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

16 kwietnia 2023

Streszczenie

. . .

. . .

Spis treści

Rozdział 1.

Definicja problemu

1.1. Cel pracy

Celem niniejszej pracy jest analiza możliwych sposobów definiowania typów ilorazowych w Coqu. Skupimy się w niej jednak jedynie na typach ilorazowych, dla których istnieje funkcja normalizująca. Przedstawimy, dla których typów można zdefiniować taką funkcję, a dla których jest to niemożliwe. W naszej analizie skoncentrujemy się na podejściu wykorzystującym predykat równości zdefiniowany w bibliotece standardowej Coqa, bez wykorzystania dodatkowych aksjomatów. Niemniej jednak, pokażemy, jakie możliwości dają ich wykorzystanie w odpowiednich konstrukcjach. Język Coq nie posiada wsparcia dla typów ilorazowych, dlatego będziemy musieli wykorzystać inne techniki, które pozwolą nam na stworzenie typów ilorazowych dla konkretnych problemów. Głównym tematem zainteresowania w tej pracy są typy indykatywne, które pozwalają na jednoznaczną reprezentację danego typu ilorazowego oraz ich definiowane na podstawie funkcji normalizujących.

1.2. Czym jest Coq?

Coq jest darmowym asystentem dowodzenia na licencji GNU LGPL. Jego pierwsza wersja powstała w 1984 roku jako implementacja bazującego na teorii typów rachunku konstrukcji (Calculus of Constructions). Od 1991 roku Coq wykorzystuje bardziej zaawansowany rachunek indukcyjnych konstrukcji (Calculus of Inductive Constructions) [1], który pozwala zarówno na logikę wyższego rzędu, jak i na statycznie typowane funkcyjne programowanie, wszystko dzięki izomorfizmowi Curry'ego - Howarda [6].

Coq pozwala na proste obliczenia, lecz jest przygotowany pod ekstrakcję już gotowych programów do innych języków funkcyjnych, jak np. OCaml. W Coqu każdy term ma swój typ, a każdy typ ma swoje uniwersum:

`Prop` - jest to uniwersum twierdzeń. Jest ono niepredykatatywne, co pozwala na tworzenie zdań, które coś mówią o innych zadaniach. To uniwersum jest usuwane podczas ekstrakcji kodu, przez co niemożliwe 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 twierdzenia są z definicji tym samym dowodem.

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

`Type` - jest to naduniwersum 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)`. Indeksy są jednak domyślne podczas dowodzenia w Coqu.

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

1.3. Czym są typy ilorazowe

W algebrze abstrakcyjnej zbiór pierwotny T, w którym utożsamiamy elementy zgodnie z relacją równoważności ($\sim$) nazywamy strukturą ilorazową. Oznaczmy ją symbolem $T/\sim$. Dobrym przykładem tego typu struktury jest grupa z modularną arytmetyką. Każdy z nas zaznajomiony jest 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

Aby sformalizować typy ilorazowe, musimy najpierw dokładnie zdefiniować, jakie relacje nazywamy relacjami równoważności. Każda taka relacja musi spełniać trzy własności:

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

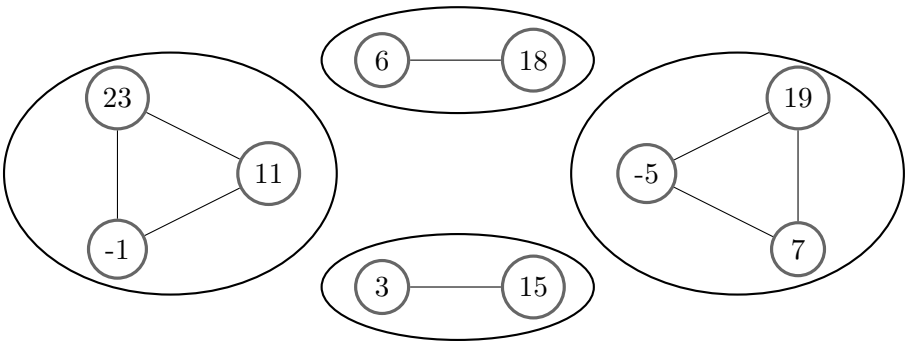
Symetryczność - jeśli a jest w relacji z b ($a \sim b$), to również relacja zachodzi w odwrotnej kolejności ($b \sim a$).

Przechodniość - jeśli istnieje punkt b, z którym zarówno a, jak 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: Klasa relacji równoważności zapisana w Coq

Klasę relacji równoważności zapisaną w Coq przedstawia kod 1. Równość jest najlepszym przykładem takiej relacji - po chwili zastanowienia widzimy, że spełnia ona wszystkie trzy wymagane własności. Relacje równoważności pozwalają na utożsamienie różnych elementów naszego pierwotnego zbioru z sobą, na przykład godzinę 13:00 z 1:00. Innym przykładem może być utożsamienie różnych reprezentacji tego samego ułamka np $\frac{1}{2}$ z $\frac{2}{4}$. Z punktu widzenia teorii mnogości utożsamione z sobą elementy tworzą klasy abstrakcji. Właściwie mówiąc, w tej teorii $T/\sim$ jest rodziną klas abstrakcji, czyli 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

Ponieważ praca skupia się na implementacji typów ilorazowych w Coqu, skupimy się na spojrzeniu teorii typów na ilorazy, w przeciwieństwie do podejścia teoriomnogościowego. W teorii typów, typ ilorazowy generowany przez relację równoważ-

ności ($\sim$) na typie pierwotnym T nazywamy $T/\sim$. Oznaczamy $a = b$ w typie $T/\sim$ wtedy i tylko wtedy, gdy $a \sim b$. Widzimy, że każdy element typu T jest również elementem typu $T/\sim$. Jednak 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 $a \sim b$ implikuje $f(a) = f(b)$. Warunek ten jest konieczny, aby nie dało się rozróżnić utożsamionych wcześniej elementów poprzez zmapowanie ich do innego typu. Myślenie o wszystkich elementach będących w relacji ($\sim$) jako o jednym elemencie, przy jednoczesnym braku tej zasady, prowadzi do złamania zasady monotoniczności aplikacji, która mówi, że $x = y \Rightarrow f(x) = f(y)$.

1.4. Relacje równoważności generowane przez funkcję normalizującą

Każda funkcja $h : T \to B$ generuje dla nas 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 wspomnieliśmy, równość jest relacją równoważności, więc łatwo pokazać, że $\sim_h$ również jest relacją równoważności. Dowolna funkcja $g : B \to X$ generuje teraz dobrze zdefiniowaną funkcję $f : T/\sim_h \to X$ w ten sposób, że $f = g \circ h$.

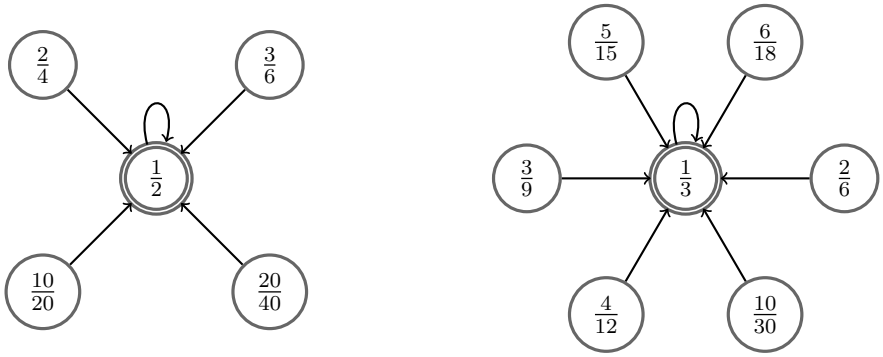
1.4.1. Funkcja normalizująca

W tej pracy skupimy się na szczególnym przypadku funkcji $h : T \to T$, dla której funkcja h jest idempotentna. Tak zdefiniowaną funkcję h będziemy nazywać funkcją normalizującą.

W klasie funkcji normalizujących, zdefiniowanej poniżej2, będziemy wymagać, aby funkcja normalizująca h spełniała warunek idempotencji. Relacja równoważności, jak wiemy, łączy ze sobą elementy w grupy, a mówiąc ściślej, klasy abstrakcji. Celem funkcji normalizujących jest wyznaczenie reprezentanta każdej z klas abstrakcji. Obrazem tej funkcji będzie zbiór naszych reprezentantów lub inaczej elementów w postaci normalnej. Warunek idempotencji jest konieczny, aby obrazem elementu w postaci normalnej był on sam, gdyż nie wymaga już normalizacji.

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

Kod źródłowy 2: Klasa funkcji normalizujących



Rysunek 1.2: Przykład działania funkcji normalizującej dla reprezentacji liczby wymiernych w postaci $\mathbb{Z} \times \mathbb{N}$

1.4.2. Przykłady funkcji normalizujących

Wcześniej przytoczony przykład liczb wymiernych można zdefiniować w postaci $\mathbb{Z} \times \mathbb{N}$, gdzie postacią normalną liczby jest para liczb względnie pierwszych. Funkcja normalizująca powinna dzielić licznik i mianownik przez ich największy wspólny dzielnik.

Dla par nieuporządkowanych, ale na których istnieje jakiś porządek liniowy (np. liczby rzeczywiste) postacią normalną może być para uporządkowana, w której mniejszy element występuje na początku, a funkcją normalizującą będzie funkcja sortująca dwa elementy.

Kolejnym przykładem jest arytmetyka modularna o podstawie m. Tutaj postacią normalną elementów jest pozostawienie jedynie elementów od zera do $m - 1$, a więc sama operacja jest naszą funkcją normalizującą.

1.5. Typy ilorazowe bez funkcji normalizujących

Wykorzystanie funkcji normalizujących w definicji typów ilorazowych jest bardzo wygodne, ponieważ pozwala ograniczyć liczbę reprezentantów danej klasy abstrakcji do jednego, który jest w postaci normalnej. Niestety, nie każdy typ ilorazowy posiada taką funkcję. W teorii mnogości z aksjomatem wyboru można zawsze wybrać zbiór selektorów z każdej rodziny niepustych zbiorów, w naszym przypadku zbiór elementów w postaci normalnej. Jednakże 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 zbudować parę wykorzystując go. Niestety, w ogólności jest to niemożliwe. Wraz z tym idzie, że zbiory, jak i multizbiory, 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 którego w oczekiwany 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 wyznaczają, jaką liczbę rzeczywistą reprezentuje dany ciąg. Chcielibyśmy, aby ciągi o tej samej granicy były ze sobą w relacji równoważności. Jednakże wyznaczenie granicy ciągu mając jedynie czarną skrzynkę, która może jedynie wyliczać kolejne jej elementy, jest niemożliwe. Załóżmy jednakże, że jest to 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 problemem przeliczanie rekurencyjnym (ciąg s_k może reprezentować czy maszyna Turinga zakończyła działania programu przed k-tą sekundą). Oba ciągi mają różne granice, a co za tym idzie, powinny mieć różne postaci normalne. Istnieje zatem jakiś element, na którym się różnią. Gdyby istniała taka obliczalna i całkowita funkcja normalizująca, moglibyśmy za jej pomocą rozwiązać problem stopu w sposób rekurencyjny, sprawdzając jedną wartość ciągu w postaci normalnej. Zatem taka funkcja nie może istnieć.

1.5.3. Monada częściowych obliczeń

Kolejnym typem ilorazowym, dla którego nie istnieje funkcja normalizująca, jest typ częściowych obliczeń. Jak widać w definicji 3, jest to typ koinduktywny, który produkuję albo wynik działania funkcji, albo dalsze obliczenia, które mogą, ale nie muszą kiedykolwiek się zakończyć.

