np z biblioteki standardowej. Po jego dodaniu funkcje "zapominają"swój wbudowany al-

```
Definition FunExt := forall (A B: Type) (f g: A -> B),
    (forall x: A, f x = g x) -> f = g.
```

Kod źródłowy 5.1..1: Aksjomat ekstensjonalności funkcji w Coqu.

gorytm na potrzeby sprawdzania równości i dwie funkcje dające dla całej przestrzeni argumentów te same wyniki będą sobie równe. Możemy wykorzystać tą właściwość do "zapomnienia"kolejności elementów bez powoływania się na porządek liniowy.

5.2. Definicja zbioru i multi-zbioru

Mając już ten koncept w głowie możemy możemy przejść do zdefiniowania typu zbioru 5.2..1, jako funkcji rozstrzygającej czy element nalezy do zbioru. W przeci-

```
Definition set (A: Type) : Type := A -> bool.
```

Kod źródłowy 5.2..1: Typ zbioru w Coqu.

wieństwie do innych definicji zbiorów przedstawionych w tej pracy zbiory przedstawione powyżej mogą być potencjalne nieskończone, dla typów które mają nieskończenie wiele elementów. Pozwala nam to na bardziej elastyczne definicje, jednak kosztem obliczalności podstawowych operacji. Jednym z takich operacji jest np sprawdzenie czy zbiór nie jest pusty. Ta operacja w ogólności jest oczywiście nieobliczalna. Możemy w łatwy sposób zdefiniować zbiór liczby rekurencyjnych wywołań (paliwa) potrzebnych aby dany algorytm zakończył obliczenia. Gdybyśmy mogli sprawdzić czy taki zbiór jest pusty czy nie moglibyśmy rozstrzygnąć czy algorytm kiedyś terminuje, czy nie. Z problemu stopu wiemy jednak, że jest to problem nie rozstrzygalny. W oczywisty sposób inne operacie takie jak sprawdzenie, czy dwa zbiory są sobie równe będą również nierozstrzygalne, z tego samego powodu.

Multi-zbiór możemy zdefiniować w bardzo podobny sposób5.2..2 jak zbiór zamieniając jedynie typ dwuelementowy `bool` na typ liczb naturalnych `nat`. Taka

```
Definition mset (A: Type) : Type := A -> nat.
```

Kod źródłowy 5.2..2: Typ multi-zbioru w Coqu.

implementacja wspiera jedynie takie mulit zbiory, które mają skończoną liczbę każdego z elementów. Można jednak zamieniać liczby naturalne na liczby co-naturalne, aby pozbyć się tego ograniczenia. Podobnie jak w przypadku zbiorów tu również występuje problem z rozstrzyganiem niepustości.

5.3. Rekurencyjne operacje na zbiorach

Wykazaliśmy, że podstawowe operacje takie jak sprawdzenie niepustości czy równość nie jest rekurencyjna na potencjalnie nieskończonych zbiorach i multi-zbiorach. Podobnie sprawa się ma do operacji mapowania zbioru, gdy istniała taka rekurencyjna funkcja moglibyśmy dokonać mapowania na typ jednoelementowy wszystkich elementów zbioru i w ten sposób rozstrzygnąć jego niepustość. Zatem zdefiniowane przez nas zbiory nie są ani funktorami ani monadami w obliczalny sposób. Możemy natomiast zdefiniować parę użytecznych funkcji. Sprawdzenie czy dany element należy do zbioru jest trywialne, gdyż sam zbiór jest taką funkcją. Możemy zdefiniować funkcje filtrującą dla zbiorów 5.3..1. Działa ona w sposób leniwy, zatem potencjalna

```
Definition set_filter {A: Type} (p: A-> bool) (s: set A) : set A :=
  fun x: A => if p x then s x else false.
```

Kod źródłowy 5.3..1: Funckja filtrująca dla zbiorów w Coqu.

nieskończoność zbioru jej nie przeszkadza. W podobny sposób można zdefiniować sumę, przekrój oraz dopełnienie 5.3..3. Wykorzystują one osobne sprawdzenie na

```
Definition set_union {A: Type} (s s': set A) : set A :=
  fun x: A => (s x) || (s' x).

Definition set_intersection {A: Type} (s s': set A) : set A :=
  fun x: A => (s x) && (s' x).

Definition set_complement {A: Type} (s: set A) : set A :=
  fun x: A => negb (s x).
```

Kod źródłowy 5.3..2: Definicja sumy, przekroju oraz dopełniania dla zbiorów w Coqu.

jednym i drugim zbiorze, a następnie łącza wyniki za pomocą operatorów na typie `bool`. Bardzo podobnie można je również zaimplementować na mulit-zbiorach używając odpowiednio sumy oraz minimum. Operacja dopełniania nie jest oczywiście zdefiniowania dla multi-zbiorów. Wadą tych leniwych obliczeń jest rosnąca złożoność sprawdzania przynależności do zbioru z każdym kolejnym przekrojem i sumą. Operacje doda elementu do zbioru da się zdefiniować w rekurencyjny sposób, niestety wymaga ona od typu bazowego posiadania zdefiniowanej rozstrzygalnej równości. Jeśli takową mamy możemy ją zdefiniować poprzez przyrównanie czy element od-

```
Definition set_add {A: Type} `{EqDec A} (a: A) (s: set A)  : set A :=
  fun x: A => if eqf x a then true else s x.

Fixpoint list_to_set {A: Type} `{EqDec A} (l: list A) : set A :=
match l with
| []        => fun _ => false
| (h :: l') => set_add h (list_to_set l')
end.
```

Kod źródłowy 5.3..3: Definicja dodawania elementu do zbioru oraz konwersji listy do zbioru w Coqu.

pytywany element to ten który dodajemy i jeśli tak to powinniśmy zwrócić `true`. Wykorzystując tą funkcję można zdefiniować tworzenie zbioru z listy elementów. Podobnie dla multi-zbiorów, lecz zamieniając zwracanie `true` na nakładanie następnika. Dowody poprawności zdefiniowanych powyżej funkcji można znaleźć w dodatku

FunctionalQuotient.v.

Rozdział 6.

Typy ilorazowe w wybranych językach

We wcześniejszych rozdziałach poznaliśmy w jaki sposób możemy zdefiniować typy ilorazowe w Coqu. A mówiąc dokładniej jak możemy w nim obejść problem braku typów ilorazowych. Część rozwiązać nie wymaga użycia typów zależnych i może zostać zaimplementowana w prawie każdym języku programowania , a inne jak te wykorzystujące pod typowanie wymaga aby język wspierał typu zależne od termów. W tym rozdziale skupimy się jednak na językach, które mają mechanizmy pozwalające na bezpośrednią implementację typów ilorazowych, a co za tym idzie można je stosować z dużo większą łatwością.

6.1. Lean

Lean, a bardziej Lean 4, gdyż skupimy się tutaj głównie na tej wersji, jest to asystent dowodzenia rozwijany od 2013 roku. Jest to narzędzie open source pozwalające podobnie jak Coq na dowodzenie poprawności programów, jak i twierdzeń matematycznych. Jest ono częścią programu Microsoft Research.

6.1.1. Różnice w stosunku do Coqa

Lean w przeciwieństwie do Coqa nie wymaga konstruktywności dla twierdzeń[3]. Oznacza to, że mamy do czynienia z systemem bardziej zbliżonym do matematycznych dowodów twierdzeń, niż do programowania. Oznacza to niestety, że nie każdy dowód jest w istocie programem, w myśl izomorfizmu Currego-Howarda. W Leanie ponadto mamy definicyjną irrelewację zdań. System formalny opiera się na trzech aksjomatach:

Aksjomat ekstensjonalności zdań - mówi on, że jeśli dwa zdania są sobie rów-

noważne, to są sobie równe.

```
Axiom prop_ext: forall P Q: Prop, (P <-> Q) <-> (P = Q)
```

Aksjomat wyboru - mówi on, że z każdego niepustego typu możemy wyprodukować jego element.

```
Inductive NonEmpty (A: Type) : Prop := intro : A -> NonEmpty A.
Axiom choise: forall A: Type, NonEmpty A -> A.
```

Aksjomat istnienia ilorazów - mówi on, że dla każdego typu oraz relacji możemy wyprodukować typ ilorazowy, w którym wszystkie elementy które są ze sobą w tej relacji są sobie równe.

Część z was zapewne wie, że jedną z głównych różnić w stosunku do Coqa jest zastosowanie w Leanie logiki klasycznej. Wydawałoby się zatem, że prawo wyłączonego środka powinno być jednym z aksjomatów w tym systemie. Okazuje się jednak, że wymienione powyżej aksjomaty wystarczą do wyprowadzenia prawa wyłączonego środka jak i egzystencjaliści funkcji. Egzystencjalność można wyprowadzić z istnienia ilorazów, natomiast dowód wyłączonego środka korzysta z konstrukcji zaproponowanej przez Diaconescu w 1975 roku [5]. Dowody te można również znaleźć w bibliotece standardowej leana pod nazwą `em` oraz `funext`.

6.1.2. Typy ilorazowe

Jak widzieliśmy powyżej, typy ilorazowe są w sercu języka Lean. Znając już główne różnice z Coqiem powinniśmy omówić dokładniej jakie aksjomaty towarzyszą typom ilorazowym w Leanie.

```
Axiom Quot : forall {A: Type}, (A -> A -> Prop) -> Type.
```

Kod źródłowy 6.1..1: Odpowiednik aksjomatu `Quot` w Coqu.