W relacji równoważności chcielibyśmy, żeby były wszystkie obliczenia, które ostatecznie zwracają ten sam wynik. Naturalnie, istnieje osobna klasa abstrakcji dla obliczeń, które nigdy się nie kończą. Jednakże, podobnie jak w przypadku powyższym, mimo że możemy dla każdej klasy abstrakcji wyznaczyć jej reprezentanta w

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

Kod źródłowy 3: Definicja monady częściowych obliczeń w Coqu.

łatwy sposób (obliczenie, które natychmiast zwraca wynik), to nie jest to funkcja całkowita rekurencyjna.

Można zauważyć, że za pomocą monady częściowych obliczeń możemy reprezentować dowolne obliczenia wykonywane na maszynie Turinga. Gdybyśmy potrafili w skończonym czasie wyznaczyć reprezentanta dla każdego obliczenia, moglibyśmy w skończonym czasie stwierdzić, czy maszyna Turinga kiedykolwiek się zatrzyma, czy też nie. Oznaczałoby to, że problem stopu byłby rozstrzygalny. Jednakże, jak wiadomo, problem stopu jest nierozstrzygalny, a więc tak funkcja nie może istnieć.

Rozdział 2.

Podtypowanie, a typy ilorazowe

2.1. Czym jest podtypowanie?

Koncept podtypowania jest dość intuicyjny. Możemy wyobrazić sobie typy jako zbiory elementów, które należą do danego typu. Wtedy podtyp to będzie podzbiór naszego pierwotnego typu. Pozwala to na wyrzucanie z typu wszystkich niepożądanych elementów z naszego pierwotnego typu. Na przykład, jeśli piszemy funkcję wyszukiwania binarnego, chcielibyśmy otrzymać jako jej argument posortowaną listę. W tym celu możemy użyć podtypowania, wymuszając, aby akceptowane były tylko listy, które spełniają predykat posortowania.

```
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 4: Dwie równoważne definicje podtypowania w Coqu.

W Coqu możemy zdefiniować podtypowanie na dwa równoważne sposoby 4. Pierwszy z nich wykorzystuje jeden konstruktor, natomiast druga rekord. Najbardziej ogólna definicja podtypowania wykorzystuje typy zależne, które nie są dostępne w większości języków programowania. W większości języków programowania na programiście spoczywa obowiązek upewnienia się, czy lista jest posortowana, ponieważ ekspresywność języka jest zbyt uboga, aby zapisać takie wymagania.

W bibliotece standardowej Coqa `Coq.Init.Specif` zdefiniowane są `sig` oraz `sigT`, która jest uogólnieniem definicji podtypowania. Konstrukcja `sig` ma notację `{a : A | P a}`, która ma przypominać matematyczny zapis $x \in A : P(x)$. Kon-

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

Kod źródłowy 5: Definicja definicja sigma typu z biblioteki standardowej Coqa.

strukcja `sigT` jest parą zależną, w której drugi element pary zależy od pierwszego. Konstrukcja ta ma notację `{a : A & P}`.

2.1.1. Związek z typami ilorazowymi

Podtypowanie jest dualnym pojęciem do typów ilorazowych, które zostały opisane wcześniej w tej pracy. Omawiamy je w tym rozdziale z powodu braku wsparcia dla podtypowania w Coqu. Nie ma możliwości, abyśmy w Coqu wymusili równość dwóch różnych elementów danego typu w żaden inny sposób niż przez użycie aksjomatu. Jednak użycie aksjomatów jest niepraktyczne, ponieważ niszczy to obliczalność dowodu, a także bardzo łatwo można takimi aksjomatami doprowadzić do sprzeczności w logice Coqa. Dlatego wykorzystamy koncept podtypowania, który będzie wymuszał istnienie jedynie normalnych postaci danej klasy abstrakcji w naszym typie ilorazowym, do zdefiniowania typów ilorazowych. Osiągniemy to poprzez wymuszenie istnienia jedynie elementów w postaci normalnej w zdefiniowanym przez nas typie. Dzięki temu będziemy mogli pracować na typach ilorazowych z wykorzy-

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

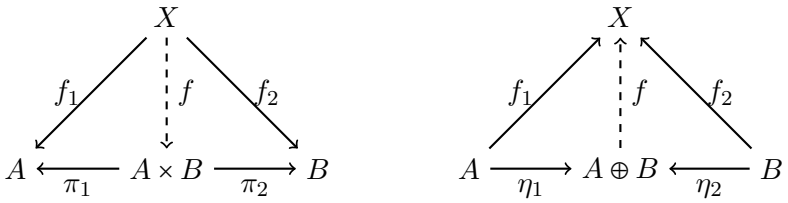
Kod źródłowy 6: Definicja podtypu kanonicznych postaci względem funkcji normalizującej f.

staniem ograniczonej liczby narzędzi, które dostarcza nam Coq.

2.2. Co oznacza dualność pojęć?

W poprzedniej sekcji wspomnieliśmy, że podtypowanie jest pojęciem dualnym do ilorazów, tutaj wyjaśnimy, co to oznacza. Dualizm to pojęcie ze świata teorii kategorii, a aby zrozumieć tę sekcję, wymagana jest podstawowa wiedza z tego zakresu. Jeśli jednak ktoś nie posiada takiej wiedzy, może spokojnie pominąć tę sekcję, 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:

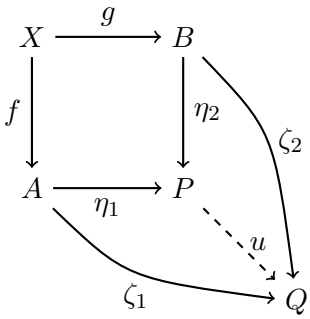
- zamianę pojęć elementu początkowego i końcowego nawzajem,
- zmianę kolejności składania morfizmów.



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

Mając to już za sobą, możemy powiedzieć prostym językiem, że dualność polega na zmianie kierunku strzałek w naszej konstrukcji. Teraz możemy zadać sobie pytanie, gdzie w ilorazach i podtypowaniu występują jakieś strzałki, których kierunki mielibyśmy zamieniać? Występują one odpowiednio w pushoutach oraz pullbackach.

2.2.1. Czym jest pushout?



Rysunek 2.2: Diagram definiujący pushout P. Diagram komutuje.

Na rysunku 2.2 przedstawiony jest komutujący diagram definiujący pojęcie pushoutu. Zauważmy, że powstaje on z dwóch morfinizmów $f : X \to A$ i $g : X \to B$. Ponieważ diagram ten komutuje wiemy, że $\eta_1 \circ f = \eta_2 \circ g$. Pushout P jest najlepszym obiektem, dla którego ten diagram zachowuje tę własność. Oznacza to, że dla każdego innego obiektu (na diagramie Q), dla którego zewnętrzna część diagramu (X, A, B, Q) również komutuje, istnieje dokładnie jeden unikatowy morfizm u z P do Q. Należy jednak zauważyć, że nie dla każdej pary morfinizmów $f : X \to A$ i $g : X \to B$ istnieje pushout. Jednakże, jeśli istnieje, to jest on unikatowy z dokładnością do unikatnego izomorfizmu.

2.2.2. Przykład pushoutu w kategorii *Set*

W definicji pushoutu łatwo zauważyć morfizmy (strzałki), ale trudno zrozumieć o co dokładnie chodzi z ilorazami, o których była mowa powyżej. Łatwiej jest zrozumieć to na przykładzie kategorii *Set*. W tej kategorii obiekty to zbiory, a morfizmy (strzałki) to funkcje między zbiorami. Na rysunku 2.2 widzimy, że obiekty A, B i P tworzą co-produkt. Zatem warto rozpocząć definiowanie P od obiektu $A \oplus B$. Jednakże musimy zadbać o to, aby nasz diagram komutował. Dla każdego $x \in X$ mamy $\eta_1(f(x)) = \eta_2(g(x))$, więc musimy zdefiniować relację równoważności $\sim$, utożsamiającą elementy $f(x)$ i $g(x)$ dla każdego $x \in X$, aby zapewnić komutatywność wewnętrznej części diagramu (X, A, B, P). Nie możemy jednak wybrać dowolnej relacji $\sim$, która spełnia ten warunek, ponieważ musimy zagwarantować istnienie funkcji u dla każdego innego zbioru, dla którego ten diagram będzie komutatywny. Oznacza to, że musi istnieć surjekcja ze zbioru P do Q. Ponieważ funkcja u musi spełniać $u \circ \eta_1 = \zeta_1$ oraz $u \circ \eta_2 = \zeta_2$, relacja równoważności $\sim$ musi być najdrobniejszą taką relacją, która spełnia ten warunek. Jeżeli nie byłaby najdrobniejsza, oznaczałoby to, że istnieje takie Q dla którego diagram również komutuje, ale mające więcej elementów, a z tego wynika, że nie może istnieć suriekcja u, gdyż byłaby zbioru mniej licznego do zbioru bardziej licznego. Jeśli nasz pushout P jest równy $(A \oplus B)/\sim$, to diagram 2.2 będzie komutatywny. Widzimy zatem, że pushout faktycznie ma związek z typami ilorazowymi, w których X definiuje, które elementy zostaną ze sobą utożsamione.

2.2.3. Sklejanie dwóch odcinków w okrąg, czyli pushout

Rozważyliśmy bardziej przyziemny, ale nadal abstrakcyjny przykład w kategorii *Set*. Skonstruujmy teraz bardziej wizualny przykład, którym będzie topologiczny okrąg na rysunku 2.3. Aby upewnić się, że rzeczywiście mamy do czynienia z topolo-

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

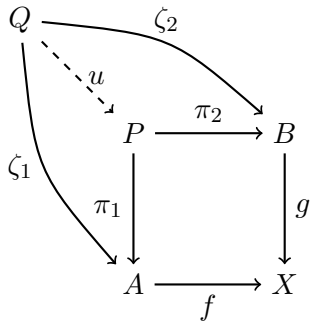
Rysunek 2.3: Diagram definiujący okrąg C używając pushoutu

gicznym okręgiem, musimy sprawdzić, czy diagram 2.3 komutuje. W tym celu musi zachodzić $\eta_1(0) = \eta_2(0)$ oraz $\eta_1(1) = \eta_2(1)$. Skleiliśmy zatem ze sobą te dwa punkty. Ponieważ C musi być najlepszym obiektem, dla którego diagram komutuje, żadne inne punkty nie mogą być ze sobą sklejone. Dlatego dla każdego $x \in (0,1)$ mamy $\eta_1(x) \neq \eta_2(x)$. W ten sposób faktycznie skonstruowaliśmy topologiczny okrąg, składający się z dwóch odcinków i dwóch punktów sklejeń. Metoda ta uogólnia się na

wyższe wymiary. Na przykład mając dwie n-wymiarowe półkule, które następnie skleimy wzdłuż $(n-1)$-wymiarowej kuli, otrzymamy kulę n-wymiarową.

2.2.4. Czym jest pullback?

Wiedząc, jak wygląda pushout, oraz 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 morfi-



Rysunek 2.4: Diagram definiujący pullback P. Diagram komutuje.

zmów $f : A \to X$ oraz $g : B \to X$. Z komutacji wiemy, że $f \circ \pi_1 = g \circ \pi_2$. Podobnie jak w przypadku pushoutu, tutaj również P musi być najlepszym obiektem, dla którego diagram komutuje, aby być pullbackiem. Oznacza to, że dla każdego innego obiektu Q, dla którego zewnętrzna część diagramu (X, A, B, Q) komutuje, istnieje unikalny 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 pullback. Jednak, jeśli istnieje, to jest on unikalny z dokładnością do unikatowego izomorfizmu.

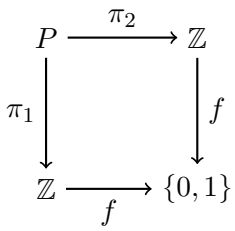
2.2.5. Przykład pullbacku w kategorii *Set*

Aby zobaczyć podtypowanie w zdefiniowanym powyżej pullbacku omawianym w tym rozdziale, przeniesiemy się do prostszego świata kategorii Set, gdzie żyją zbiory i 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 $A \times B$. Jednakże, aby diagram komutował, 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, które nie spełniają tego warunku, otrzymując ostatecznie $P = (a, b) \in A \times B : f(a) = g(b)$. Upewnijmy się jeszcze, że 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 powodu komutowania zewnętrznej części diagramu. Możemy w tym miejscu zauważyć, że rzeczywiście pullback ma związek z podtypowaniem, gdyż definiujemy go 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 2.4 widzimy definicję pullbacku P za pomocą morfizmu



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

$f : \mathbb{Z} \to 0, 1$, zdefiniowanego jako $f(n) = n; \mathrm{mod}; 2$. Jak wiemy z poprzedniego przykładu, $P = (n, m) \in \mathbb{Z} \times \mathbb{Z} : n; \mathrm{mod}; 2 = m; \mathrm{mod}; 2$. A więc jest to zbiór par dwóch liczb całkowitych, które przystają do siebie modulo 2, czyli mówiąc inaczej, mają tę samą parzystość.

2.2.7. Konkluzja

Jak już zauważyliśmy w poprzednich przykładach, pushouty pozwalają na wyznaczanie obiektów, które utożsamiają pewne elementy w stosunku do pewnej relacji równoważności generowanej przez morfizmy. Z kolei pullbacki pozwalają nam tworzyć obiekty z elementami, które spełniają określony przez morfizmy warunek, co czyni je odpowiednikami podtypów. Ponieważ te pojęcia są dualne - co możemy zobaczyć na diagramach 2.2 oraz 2.4 - możemy mówić o nich jako o wzajemnie dualnych.

2.3. Unikatowość reprezentacji w podtypowaniu

Niestety, podtypowanie w Coqu nie dostarcza nam tak prostego interfejsu, jak moglibyśmy się spodziewać na podstawie matematycznego podejścia do tego konceptu. W teorii zbiorów jesteśmy przyzwyczajeni do zapisów w stylu $8 \in x \in \mathbb{N} : \mathrm{even}(x)$, jednak w Coqu zapis `8 : {x : nat | even x}` powoduje konflikt typów, ponieważ

8 jest typu `nat`, a nie typu `{x : nat | even x}`. Wynika to z definicji typu `sig`, który jest parą zależną. Aby skonstruować element tego typu, potrzebujemy dwóch składników: wartości i dowodu, że ta wartość spełnia wymagany przez podtypowanie predykat, w tym przypadku `even`.

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

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

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

Kod źródłowy 7: Przykład elementu typu naturalnej liczby parzystej w Coqu

Tak zdefiniowane podtypowanie rodzi pytanie o unikatowość reprezentacji. Cała koncepcja używania podtypowania do reprezentacji typów ilorazowych opiera się na założeniu, że istnieje tylko jeden element w postaci normalnej dla każdej klasy abstrakcji. Istnienie wielu takich elementów różniących się jedynie dowodem uniemożliwiłoby zastosowanie podtypowania do tego celu.

```
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 8: Twierdzenie mówiące o unikalności reprezentacji w podtypowaniu

Niestety, w Coqu nie da się udowodnić twierdzenia 9 bez dodatkowych aksjomatów. Wersja tego twierdzenia dla typu `sigT` jest natomiast po prostu fałszywa. Musimy więc zredukować nasze oczekiwania lub dodać dodatkowe założenia.

2.3.1. Dodatkowe aksjomaty

Pomimo tego, że w tej pracy unikamy używania dodatkowych założeń spoza Coq, warto rozważyć, jakie rezultaty mogłoby przynieść ich zastosowanie.

Aksjomat irrelewancji

Jest to aksjomat mówiący o tym, że wszystkie dowody tego samego twierdzenia są tym samym dowodem.

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

Kod źródłowy 9: Definicja irrelewancji w Coqu

Jak możemy przypuszczać, posiadając tak potężne narzędzie, jakim jest aksjomat irrelewancji, z łatwością możemy udowodnić twierdzenie 9.

```
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 10: Dowód unikalności reprezentacji używając irrelewancji w Coq

Dodatkowo, możemy udowodnić, że unikatowość reprezentacji jest równoważna aksjomatowi irrelewantności dowodów.

```
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 11: Dowód, że unikalności reprezentacji implikuje irrelewancję w Coq

Aksjomat K

Aksjomat ten został wprowadzony przez Habila Streichera w swojej pracy "Investigations Into Intensional Type Theory"[11]. W tej pracy będziemy korzystać z nieco zmodyfikowanej wersji tego aksjomatu, zwanego UIP (uniqueness of identity proofs), która lepiej odzwierciedla jego konsekwencje.

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

Kod źródłowy 12: Aksjomat K w Coq

Jest to nieco słabsza wersja aksjomatu irrelewantności, która mówi jedynie o irrelewantności dowodów równości. Ma ona pewną ciekawą konsekwencję, opisaną

w [11], mianowicie pozwala na zanurzenie równości na parach zależnych w zwykłą równość.

```
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 13: Twierdzenie o zanurzenie równości na parach zależnych w Coq

Dowód tego twierdzenia zostanie pominięty, jednakże znajduje się on w dodatku *Extras/Stricher.v*. Warto zauważyć, że aksjomat K nie jest słabszy od aksjomatu irrelewanccji, co oznacza, że nie można za jego pomocą w ogólności udowodnić unikalności reprezentacji. W przypadku typów ilorazowych generowanych przez funkcję normalizującą, będziemy jednak potrzebować jedynie unikalności predykatów równości.

```
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 14: Definicja podtypu postaci kanoniczych generowanych przez funkcję normalizującą f, oraz projekcji dla niego w Coq

Unikatowość reprezentacji dla tego typu można z łatwością udowodnić wykorzystując aksjomat K.

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

Kod źródłowy 15: Dowód unikalności reprezentacji dla podtypu postaci kanoniczych generowanych przez funkcję normalizującą f w Coq

Możemy tutaj pójść o krok dalej i zdefiniować, że wszystkie elementy należące do tej samej klasy abstrakcji mają unikalnego reprezentanta, pod warunkiem założenia aksjomatu K. Rozpoczniemy od dowodu, że funkcja f rzeczywiście generuje relację równoważności 1 o nazwie `norm_equiv`.

```
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 16: Definicja relacji równoważności generowanej przez funkcję normalizującą w Coq

Możemy teraz przejść do właściwego dowodu, mając już zdefiniowaną relację równoważności.

```
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 17: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta używając aksjomatu K

Związek między tymi aksjomatami

Jak już wcześniej wspomnieliśmy, aksjomat K jest szczególnym przypadkiem aksjomatu irrelewancji. Oznacza to, że nie są one równoważne. Jednak 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 18: Predykat ekstensjonalności dowodów

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

Kod źródłowy 19: 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 20: Dowód, że ekstensjonalność dowodów oraz aksjomat K implikuje irrelewancję

2.3.2. Wykorzystując uniwersum SProp

SProp jest uniwersum predykatów z definicyjną irrelewancją. Oznacza to, że wszystkie dowody tego samego twierdzenia 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, który jest indeksowany wyrażeniem w Prop. Pozwala na przeniesienie dowolnego predykatu do świata SProp, co z uwagi na niewielką ilość konstrukcji w bibliotece standardowej jest bardzo użyteczne.

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

`sUnit` - jest to odpowiednik `True : Prop`. Również pozwala się wydostać ze 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 21: 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 22: Typ postaci normalnych w SProp

Mając już podstawowe definicje możemy przejść do właściwego dowodu.

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

Kod źródłowy 23: Dowód że wszystkie elementy w tej samej klasie abstrakcji mają wspólnego reprezentanta w `Squotient`

Korzystanie z SProp niesie jednak ze sobą poważny problem polegający na próbie przeniesienia dowodu predykatu do Prop. Gdybyśmy zamienili `Seq` na zwykłą równość (=), nie byłoby możliwe udowodnienie tego twierdzenia. W Coqu, dowody są domyślnie w uniwersum Prop, a z tego uniwersum chcielibyśmy wyprowadzać dowody dla naszych typów ilorazowych. Z uniwersum SProp możemy wydostać się jedynie eliminując `sEmpty` lub `sUnit`, co nie gwarantuje możliwości wyprowadzenia analogicznego dowodu w Prop. Oznacza to, że większość twierdzeń dotycząca zdefiniowanych w ten sposób podtypów, będzie również musiała żyć w SProp, co jest dojść wysoką ceną.

2.3.3. Homotopiczne podejście

Co jednak, jeśli nie chcemy używać dodatkowych aksjomatów i pracować jedynie w uniwersum `Prop`? W takiej sytuacji z pomocą przychodzi homotopiczna teoria typów. Jest to stosunkowo nowa gałąź matematyki, która zajmuje się dowodzeniem równości w różnych typach[8]. Homotopiczną interpretacją równości jest ω-graf, w którym punkty reprezentują elementy typów, a ścieżki reprezentują dowody równości, ścieżki między ścieżkami dowody równości dowodów równości i tak dalej.

N-typy

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

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

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

Kod źródłowy 24: 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 25: Klasa typów żyjących w uniwersum HProp.

HSet - tutaj 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 o tym, że dla tych typów prawdziwy jest aksjomat K. Przykładem takiego typu jest `bool`. Jednakże, dlaczego ma on unikatowe dowody równości, wyjaśnimy później.

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

Kod źródłowy 26: Klasa typów żyjących w uniwersum HSet.

Nie są to jedynie uniwersa. Istnieją kolejne poziomy, w których żyją typy, dla których dowody równości między dowodami równości są zawsze tym samym dowodem itd. Definicję dowolnego uniwersum możemy znaleźć w 27.

```
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 27: Klasa typów żyjących w n-tym uniwersum.

Jak widzimy, typy dowodów równości między elementami typu żyjącego w $(n+1)$-szym uniwersum żyją 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 **??** każde kolejne

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

uniwersum zawiera w sobie poprzednie. Dowód twierdzenia `contr_bottom` pominiemy tutaj, ale można go znaleźć w dodatku *Lib/HoTT.v*. Wracając jednak do naszego podtypowania, widzimy, że dopóki zajmujemy się typami, które żyją w uniwersum HSet, nie musimy się martwić o dowody równości między elementami tego typu. Dlatego doskonale nadają się one do bycia pierwotnymi typami dla naszych typów ilorazowych. Pozostaje tylko ustalić, które typy należą do tego uniwersum. Tu z pomocą przychodzi homotopiczna teoria typów oraz twierdzenie Hedberga [**?**].

Typy z rozstrzygalną równością

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

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

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

Kod źródłowy 29: Definicja rozstrzygalności, oraz rozstrzygalnej równości.

Dobrym przykładem rozstrzygalnego typu jest `bool`. Natomiast przykładem typu, dla którego nie istnieje rozstrzygalna równość, jest typ funkcji `nat -> nat`. Warto dodać, że wspomniane wcześniej twierdzenie Hedberg'ga [7] mówi o tym, że każdy typ z rozstrzygalną równością należy do uniwersum *HSet*. Dowód tego twierdzenia zaczyna się od zdefiniowania klasy typów sprowadzalnych.

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

Kod źródłowy 30: Definicja sprowadzalności.

Pokażemy, że każdy typ rozstrzygalny jest sprowadzalny 31.

Szczególnym przypadkiem są więc typy z rozstrzygalną równością, które mają sprowadzalne dowody równości (ścieżki) 32.

Mając już zdefiniowane sprowadzalne ścieżki, potrzebujemy jeszcze szybkiego dowodu33, że ścieżka złożona z swoja odwrotnością jest ścieżką trywialną.