Pierwszy aksjomat `Quot` 6.1..1 postuluje istnienie typów ilorazowych, tworzonych z dowolnego typu bazowego, oraz dowolnej relacji na tym typie.

```
Axiom Quot_mk : forall {A:  Type} (r: A -> A -> Prop),
  A -> Quot r.
```

Kod źródłowy 6.1..2: Odpowiednik aksjomatu `Quot.mk` w Coqu.

Drugi aksjomat `Quot.mk` 6.1..2 postuluje istnienie funkcji, która tworzy elementy typu ilorazowego.

```
Axiom Quot_ind :
  forall (A:  Type) (r: A -> A -> Prop) (P: Quot r -> Prop),
    (forall a: A, P (Quot_mk r a)) -> forall q: Quot r, P q.
```

Kod źródłowy 6.1..3: Odpowiednik aksjomatu `Quot.ind` w Coqu.

Trzeci aksjomat `Quot.ind` 6.1..3 postuluje prawa indukcji na typach ilorazowych. Mówią one, że jeśli dla każdego elementu powstałego z elementu typu bazowego zachodzi predykat, to zachodzi on dla całego typu ilorazowego.

```
Axiom Quot_lift :
  forall (A:  Type) (r: A -> A -> Prop) (B: Type) (f: A -> B),
    (forall a b: A, r a b -> f a = f b) -> Quot r -> B.

Axiom Quot_lift' : forall {A:  Type} {r: A -> A -> Prop} {B: Type}
  (f: A -> B) (P: forall a b: A, r a b -> f a = f b) (x: A),
    f x = Quot_lift f P (Quot_mk r x).
```

Kod źródłowy 6.1..4: Odpowiednik aksjomatu `Quot.lift` w Coqu.

Czwarty aksjomat `Quot.lift` 6.1..4 mówi o tym jak możemy aplikować funkcje, z typu bazowego na elementach typu ilorazowego. Aby taka aplikacja była możliwa wymaga on, aby funkcja szanowała relacje, a więc jeśli dwa elementy są ze sobą w relacji, to wynik funkcji dla nich obu musi być taki sam. W Leanie aksjomat `Quot.lift` przychodzi w parze z regułą przepisywania, która mówi iż rzeczywiście zaaplikowanie funkcji f respektującej relację na elemencie typu ilorazowego daje identyczny efekt, jak zaaplikowanie jej na tym samym elemencie typu pierwotnego. Stad też aby można było skorzystać z typów ilorazowych takich jak w Leanie należy w Coqu zapostulować dodatkowy aksjomat z tą regułą przepisywania.

```
Axiom Quot_sound :
  forall (A: Type) (r: A -> A -> Prop) (a b: A),
    r a b -> Quot_mk r a = Quot_mk r b.
```

Kod źródłowy 6.1..5: Odpowiednik aksjomatu `Quot.sound` w Coqu.

Piąty i ostatni aksjomat `Quot.sound` 6.1..5 postuluje równość elementów typu ilorazowego, jeśli są one w relacji. Oznacza to, że typy ilorazowe w Leanie rzeczywiście są ilorazowe, czyli sklejone zgodnie z relacją.

Przedstawione powyżej aksjomaty są wystarczające, żeby wprowadzić do języka typy ilorazowe. Możemy je bez problemu również wprowadzić do Coqa i cieszyć się w nim typami ilorazowymi opartych na tych aksjomatach.

6.2. Agda

Agda jest językiem programowania stworzonym z myślą o wsparciu dla typów zależnych [2]. Postał on jako rozszerzenie teorii typów Martina-Löfa [9]. Z uwagi na te cechy może służyć jako asystent dowodzenia, w przeciwieństwie jednak do języków jakich jak Coq czy Lean Agda nie posiada języka taktyk, co znacząco utrudnia wykorzystanie jej w tym celu. Natomiast jej obsługa typów zależnych jest na dużo wyższym poziomie niż to co możemy doświadczyć w Coqu. Potrafi sama w wielu wypadkach wywnioskować, że danych przypadek jest niemożliwy, przez co definiowanie funkcji zależnych jest duża łatwiejsze niż w Coqu.

```
lookup :  {A} {n} → Vec A n → Fin n → A
lookup (x  xs) zero    = x
lookup (x  xs) (suc i) = lookup' xs i
```

Kod źródłowy 6.2..1: Definicja funkcji zwracającej n-ty element zależnego wektora w Agdzie.