```
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 31: 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 32: 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 33: Dowód, że każda pętla z ścieżek jest `eq_refl`.

Mając to już za sobą możemy przejść do właściwego dowodu, że dowolny typ z sprawdzalnymi ścieżkami jest *HSet*'em 34.

```
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 34: Dowód, że każdy typ z sprowadzalnymi ścieżkami jest HSet'em.

Jak więc widzimy, każdy typ z rozstrzygalną równością posiada tylko jeden dowód równości między dowolną parą równych sobie elementów. Oznacza to, że bez żadnych dodatkowych aksjomatów możemy udowodnić unikalność reprezentacji dla naszych typów ilorazowych, które posiadają rozstrzygalny typ pierwotny. Z uwagi na to, że zdefiniowanie nie trywialnej funkcji normalizującej na typie bez rozstrzygalnej równości jest prawie niemożliwe, typy z rozstrzygalną równością wystarczą nam w tej pracy.

Równość między parami zależnymi

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

```
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 35: Charakterystyka równości dla par.

Jeśli jednak spróbujemy scharakteryzować równość w podobny sposób dla zależnych par, otrzymamy błąd wynikający z niezgodności typów dowodów. Nawet jeśli pozbędziemy się go, ustalając wspólną pierwszą pozycję w parze, nie uda nam się udowodnić, że równość pary implikuje równość drugiego elementu tych par. Taka zależność implikowałaby bowiem aksjomat K [11]. Aby zrozumieć równość zależnych par, musimy najpierw zdefiniować transport.

```
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 36: Definicja transportu.

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.

```
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 37: Charakterystyka równości dla par zależnych.

Jak więc wynika z twierdzenia 37 równość par zależnych składa się z równości na pierwszych elementów, oraz na równości drugich elementów przetransportowanej wzdłuż pierwszej równości.

Rozdział 3.

Typów ilorazowe z pod-typowaniem

W poprzednim rozdziale omówiliśmy, czym jest podtypowanie oraz jak można go wykorzystać w implementacji typów, które możemy używać niemal jak typów ilorazowych. W niniejszym rozdziale przyjrzymy się typom ilorazowym, które można zaimplementować w ten sposób. Skupimy się tutaj na typach z uniwersum zbiorów (czyli takim bez nietrywialnych dowodów równości) i będziemy pracować z domyślną równością oraz w uniwersum `Prop`.

3.1. Pary nieuporządkowane

Wcześniej wspomnieliśmy, że niemożliwe jest zdefiniowanie pary nieuporządkowanej dla dowolnego typu bazowego, ponieważ nie wiadomo, który element powinien być pierwszy. Dlatego ograniczamy się jedynie do typów bazowych z rozstrzygalnym pełnym porządkiem na nich 38. Słaba asymetria jest potrzebna do udowodnienia unikalności, a pełność zapewnia możliwość zbudowania takiej pary dla każdych dwóch elementów.

```
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 38: Definicja rozstrzygalnego pełnego porządku w Coqu.

Takie pary definiujemy jako trójki elementów: pierwszy element pary, drugi element pary, oraz dowód poprawnego uporządkowania 39.

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

Kod źródłowy 39: Definicja pary nieuporządkowanej w Coqu.

3.1.1. Relacja równoważności

Będziemy chcieli utożsamić ze sobą takie pary, które zawierają takie same dwa elementy, a więc są ze sobą w relacji `sim` 40.

```
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 40: Relacja równoważności par nieuporządkowanych.

3.1.2. Dowód jednoznaczności reprezentacji

Mając te definicje, możemy łatwo udowodnić, że jeśli dwie pary nieuporządkowane zawierają takie same elementy, to są sobie równe w Coqu. Oczywiście, będziemy musieli skorzystać tu z dowodu `bool_is_hset`. Jednak wiedząc, że `bool` w trywialny sposób ma rozstrzygalną równość oraz korzystając z wiedzy z poprzedniego rozdziału, można w prosty sposób udowodnić, że wszystkie dowody równości w `bool` są trywialne. Wymaganie istnienia rozstrzygalnego porządku na typie bazowym ogra-

```
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 41: Dowód, że pary uporządkowane zawierające te same elementy są tą samą parą.

nicza zastosowania pary uporządkowanej. Jednakże, jak już wspomnieliśmy, w prawe wszystkich praktycznych zastosowaniach programistycznych struktury danych są reprezentowane za pomocą ciągów bitów, co umożliwia zdefiniowanie rozstrzygalnego, pełnego porządku. Dlatego też, pary uporządkowane są często używane w większości programistycznych zadań.

3.2. Muti-zbiory skończone

Kolejnym klasycznym przykładem typu ilorazowego, który chcielibyśmy mieć w Coqu, jest multi-zbiór skończony. Jest to struktura danych, która może przechowywać wiele tych samych elementów, a kolejność nie ma w niej znaczenia. Można o nich myśleć jako o nieuporządkowanych listach. Podobnie jak one, multi-zbiory są również monadami i mogą być wykorzystane np. do sprawdzenia, czy różne procesy produkują takie same elementy. Jednakże, podobnie jak w przypadku pary nieuporządkowanej, również tutaj będziemy musieli dodać pewne ograniczenia na typ bazowy w postaci rozstrzygalnego porządku liniowego (42).

```
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 42: Definicja rozstrzygalnego porządku liniowego dla typu A w Coqu.

3.2.1. Relacja równoważności

Chcemy, aby dwa równoważne sobie multi-zbiory miały takie same elementy. Innymi słowy, aby dwa multi-zbiory były sobie równoważne, muszą być swoimi własnymi permutacjami. W tym celu proponujemy zastosowanie nieco sprytniejszej definicji permutacji, która jest równoważna dla typów z rozstrzygalną równością (43). Definicja ta nie wymaga definiowania wszystkich praw permutacji, które mogą być trudne w użyciu, ale opiera się na idei, że dwie listy, które są permutacjami, muszą zawierać takie same elementy z dokładnością do ich ilości. 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). Jedynym problemem z tą definicją jest fakt, że nie działa ona dla typów z nierozstrzygalną równością. Jednakże, rozstrzygalny porządek liniowy implikuje rozstrzygalną równość na typie. Dowód równoważności tych definicji można znaleźć w pliku *Extras/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 43: Definicja permutacji dla list z rozstrzygalną równością.

3.2.2. Funkcja normalizująca

Nie trudno zauważyć, że jako idempotentną funkcję do wyznaczania postaci normalnej dla list możemy użyć funkcji sortującej. W celu osiągnięcia tego celu, proponuję użycie sortowania przez scalanie wykorzystującego drzewo, które przechowuje elementy tylko w liściach 44. Dzięki temu podejściu łatwo jest podzielić listę na dwie równe części, a sama procedura sortowania przez scalanie może być łatwo zaimplementowana w funkcyjnym języku programowania. Oczywiście, można również zastosować dowolną inną funkcję sortującą.

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 sortowanie nie utożsamia ze sobą list, które nie były ze sobą w relacji permutacji. Pierwsze kryterium gwarantuje nam natomiast bezsilność sortowania, gdyż jak wiemy, sortowanie nie zmienia już posortowanej listy. Dowód tego faktu oraz tego, że zaprezentowana powyżej funkcja `mergeSort` (patrz 44) rzeczywiście spełnia przedstawione wymagania, można znaleźć w dodatku $Lib/MergeSort.v$.

3.3. Zbiory skończone

Kolejnym użytecznym typem ilorazowym są zbiory skończone. Różnią się one od multi-zbiorów zdefiniowanych powyżej tym, że każdy element występuje w zbiorze co najwyżej raz. Dodanie elementu do zbioru, który już się w nim znajduje, daje identyczny zbiór jak ten przed tą operacją. Podobnie jak w przypadku multi-zbiorów, w tym typie również wymagamy, aby typ bazowy miał rozstrzygalny porządek liniowy (patrz 42).

```
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 44: Sortowanie przez scalanie z wykorzystaniem drzewa przechowującym wartości w liściach.

3.3.1. Relacja równoważności

W przypadku zbiorów, chcemy utożsamiać ze sobą listy, które zawierają te same elementy, niezależnie od ich liczności. Możemy zatem nieco zmodyfikować definicję permutacji z multi-zbiorów, tak aby rozróżniała jedynie, czy wynik jest zerem, czy nie (patrz 45). Wystarczy zatem zamienić funkcję zliczającą na funkcję, która sprawdza, czy na liście istnieje element spełniający predykat.

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

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

Kod źródłowy 45: 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, zalecamy tutaj użycie funkcji opartej na drzewie, tym razem na klasycznym drzewie binarnym. Funkcja `DSort` (patrz 46) działa poprzez wkładanie kolejnych elementów do drzewa binarnego, pomijając element, jeśli taki już się w nim znajduje, a na końcu spłaszcza drzewo i tworzy z niego listę.

3.3.3. Dowód jednoznaczności reprezentacji

Podobnie jak w przypadku sortowania, każda funkcja sortująca i usuwająca duplikaty powinna spełniać dwa kryteria. Po pierwsze, wynik takiej funkcji powinien być listą, która nie zawiera powtarzających się elementów. Po drugie, elementy te powinny być 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 sortująca i usuwająca duplikaty jest idempotentna. Po trzecie, lista wejściowa i wyjściowa powinna być w relacji zawierania tych samych elementów, co sprawia, że listy zawierające różne elementy nie zostaną ze sobą utożsamione. Dowód faktu, że nasza

```
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 46: Definicja funkcji sortująca i usuwająca duplikaty w Coq.

funkcja sortująca i usuwająca duplikaty `DSort` spełnia kryteria funkcji sortującej i usuwającej duplikaty, możemy znaleźć w dodatku *Lib*/*DedupSort*.*v*.

Rozdział 4.