6.2.1. Kubiczna Agda

Cubical jest rozszerzeniem do języka Agda, który rozbudowuje możliwości języka o kubiczną teorię typów [10]. Pozwala ono na modelowanie konstrukcji z homotopicznej teorii typów natywnie w Agdzie. A więc daje dostęp do takich funkcji jak transport 2.3..30 poznanego w wcześniej. Wprowadza ono koncepcję ścieżek jako dowodów równości między elementami. Wszystkie te własności mogliśmy w większym lub mniejszym stopniu zamodelować w Coqu, tym co wyróżnia kubiczną Agdę na tle innych asystentów dowodzenia, są wyższe typy indykatywne (HIT). Tak jak zwykłe typy indykatywne pozwalają na definiowanie pewnych struktur danych, w przeciwieństwie jednak do znanych nam już typów induktywnych które automatycznie generują równości między elementami, w wyższych typach induktywnych możemy ręcznie dodać kolejne ścieżki. Oznacza to pełną dowolność w definiowaniu równości na definiowanym typie.

```
data S : Set where
  base : S
  loop : base = base
```

Kod źródłowy 6.2..2: Definicja okręgu w kubicznej Agdzie.

6.2.2. Definicja typów ilorazowych za pomocą wyższych typów induktywnych

Jak możemy zobaczyć definicja multi-zbioru skończonego 6.2..3 jest niezwykle łatwa w kubicznej Agdzie. Wystarczyło dodać dodatkową ścieżkę (dowód równości) na listach, które mają dwa początkowe elementy zamienione kolejnością. Przechodniość równości jest zapewniona z jej definicji, a więc nie musimy jej definiować, natomiast równość dla pustego mulit zbioru oraz następnika dostajemy za darmo, z ich definicji. Ostatecznie więc definicja ta jest krótsza niż klasyczna definicja permutacji w Coqu.

```
data Bag (X : Set) : Set where
  nil  : Bag X
  _::_ : X -> Bag X -> Bag X
  swap : (x y : X) (z : Bag X) -> x :: y :: z = y :: x :: z
```

Kod źródłowy 6.2..3: Definicja multi-zbioru skończonego w kubicznej Agdzie.

W bardzo podobny sposób możemy zdefiniować liczby całkowite w kubinczej Agdzie, za pomocą zera, następnika oraz poprzednika. Aby zagwarantować jednak izomorfizm tej konstrukcji z znanymi nam z matematyki liczbami całkowitymi musimy dodać dwie dodatkowe równości mówiące w jaki sposób następnik i poprzednik nawzajem się niwelują. Otrzymaliśmy zatem prawie liczby całkowite 6.2..4, ich problemem jest fakt, że zgodnie z naszą definicją istnieje wiele dowodów równości między tymi samymi liczbami całkowitymi, możemy go rozwiązać dodając jeszcze jedną równość tym razem między dowodami równości, ale na potrzeby tej pracy możemy uznać uznać tą definicję za sukces.

```
data Int : Set where
  zero  : Int
  succ  : Int -> Int
  pred  : Int -> Int
  ps_eq : (z: Int) -> pred (succ z) = z
  sp_eq : (z: Int) -> succ (pred z) = z
```

Kod źródłowy 6.2..4: Definicja liczb całkowitych w Agdzie.

Podsumowując wyższe typy indykatywne nadają się doskonale do definiowania typów ilorazowych, niestety nie możemy ich zastosować do definiowania typów ilorazowych w Coqu gdyż jego system typów nie wspiera tak zaawansowanych konstrukcji. Próba odtworzenia dodawania równości między elementami za pomocą aksjomatów bardzo szybko skoczy się wprowadzeniem do systemu sprzeczności,

Bibliografia

[1] Calculus of inductive constructions.

[2] Agda documentation, 2021. Accessed on Febrauary 22, 2023.

[3] Theorem proving in lean 4, 2022. Accessed on Febrauary 18, 2023.

[4] Y. Bertot. A simple canonical representation of rational numbers. *Electronic Notes in Theoretical Computer Science*, 85(7):1–16, 2003. Mathematics, Logic and Computation (Satellite Event of ICALP 2003).

[5] R. Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975.

[6] P. D. Groote. *The Curry-Howard Isomorphism*. Academia, 1995.

[7] M. HEDBERG. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[8] N. Institute for Advanced Study (Princeton, U. F. Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations Program, 2013.

[9] P. Martin-Löf. An intuitionistic theory of types: Predicative part. H. Rose, J. Shepherdson, redaktorzy, *Logic Colloquium '73*, wolumen 80 serii *Studies in Logic and the Foundations of Mathematics*, strony 73–118. Elsevier, 1975.

[10] A. Mörtberg. Cubical methods in homotopy type theory and univalent foundations. *Mathematical Structures in Computer Science*, 31(10):1147–1184, 2021.

[11] T. Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.