Typy ilorazowe jako ślad normalizacji

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

4.1. Wolne monoidy

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

4.1.1. Czym jest wolny monoid?

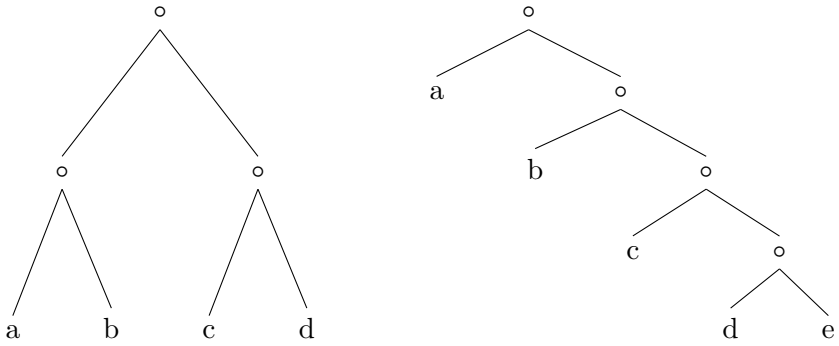
Monoid jest strukturą algebraiczną $(\mathbf{A}, \circ)$, gdzie $\mathbf{A}$ jest nośnikiem struktury, a $\circ$ jest działaniem w tej strukturze. Nośnikiem mogą być na przykład liczby naturalne lub jakiekolwiek inne obiekty, na których możemy wykonywać działania. Działanie w strukturze jest funkcją binarną działającą na nośniku: $\circ : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$. Każdy monoid musi spełniać następujące aksjomaty:

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

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

Dobrym przykładem struktury, która jest monoidem, są macierze z operacją mnożenia lub, w mniej oczywisty sposób, funkcje jednoargumentowe na tym samym typie wraz z operacją ich składania. Wolność struktury algebraicznej polega na tym, że

ograniczamy się jedynie do praw wynikających z jej własności, a nie uwzględniamy praw wynikających z jej nośnika. Dlatego dla wolnego monoidu z liczbami naturalnymi jako nośnikiem nie utożsamiamy ze sobą wyrażeń 2 oraz $1+1$, ponieważ nie wynikają one z praw monoidu. Natomiast wyrażenie $(1+2)+3$ jest tym samym co $1+(2+3)$, ponieważ łączność działania jest jedną z własności monoidu. Dlatego dla dowolnego nośnika możemy myśleć o wolnym monoidzie jako o 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

Możemy zobaczyć na rysunku 4.1, że to samo wyrażenie można zapisać na wiele równoważnych sposobów w wolnym monoidzie. Wynika to z faktu, że nawiasy nie mają znaczenia ze względu na łączność działania, a ponadto można dodać dowolną ilość elementów neutralnych, które nie zmieniają wartości wyrażenia. W sposób naiwny można by zatem zaimplementować wolny monoid korzystając z trzech konstruktorów, jak pokazano w kodzie 47: elementu neutralnego, dowolnej wartości oraz łączącego je operatora.

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

Kod źródłowy 47: Naiwna definicja wolnego monoidu w Coqu.

Taka naiwna definicja, jak przedstawiono na rysunku 4.1, prowadzi do problemów niejednoznaczności reprezentacji. Aby rozwiązać ten problem, musimy ustalić, którą 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 powinno dostrzec w powyższym wyrażeniu znaną każdemu indukcyjną definicję listy (patrz 48), gdzie elementem neutralnym jest `nil`, a kolejne elementy są połączone konstruktorem `cons`.

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

Kod źródłowy 48: Definicja listy z biblioteki standardowej Coqa.

Nasza funkcja normalizująca powinna zatem przechodzić po wszystkich wierzchołkach drzewa `FreeMonoid` (patrz 47) 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 (patrz 49).

```
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 49: Definicja funkcji normalizującej wolny monoid do postaci list w Coqu.

4.2. Liczby całkowite

Klasycznym przykładem wykorzystania typów ilorazowych są liczby całkowite. Są one naturalnym rozszerzeniem liczb naturalnych do grupy addytywnej, w której każdy element ma swój element przeciwny. Możemy je zaimplementować w prosty sposób za pomocą pary liczb naturalnych (patrz 50). Liczba po lewej stronie będzie reprezentować liczbę poprzedników, a liczba po prawej stronie - liczbę następników, z których składa się definiowana liczba.

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

Kod źródłowy 50: Naiwna reprezentacja liczb całkowitych w Coqu.

Taka reprezentacja jest bardzo wygodna w implementacji operacji takich jak suma (patrz 51), następnik czy poprzednik. Wynika to z faktu, że możemy w tym

celu wykorzystać już istniejące operacje na liczbach naturalnych.

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

Kod źródłowy 51: Dodawanie naiwnie zdefiniowanych liczb całkowitych w Coqu.

Niestety, taka definicja pozostawia problem niejednoznaczności reprezentacji. Żeby się na niego natknąć, wystarczy porównać wyniki $1+(-1)=0$ oraz $2+(-2)=0$. Pomimo, że oba te wyrażenia mają ten sam wynik, to w naiwnej postaci `Int` (patrz 50), będą one reprezentowane odpowiednio jako $(1,1)$ 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 postać normalną uznamy taką, w której przynajmniej jednym z elementów pary jest równy zero. Funkcję normalizującą będziemy nazywać taką, która będzie odejmować jedynkę z lewej i prawej strony, tak długo, aż nie będzie to już możliwe 52.

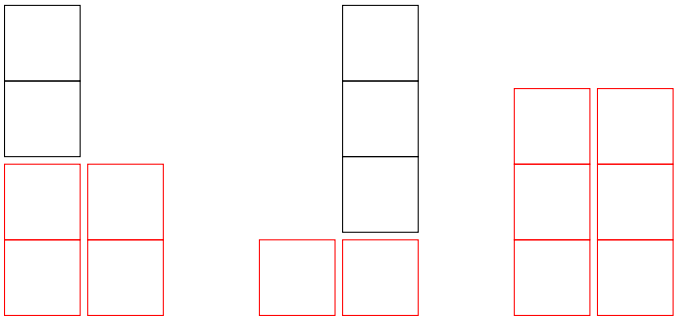
```
Function int_norm' (x y : nat) : (nat * nat) :=
match x, y with
| S x', S y' => int_norm' x' y'
| _, _       => (x, y)
end.

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

Kod źródłowy 52: 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 wyznaczała wartość bezwzględną całej liczby całkowitej.

4.2.2. Indukcyjny typ liczb całkowitych z jednoznaczną reprezentacją

Mając już do dyspozycji funkcję normalizującą, możemy wykorzystać ideę jej działania do stworzenia typu indukcyjnego, który będzie charakteryzował się jednoznacznością reprezentacji. Sam ślad funkcji nie jest szczególnie interesujący w tym



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.

przypadku. Jednak możemy dużo nauczyć się z ostatniego kroku procesu normalizacji. W kodzie 52 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 53: Definicja typu liczb całkowitych z jednoznaczną reprezentacją w Coqu.

Łatwo można zmodyfikować funkcję normalizującą 52 tak, aby przekształcała naszą naiwną reprezentację na nową, jednoznaczną.

Tak przedstawiona definicja nie pozostawia wątpliwości co do swojej jednoznaczności.

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` 53. Podobnie jak w przypadku liczb

```
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 54: Definicja funkcji przekształcającej naiwną reprezentację w jednoznaczną w Coqu.

naturalnych, warto zacząć od zdefiniowania następnika, a z racji istnienia liczb ujemnych również poprzednika.

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

Kod źródłowy 55: Definicja następnika dla liczb całkowitych `Z` 53.

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

Kod źródłowy 56: Definicja poprzednika dla liczb całkowitych `Z` 53.

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

Kod źródłowy 57: Definicja negacji dla liczb całkowitych `Z` 53.

W obu przypadkach definicja jest dość prosta i nie pozostawia wątpliwości co do swojej poprawności, jednakże konieczny był specjalny czwarty przypadek, który odpowiada za przejście do zera. Definicja negacji jest wyjątkowo trywialna w przypadku tej reprezentacji, więc nie wymaga omówienia. Przejdziemy zatem do 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 58: Definicja dodawania dla liczb całkowitych `Z` 53.

W przedstawionej w niniejszej pracy definicji dodawania 58 została wykorzystana funkcja pomocnicza `map_n` 58, która pozwala na nałożenie n operacji na daną wartość. Z uwagi na to, że liczby całkowite wykorzystują liczby naturalne, oczywistym jest zdefiniowanie dodawania jako wykonanie n operacji następnika, lub poprzednika jeśli liczba, którą dodajemy, jest ujemna. Odejmowanie można w łatwy sposób zdefiniować jako dodanie liczby przeciwnej. Możemy zatem zdefiniować ostatnią, lecz równie ważną operację - mnożenie.

```
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 59: Definicja mnożenia dla liczb całkowitych `Z` 53.

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 dobrze znana rekurencyjna definicja mnożenia, więc nie będziemy poświęcać zbyt dużo czasu na jej omówienie. Wszystkie przedstawione powyżej operacje i wiele innych, wraz z dowodami ich podstawowych praw, takich jak łączność,

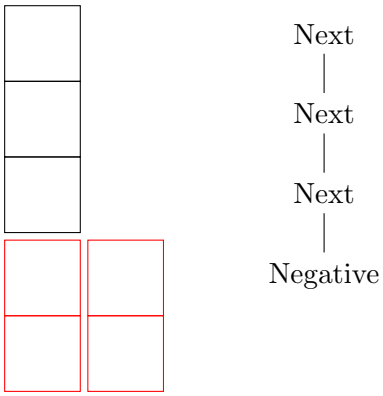
przemienność oraz rozdzielność mnożenia względem dodawania, można znaleźć w dodatku *Integer.v*.

4.3. Egzotyczne liczby całkowite

Poprzednia definicja liczb całkowitych wydaje się być naturalnym kandydatem ze względu na swoje powiązanie z funkcją normalizującą i łatwość obliczeń. Większość konkurencyjnych definicji jest bardzo podobna, czasami zamiast symetrycznych trzech konstruktorów wykorzystują tylko dwa, z których jeden jest obciążony zerem. Niemniej jednak, może budzić niesmak fakt, że w ich definicji wykorzystuje się liczby naturalne. Pojawia się zatem pytanie, czy możliwe jest zdefiniowanie liczb całkowitych bez odwoływania się do liczb naturalnych?

4.3.1. Inne spojrzenie na normalizację

W poprzedniej sekcji skupiliśmy się na procesie normalizacji "od dołu do góry", który polegał na usuwaniu następników z obu elementów pary, aż jeden z nich będzie zerem. Jednakże, możemy odwrócić ten proces, tworząc swego rodzaju pseudo-funkcję normalizacji, która patrzy "od góry do dołu". W tym procesie zliczamy, ile elementów jest powyżej pewnego punktu, nie wiedząc jednak, czy zliczamy następniki, czy też poprzedniki. Dopiero po osiągnięciu punktu, w którym obie wartości są równe, jesteśmy 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 induktywny, bazujący na śladzie zaprezentowanej pseudo-normalizacji. Jednak, aby był on jednoznaczny, musimy pozbyć się problemu związanego z zerem. W zaprezentowanej powyżej intuicji nie wiadomo, w jaki sposób

powinno ono być reprezentowane, ponieważ sytuacja, w której zaczynamy od równych wartości, nie pozwala na wybór konstruktora, który należy użyć. Ten problem rozwiązujemy poprzez zaburzenie symetrii między konstruktorami. Jeden z nich będzie reprezentował zero, a w jego kontekście `Next` będzie następnikiem, co oznacza, że `Next Zero` będzie reprezentować jedynkę. Drugi konstruktor będzie reprezentował minus jedynkę, a `Next` zaaplikowany na nim oznaczał będzie poprzednik.

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

Kod źródłowy 60: Definicja liczby całkowitych z następniko-poprzednikiem.

4.3.3. Operacje na liczbach z następniko-poprzednikiem

Konstrukcja liczb z następnikiem i poprzednikiem pozwala na jednoznaczną reprezentację, jednak nie jest zbyt wygodna do przeprowadzania obliczeń. Podstawowe operacje, takie jak dodawanie następnika czy odejmowanie poprzednika, mają liniową złożoność obliczeniową względem wielkości liczby. Wynika to z faktu, że aby określić, czy powinniśmy dodać następniko-poprzednik, czy go usunąć, musimy znać wartość reprezentowanej liczby. W tym celu musimy sprawdzić ostatni konstruktor, co wprowadza dodatkowe koszty obliczeniowe.

```
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 61: Definicja następnika liczb całkowitych z następniko-poprzednikiem 60.

Oczywiście, można zdefiniować bardziej zaawansowane operacje, takie jak dodawanie czy mnożenie, jednak ich definicje są dość skomplikowane. W niniejszej pracy nie będziemy się nimi zajmować. Definicję dodawania, wraz z dowodem na izomorfizm pomiędzy tą definicją liczb całkowitych, a tą nieco bardziej klasyczną, zaprezentowaną w poprzedniej sekcji, można znaleźć w dodatku *ExoticInteger.v*.

```
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 62: Definicja poprzednika liczb całkowitych z następniko-poprzednikiem 60.

4.4. Dodatnie liczby wymierne

W przeciwieństwie do liczb całkowitych, znalezienie jednoznacznej reprezentacji dla liczb wymiernych nie jest zadaniem prostym. Ta sekcja opiera się na pracy Yves'a Bertota [4], w której przedstawiono koncepcję reprezentacji 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 $\frac{50}{100}$ wyrażają tę samą wartość - pół. Za postać normalną uznaje się zwykle taką formę, w której nie można dokonać dalszych operacji skracania ułamka, czyli podzielenia jego licznika i mianownika przez tę samą liczbę. Aby uzyskać postać nieskracalną, należy podzielić obie strony ułamka przez największy wspólny dzielnik licznika i mianownika. Jednym ze sposobów na jego wyznaczenie największego wspólnego dzielnika jest właśnie zastosowanie algorytmu Euklidesa, jak przedstawiono w 63.

```
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 63: Definicja klasycznego algorytmu Euklidesa w pseudo Coqu.

Algorytm Euklidesa bazuje na prawie $\text{NWD}(a, b) = \text{NWD}(a - b, b)$, jeśli $a > b$. Inną ciekawą obserwacją, z której skorzystamy, jest to, że ślad algorytmu pozostanie niezmieniony, jeśli obie liczby przemnożymy przez dowolną dodatnią liczbę naturalną

k. Przykład takiego przemnożenia przedstawiono poniżej.

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

Jak widzimy w powyższym przykładzie, ślad - czyli to, która liczba okazała się większa - nie zależy od wybranej wartości k. Jedynie wyliczony największy wspólny dzielnik zostaje przemnożony przez stałą k. Znając ślad, możemy odtworzyć argumenty, z dokładnością do największego wspólnego dzielnika. Dlatego też możemy użyć śladów do jednoznacznej reprezentacji liczb wymiernych. Dowód tego faktu oraz innych związanych z śladami algorytmów Euklidesa znajduje się w dodatku *Qplus.v*.

4.4.2. Typ liczb wymiernych dodatnich

Zdefiniujmy zatem typ śladów algorytmu Euklidesa 64. Jak można było się spo-

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

Kod źródłowy 64: Definicja typu śladów algorytmu Euklidesa w Coqu.

dziewać, reprezentacja liczby wymiernej składa się z trzech konstruktorów: `N` - kiedy licznik jest większy od mianownika, `D` - gdy mianownik jest większy, oraz `One` - gdy obie wartości są równe, co kończy proces normalizacji. Taka konstrukcja dodatkowo zapewnia przyjazną indukcję po wszystkich dodatnich liczbach wymiernych. W przeciwieństwie do naiwnej reprezentacji 65, która wykorzystuje parę liczb naturalnych.

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

Kod źródłowy 65: Definicja naiwnej reprezentacji liczb wymiernych w Coqu.

4.4.3. Funkcje przejścia między reprezentacjami

Zauważając, że funkcja `Qplus` 64 reprezentuje ślady algorytmu Euklidesa, powinniśmy być w stanie względnie łatwo napisać kod funkcji generującej ślady algorytm Euklidesa. Niestety, rzeczywistość w Coqu jest dość brutalna, a mechanizm sprawdzania terminacji nie potrafi wyznaczyć dobrze ufundowanego porządku na argumentach algorytmu Euklidesa. Znamy taki porządek, którym jest suma argumentów, ale przekracza to możliwości automatycznego sprawdzania terminacji w Coqu. W związku z tym, posłużymy się konceptem zwany "paliwem". Polega on na dodaniu dodatkowego parametru do funkcji, który będzie się zmniejszał z każdym kolejnym wywołaniem rekurencyjnym funkcji, gwarantując tym samym terminację funkcji w oczach Coqa. Ustalając wartość tego parametru na odpowiednio wysokim poziomie, będziemy w stanie obliczyć ślad, zanim paliwo się skończy.

```
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 66: Definicja funkcji przekształcającą naiwną reprezentację w ilorazowy typ dodanych liczb wymiernych w Coqu.

Jak widzimy w definicji funkcji 67, paliwo zostało ustalone jako suma argumentów podzielona przez ich największy wspólny dzielnik. Wiemy, że w każdej iteracji algorytmu Euklidesa, suma argumentów zostaje zmniejszona przynajmniej o ich największy wspólny dzielnik, co oznacza, że będziemy w stanie wyliczyć cały ślad tego algorytmu zanim paliwo się skończy.

Funkcja odwrotna (z dokładnością do największego wspólnego dzielnika) do tej, którą przedstawiliśmy powyżej, ma równie oczywistą konstrukcję. Będziemy odwracać proces poprzez dodawanie wartości drugiego argumentu do tego, który w trakcie normalizacji był większy. Wynika to z dość 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 67: 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 na wszystkie liczby wymierne. Skorzystamy tutaj z konstrukcji podobnej do tej, którą przedstawiliśmy dla liczb dodatnich, czyli 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 68: 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, a teraz przyszedł czas na ich bardziej zaawansowaną wersję, czyli grupy. Grupa ma trzy podstawowe prawa:

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

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

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

Widzimy zatem, że każda grupa jest również monoidem, a jedyną różnicą jest wprowadzenie nowej operacji - odwrotności. Możemy zatem przypuszczać, że wolną grupą

będą listy elementów oraz ich odwrotności. Naiwną implementację możemy zapisać jako listę par, w których pierwszy element to element grupy, a drugi element to wartość boolowska określająca, czy na tym elemencie została już zaaplikowana operacja odwrotności.

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

Kod źródłowy 69: Naiwna implementacja wolnej grupy w Coqu.

4.5.1. Normalizacja wolnej grupy

Od znormalizowanej wolnej grupy będziemy wymagać, aby dwa wzajemnie odwrotne elementy nie były sąsiadami, ponieważ, jak wiemy z praw grupy, takie elementy się ”anihilują”i są równoważne elementowi neutralnemu.

```
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 70: Predykat bycia w postaci normalnej dla wolnej grupy w Coqu.

```
Class EqDec (A : Type) := {
  eqf : A -> A -> bool ;
  eqf_leibniz : forall x y: A, reflect (x = y) (eqf x y)
}.
```

Kod źródłowy 71: Klasa typów z rozstrzygalną równością w Coqu.

Predykat bycia w postaci normalnej możemy wyrazić za pomocą typu induktywnego z trzema konstruktorami dla pustej grupy, singletona, oraz dla następnika 70. Funkcję normalizującą73 również bez większych problemów da się wyrazić, jednak, żeby to zrobić nasz typ bazowy będzie musiał posiadać zdefiniowaną rozstrzygalną równość na naszym typie bazowym (patrz 71).

4.5.2. Jednoznaczny typ dla wolnych grup w Coqu

Wiedząc już, czym charakteryzują się wolne grupy znormalizowane, możemy przejść do stworzenia dla nich typu ilorazowego, bazującego na funkcji normalizacji.

```
Class Group (A : Type) := GroupDef {
  zero     : A;
  op       : A -> A -> A;
  inv      : A -> A;
  l_op_id  : forall x: A, op zero x = x;
  r_op_id  : forall x: A, op x zero = x;
  l_op_inv : forall x: A, op (inv x) x = zero;
  r_op_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);
}.

Class EqGroup (A : Type) `{E: EqDec A} `{G: Group A} := {}.

Definition sub {A: Type} `{Group A} (x y: A) := op x (inv y).
```

Kod źródłowy 72: Klasa grup oraz grupy z rozstrzygalną równością oraz definicja różnicy w Coqu.

```
Fixpoint normalize {A: Type} `{EqGroup 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 73: Funkcja normalizująca wolną grupą w Coqu.

Pierwszą ważną obserwacją jest to, że normalizacja zaczyna się od końca. Po drugie, wyrażenie `eqf v v'` jest równoważne przyrównaniu do elementu neutralnego, czyli różnicy między `v` oraz `v'`, czyli `eqf (op v (inv v')) zero`. Innymi słowy, będą nas interesować różnice między elementami. Po trzecie, podobnie jak w przypadku elementów, dla wartości boolowskich również będą nas interesować różnice między kolejnymi znakami. Nasz typ ilorazowy będzie zatem listą różnic do poprzedniego elementu, zawierającą zarówno różnicę wartości, jak i znaków. Dodatkowo wymuszamy, aby te same elementy nie mogły stać obok siebie z różnymi znakami.

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

Definition QuotFreeGroup (A: Type) `{EqGroup A} :=
option (NEQuotFreeGroup A).
```

Kod źródłowy 74: Ilorazowy typ wolnej grupy w Coqu.

W tekście widzimy, że definicja wolnej grupy 74 składa się z dwóch typów, niepustych grup oraz potencjalnie pustych. Wynika to z problemu niejednoznaczności elementu neutralnego. Aby uniknąć tej niejednoznaczności, zdefiniowano osobny typ dla niepustych grup, który składa się z trzech konstruktorów.

Pierwszy konstruktor, `Singleton`, jest prosty i po prostu definiuje ostatni element wolnej grupy. Drugi konstruktor, `Switch`, reprezentuje zmianę znaku w stosunku do poprzedniego elementu. Wymaga on, aby różnica między elementami była różna od zera, ponieważ w przeciwnym razie obok siebie stałyby te same elementy z przeciwnymi znakami. Aby uprościć dowody tego faktu, konstruktor ten jest opakowany w `Squash` - niezależny od dowodów. Ostatni z konstruktorów, `Stay`, reprezentuje różnicę między kolejnymi elementami z tym samym znakiem. W tym przypadku nie potrzeba dodatkowych dowodów. Potrzeba wyliczania różnicy między elementami wymusza, aby typ bazowy był sam w sobie grupą z operacją odwrotności.

W językach bez typów zależnych, można by zastosować dwa różne typy bazowe, różniące się jedynie istnieniem elementu neutralnego, co umożliwiłoby łatwiejsze wymuszanie warunków na konstruktorze `Switch`. Jednak, w przypadku korzystania z typów zależnych, zdecydowano się na jednoznaczniejszą definicję, która zapewnia bardziej czytelną konstrukcję.

4.5.3. Funkcje przejścia między reprezentacjami

Mając już zdefiniowany typ ilorazowy dla list wolnych, powinniśmy zdefiniować funkcję, która pozwoli nam przekonwertować naszą nawianą reprezentację na typ ilorazowy. Jednakże jej definicja nie jest trywialna ze względu na zastosowany typ zależny w drugim konstruktorze.

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

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

Definition all_to_quot {A: Type} `{EqGroup A} (x: CanonFreeGroup A) :
  QuotFreeGroup A := to_quot (normalize x).
```

Kod źródłowy 75: Definicja funkcji przekształcającej naiwną reprezentacje w ilorazową w Coqu.

Jak widzimy w definicji funkcji `to_quot` 75, w przypadku, gdy 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, wykorzystując rozstrzygalną równość. Funkcja `to_quot` działa poprawnie tylko i wyłącznie dla znormalizowanych wolnych grup, zatem w przypadku, gdy obok siebie występują wzajemnie odwrotne elementy, zwraca ona cokolwiek. W celu stworzenia funkcji przejścia do postaci ilorazowej dowolnej wolnej grupy w postaci kanonicznej, musimy najpierw ją znormalizować, a więc złożyć funkcję przejścia z funkcją normalizującą.

Funkcja odwrotna `to_canon` 76 ma prostszą konstrukcję, ponieważ nie wymaga

```
Fixpoint to_canon' {A: Type} `{EqGroup A} (x: NEQuotFreeGroup A) :
  bool * A * list (bool*A) :=
match x with
| Singl b v      => ((b, v), [])
| Stay v x'      => match to_canon' x' with
                    | ((b, v'), y)  => ((b, op v v'), (b, v') :: y)
                    end
| Switch v _ x' => match to_canon' x' with
                    | ((b, v'), y)  => ((negb b, op v v'), (b, v') :: y)
                    end
end.

Definition to_canon {A: Type} `{EqGroup A} (x: QuotFreeGroup A) : CanonFreeGroup A :=
match x with
| None => []
| Some x' => let (h, t) := to_canon' x' in h :: t
end.
```

Kod źródłowy 76: Definicja funkcji przekształcającej ilorazową reprezentacje w naiwną w Coqu.

konstruowania typów zależnych. Polega ona na dodawaniu do siebie kolejnych różnic w celu wyliczenia kolejnych wartości. Dla konstruktora `Stay` przepisywany jest znak poprzedniego elementu, a dla konstruktora `Switch` przeciwny znak. Ponieważ funkcja `to_canon'` zajmuje się niepustymi wolnymi grupami, zwraca ona niepustą listę elementów zaimplementowaną w postaci pary zawierającej głowę oraz teraz już potencjalnie pustą listę.

4.5.4. Wolna grupa jako grupa

Jak wiemy, wolne grupy same stanowią grupę, gdzie operatorem jest konkatenacja wolnych grup, elementem neutralnym jest wolna grupa składająca się jedynie z elementu neutralnego, a odwrotnością jest odwrócona wolna grupa z przeciwnymi znakami, zgodnie z prawem odwrotności w grupach:

$$(xy)^{-1} = y^{-1}x^{-1}$$

Możemy zatem zdefiniować odpowiednie operacje i wykazać, że spełniają one prawa grup. Operacje te możemy oczywiście zdefiniować na typie ilorazowym 74, jednak z uwagi na to, że typ ten jest reprezentowany jako lista różnic, operacje takie jak łączenie czy odwracanie kolejności są niezwykle niewygodne w implementacji oraz nieoptymalne obliczeniowo. Dlatego też zdefiniujemy te operacje na naiwnej implementacji wolnej grupy z wykorzystaniem listy, tak aby wynik dla znormalizowanych

argumentów był znormalizowany.

```
Definition canon_cons {A: Type} `{EqGroup A} (b: bool) (v: A)
  (x: CanonFreeGroup A) : CanonFreeGroup A :=
  match x with
  | [] => [(b, v)]
  | (b', v') :: t => if negb (eqf v v') || eqb b b'
                     then (b, v) :: (b', v') :: t
                     else t
  end.

Fixpoint canon_concat {A: Type} `{EqGroup A}
  (x y: CanonFreeGroup A) : CanonFreeGroup A :=
  match x with
  | []           => y
  | (b, v) :: x' => canon_cons b v (canon_concat x' y)
  end.
```

Kod źródłowy 77: Definicja konkatenacji wolnych grup w Coqu.

Konkatenacja wolnych grup 77 działa podobnie jak konkatenacja list, polegając na przenoszeniu elementów z pierwszej listy do drugiej. Jednakże, w przypadku wolnych grup, dodanie elementu na początek grupy może prowadzić do "anihilacji"jej głowy, jeśli był to ten sam element, ale z przeciwnym znakiem.

```
Fixpoint canon_apinv {A: Type} `{EqGroup A} (x y: CanonFreeGroup A) :=
  match x with
  | [] => y
  | (b, v) :: x' => canon_apinv x' ((negb b, v) :: y)
  end.

Definition canon_inv {A: Type} `{EqGroup A} (x: CanonFreeGroup A)
  := canon_apinv x [].
```

Kod źródłowy 78: Definicja funkcji odwrotności dla wolnych grup w Coqu.

Podobnie jak w przypadku odwracania zwykłych list, aby zdefiniować funkcję odwracającą wolne grupy 78 w czasie liniowym, posłużymy się funkcją pomocniczą `canon_apinv`. Funkcja ta przekłada elementy z jednej listy na drugą, ze zmienionym znakiem. Ponieważ zakładamy, że wolna grupa jest w postaci normalnej, po odwróceniu i zmianie znaku każdego elementu nie powstają żadne pary, które uprościłyby się do elementu neutralnego.

W celu zdefiniowania operacji na wolnych grupach w postaci ilorazowej, można

posłużyć się złożeniem zdefiniowanych powyżej funkcji z funkcjami przejścia. W pliku *FreeGroup.v* zdefiniowana została również funkcja konkatenacji bezpośrednio na typie ilorazowym, jednak jej złożoność obliczeniowa wynosi iloczyn długości list. Dodatkowo, zdefiniowany został izomorfizm między znormalizowanymi grupami w naiwnej postaci a tymi w postaci ilorazowej dla odpowiednich funkcji konkatenacji. Na koniec udowodniono, że wolna grupa wraz z zdefiniowanymi powyżej operacjami spełnia prawa grupy.

4.5.5. Wolna grupa jako monada

Każda wolna grupa jest również monadą, a co za tym idzie, funktorem. Oczywiście, aby zdefiniować ją dla typu ilorazowego, musimy ograniczyć się jedynie do typów, które spełniają interfejs grupy **??**.

```
Definition canon_pure {A: Type} (x: A) := [(true, x)].
```

Kod źródłowy 79: Definicja funkcji czystej dla wolnych grup w Coqu.

```
Fixpoint canon_bind {A B: Type} `{EqGroup A} `{EqGroup B}
  (x : CanonFreeGroup A) (f: A -> CanonFreeGroup B) : CanonFreeGroup B :=
  match x with
  | [] => []
  | (b, v) :: x' =>
      if b
      then canon_concat (f v) (canon_bind x' f)
      else canon_concat (canon_inv (f v)) (canon_bind x' f)
  end.
```

Kod źródłowy 80: Definicja funkcji wiążącej dla wolnych grup w Coqu.

Definicja funkcji czystej to po prostu singleton zawierający przekazany element z "dodatnim"znakiem. Funkcję wiążącą można opisać następująco:

$$\mathrm{bind}_f\ (a_1 \circ a_2^{-1} \circ \cdots \circ a_n) = f(a_1) \circ f(a_2)^{-1} \circ \cdots \circ f(a_n)$$

Dodatkowo, w pliku *Extras/FreeGroupMonad.v* znajdują się dowody praw monady dla typu wolnych grup z operacjami zdefiniowanymi powyżej.

4.5.6. Zastosowania wolnych grup

TODO

4.6. Listy różnicowe jak multi-zbiory

W poprzednim rozdziale pokazaliśmy, jak można zdefiniować zbiory i multi-zbiory w Coqu używając pod-typowania. Jest to skuteczna strategia, ale niestety nie można jej przenieść do języków, które nie posiadają typów zależnych, takich jak Haskell. W związku z tym, nasuwa się 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 przypuszczać, w celu znormalizowania listy elementów w multi-zbiorze posłużymy się funkcją sortującą, podobnie jak w poprzednim rozdziale.

Jednakże w tym przypadku przyjrzymy się zmodyfikowanemu algorytmowi sortowania przez wybieranie dla liczb naturalnych 81. Modyfikacja ta polega na zoptymalizowaniu kolejnych operacji porównań poprzez odjęcie od każdej liczby znalezionego minimum. Porównywanie liczb naturalnych w systemie unarnym jest liniowe względem mniejszej z liczb. Z tego wynika, że im mniejsze liczby znajdują się na liście, tym proces szukania minimum jest szybszy. Możemy teraz przyjrzeć się śladowi tej funkcji, a konkretnie temu, jak będą wyglądać kolejne minima na sortowanej liście. Ponieważ w każdym wywołaniu rekurencyjnym wartości są pomniejszane o minimum, kolejne wyliczane minima będą stanowić różnicę między kolejnymi (zgodnie z porządkiem na liczbach naturalnych) elementami.

4.6.2. Typ ilorazowy dla multi-zbiorów

Zanim przystąpimy do implementacji typu ilorazowego, spróbujemy uogólnić przedstawiony powyżej schemat na więcej typów. Po pierwsze, potrzebujemy, aby na danym typie istniał porządek liniowy, ponieważ w przypadku, gdy istnieją dwie różne minimalne wartości, nie byłoby możliwe wyznaczenie unikalnego minimum. Po drugie, musi istnieć jakaś operacja różnicy dwóch elementów oraz operacja dodawania różnicy do wartości, aby móc odzyskać oryginalny multi-zbiór.

W niniejszej pracy została ostatecznie wykorzystana definicja 82, która swoją konstrukcją implikuje istnienie porządku liniowego, nie wymagając jego istnienia jako przesłanki. Wymaga ona za to istnienia typu różnic symetrycznych, funkcji, która dla dwóch elementów wylicza takową różnicę, oraz funkcji dodającej wskazaną różnicę do elementu. Dodatkowo, będziemy wymagać spełnienia pięciu praw. Po pierwsze, wymusza symetryczność różnicy poprzez prawo przemienności operacji różnicy. 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

```
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 81: Definicja sortowania przez wybieranie dla liczb naturalnych w Coqu.

```
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 82: Definicja klasy typów dla której istnieją unikatowe listy różnicowe w Coqu.

jakąś różnicę i wynik się nie zmienił, to znaczy, że ta różnica jest neutralna; prawo to wraz z trzecim implikuje antysymetryczność 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, w pliku *DifferentialLists.v* znajduje się dowód tego faktu. Mając zdefiniowaną klasę, możemy napisać typ list różnicowych 83. Typ ten jest zdefiniowany za pomocą opcji, która pozwala na istnienie

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

Kod źródłowy 83: Definicja list różnicowych w Coqu.

pustych multi-zbiorów. Wewnątrz opcji znajduje się głowa listy typu bazowego oraz lista różnic.

4.6.3. Funkcje przejścia między reprezentacjami

Po zdefiniowaniu typu multi-zbiorów za pomocą list różnicowych, możemy napisać funkcje umożliwiające przejście między reprezentacją listową a ilorazową.

Funkcja `from_diff` 84 pozwala na wygenerowanie listy elementów z listy różnicowej. Zasada jej działania jest bardzo prosta. Jeśli lista różnicowa jest pusta, funkcja zwraca pustą listę. Jeśli lista różnicowa składa się tylko z głowy, zwracany jest singleton. Dla dłuższych list, kolejne wartości są sumą elementu poprzedzającego i kolejnej różnicy.

```
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 84: Definicja funkcji przejścia z listy różnicowej do listowej postaci multi-zbioru w Coqu.

```
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 85: Definicja funkcji przejścia z listowej postaci multi-zbioru do listy różnicowej w Coqu.

Funkcja `to_diff` 84 to złożenie dwóch funkcji: sortującej i funkcji generującej listy różnicowe. Zasada działania `to_diff_sorted` jest bardzo prosta - polega na wyliczaniu kolejnych różnic pomiędzy elementami listy. Naturalnie, pierwszy element listy zostaje zapisany jako głowa listy różnicowej. Obie te funkcje są dość 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 86, dodającej element x.

```
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 86: Definicja funkcji dodającej element do listy różnicowej w Coqu.

Funkcja 86 przechodzi 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 elementem x, oraz pomiędzy elementem 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 to, że długość listy zwiększyła się o 1. Dowody tych faktów można również znaleźć w dodatku $Differential Lists.v$.

Rozdział 5.

Funkcje jako typy ilorazowe

W tym rozdziale przedstawiony zostanie sposób wykorzystania typu funkcji do zdefiniowania pewnych typów ilorazowych. Niestety, poniższe przykłady generują więcej problemów niż rozwiązań i w związku z tym są praktycznie bezużyteczne w rzeczywistych aplikacjach. Niemniej jednak, stanowią ciekawy przykład niekonwencjonalnego podejścia do problemu i dlatego zostały zamieszczone w tym krótkim rozdziale.

5.1. Jak funkcje utożsamiają elementy

Na początku pracy wspomnieliśmy, że nie można zdefiniować typu zbiorów oraz multi-zbiorów dla dowolnego typu bazowego. Jest to prawdą w przypadku typów induktywnych, gdzie kolejność konstruktorów ma znaczenie. Jednakże, możemy wykorzystać wbudowany w praktycznie każdy język programowania typ funkcji. Oczywiście, aby móc w rozsądny sposób rozumować o równościach na funkcjach, będziemy musieli założyć aksjomat ekstensjonalności funkcji87. Wprowadzenie tego aksjomatu nie wprowadza sprzeczności do systemu Coq, więc możemy go bezpiecznie dodać z biblioteki standardowej. Po dodaniu tego aksjomatu, funkcje "zapominają"swój

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

Kod źródłowy 87: Aksjomat ekstensjonalności funkcji w Coqu.

wbudowany algorytm sprawdzania równości. Dwie funkcje, które dają takie same wyniki dla całej przestrzeni argumentów, będą sobie równe. Możemy wykorzystać tę właściwość do "zapomnienia"kolejności elementów, nie powołując się na porządek liniowy.

5.2. Definicja zbioru i multi-zbioru

Mając już ten koncept w głowie, możemy przejść do zdefiniowania typu zbioru 88 jako funkcji rozstrzygającej, czy element należy do zbioru. W przeciwieństwie do

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

Kod źródłowy 88: Typ zbioru w Coqu.

innych definicji zbiorów przedstawionych w tej pracy, zbiory przedstawione powyżej mogą być potencjalnie nieskończone dla typów, które mają nieskończenie wiele elementów. Pozwala nam to na bardziej elastyczne definicje, jednak kosztem obliczalności podstawowych operacji. Jedną 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. Problemu stopu jak wiemy jest jednak problem nierozstrzygalny. W oczywisty sposób, inne operacje, takie jak sprawdzenie, czy dwa zbiory są sobie równe, również będą nierozstrzygalne z tego samego powodu.

Multi-zbiór możemy zdefiniować w bardzo podobny sposób [**?**] jak zbiór, zamieniając jedynie typ dwuelementowy `bool` na typ liczb naturalnych `nat`. Taka imple-

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

Kod źródłowy 89: Typ multi-zbioru w Coqu.

mentacja wspiera jedynie takie multi-zbiory, które mają skończoną liczbę każdego z elementów. Jednakże, można zamienić liczby naturalne na liczby co-naturalne, aby pozbyć się tego ograniczenia. Podobnie jak w przypadku zbiorów, również tu występuje problem z rozstrzyganiem niepustości.

5.3. Rekurencyjne operacje na zbiorach

Wykazaliśmy, że podstawowe operacje, takie jak sprawdzenie niepustości czy równość, nie są rekurencyjne na potencjalnie nieskończonych zbiorach i multi-zbiorach. Podobnie jest w przypadku operacji mapowania zbioru, gdyż gdyby 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 jednak zdefiniować parę użytecznych funkcji. Sprawdzenie, czy dany element należy do zbioru, jest trywialne, gdyż sam zbiór jest definiowany przez taką funkcją. Możemy także zdefiniować funkcję filtrującą dla zbiorów [**?**]. Funkcja ta

```
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 90: Funckja filtrująca dla zbiorów w Coqu.

działa w sposób leniwy, zatem potencjalna nieskończoność zbioru jej nie przeszkadza. W podobny sposób można zdefiniować sumę, przekrój oraz dopełnienie [**?**]. Wykorzy-

```
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 91: Definicja sumy, przekroju oraz dopełniania dla zbiorów w Coqu.

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

```
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 92: Definicja dodawania elementu do zbioru oraz konwersji listy do zbioru w Coqu.

porównanie badanego elementu z tym, który dodajemy, a następnie zwrócenie wartości `true`, jeśli są one równe. Wykorzystując tę funkcję, można zdefiniować tworzenie zbioru z listy elementów. Podobnie dla multi-zbiorów, lecz zamiast zwracania wartości `true`, nakładany jest następnik. Dowody poprawności zdefiniowanych powyżej

funkcji można znaleźć w dodatku *FunctionalQuotient.v*.

Rozdział 6.

Typy ilorazowe w wybranych językach

W poprzednich rozdziałach omówiliśmy, w jaki sposób możemy zdefiniować typy ilorazowe w Coqu, a dokładniej, jak możemy obejść problem braku typów ilorazowych w tym języku. Niektóre z tych rozwiązań nie wymagają użycia typów zależnych i mogą zostać zaimplementowane w prawie każdym języku programowania. Inne rozwiązania wymagają takiego wsparcia. W tym rozdziale przyjrzymy się językom które posiadają wbudowane wsparcie dla typów ilorazowych. Poznamy w jaki sposób można z nich korzystać oraz czy podobnego typu mechanizmy da się wykorzystać w Coqu.

6.1. Lean

Lean, a szczególnie jego najnowsza wersja - Lean 4, jest asystentem dowodzenia rozwijanym od 2013 roku. Jest to narzędzie open source pozwalające na dowodzenie poprawności programów oraz twierdzeń matematycznych, podobnie jak Coq. Projekt Lean jest rozwijany przez Microsoft Research.

6.1.1. Różnice w stosunku do Coqa

W przeciwieństwie do Coqa, Lean nie wymaga konstruktywności dla twierdzeń [3]. Oznacza to, że mamy do czynienia z systemem bardziej zbliżonym do matematycznych dowodów niż do programowania. Niestety, oznacza to również, że nie każdy dowód jest w istocie programem zgodnie z izomorfizmem Currego-Howarda. Ponadto, w Leanie 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.

Niektórzy być może wiedzą, że jedną z głównych różnic między Leanem a Coqiem jest stosowanie logiki klasycznej w Leanie. W związku z tym można by przypuszczać, że prawo wyłączonego środka powinno być jednym z aksjomatów w tym systemie. Okazuje się jednak, że wymienione wyżej trzy aksjomaty wystarczają do wyprowadzenia prawa wyłączonego środka, jak również egzystencjalności funkcji. Egzystencjalność można wyprowadzić z istnienia ilorazów, a dowód wyłączonego środka korzysta z konstrukcji zaproponowanej przez Diaconescu w 1975 roku [5]. Dowody te można również znaleźć w bibliotece standardowej Lean pod nazwami `em` oraz `funext`.

6.1.2. Typy ilorazowe

Jak już widzieliśmy wcześniej, typy ilorazowe stanowią kluczowy element języka Lean. Po zapoznaniu się z podobieństwami i różnicami między Leanem a Coqiem, warto bliżej przyjrzeć się aksjomatom, które towarzyszą typom ilorazowym w Leanie.

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

Kod źródłowy 93: Odpowiednik aksjomatu `Quot` w Coqu.

Pierwszy aksjomat, którym warto się przyjrzeć, to `Quot` 93. Postuluje on 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 94: Odpowiednik aksjomatu `Quot.mk` w Coqu.

Drugim aksjomatem, który warto omówić, jest `Quot.mk` 94. Postuluje on 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 95: Odpowiednik aksjomatu `Quot.ind` w Coqu.

Trzeci aksjomat, który warto wspomnieć, to `Quot.ind` 95. Postuluje on prawa indukcji dla typów ilorazowych. Oznacza to, że jeśli dla każdego elementu powstałego z elementu typu bazowego zachodzi pewien predykat, to zachodzi on również 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 96: Odpowiednik aksjomatu `Quot.lift` w Coqu.

Czwarty aksjomat, `Quot.lift` 96, opisuje, jak aplikować funkcje z typu bazowego na elementach typu ilorazowego. Wymaga to, aby funkcja szanowała relacje - jeśli dwa elementy są ze sobą w relacji, to wynik funkcji dla nich obu musi być taki sam. W Leanie aksjomat `Quot.lift` jest łączony z regułą przepisywania, która mówi, że aplikacja funkcji f respektującej relację na elemencie typu ilorazowego daje identyczny efekt jak aplikacja jej na tym samym elemencie typu pierwotnego. Dlatego, aby korzystać z typów ilorazowych jak w Leanie, konieczne jest postulowanie dodatkowego aksjomatu z tą regułą przepisywania w Coqu.

```
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 97: Odpowiednik aksjomatu `Quot.sound` w Coqu.

Piąty i ostatni aksjomat, który warto omówić, to `Quot.sound` 97. Postuluje on równość elementów typu ilorazowego, jeśli znajdują się one w relacji. Oznacza to, że typy ilorazowe w Leanie są rzeczywiście ilorazowe, czyli sklejone zgodnie z relacją.

Przedstawione powyżej aksjomaty są wystarczające, aby wprowadzić typy ilorazowe do języka Lean. W ten sposób możemy cieszyć się typami ilorazowymi opartymi na tych aksjomatach. Co do Coqa, to również można wprowadzić do niego te same aksjomaty, co pozwoliłoby na korzystanie z typów ilorazowych w tym języku.

6.2. Agda

Agda jest językiem programowania stworzonym z myślą o wsparciu dla typów zależnych [2]. Powstał on jako rozszerzenie teorii typów Martina-Löfa [9]. Ze względu na te cechy, Agda może służyć jako asystent dowodzenia. W przeciwieństwie jednak do języków takich jak Coq czy Lean, Agda nie posiada języka taktyk, co znacząco utrudnia jej wykorzystanie w tym celu. Natomiast obsługa typów zależnych w Agdzie jest na znacznie wyższym poziomie niż to, co możemy doświadczyć w Coqu. Agda potrafi sama w wielu wypadkach wywnioskować, że dany przypadek jest niemożliwy, co sprawia, że definiowanie funkcji zależnych jest dużo 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 98: Definicja funkcji zwracającej n-ty element zależnego wektora w Agdzie.

6.2.1. Kubiczna Agda

Cubical to rozszerzenie języka Agda, które rozbudowuje możliwości języka o kubiczną teorię typów [10]. Pozwala to na modelowanie konstrukcji z homotopicznej teorii typów natywnie w Agdzie i daje dostęp do takich funkcji jak transport 36, które już wcześniej poznaliśmy. Wprowadza ono koncepcję ścieżek jako dowodów równości między elementami. Wszystkie te właściwości można było w większym lub mniejszym stopniu zamodelować w Coqu. Jednak to, co wyróżnia kubiczną Agdę na tle innych asystentów dowodzenia, to wyższe typy induktywne (HIT). Podobnie jak zwykłe typy induktywne, pozwalają one na definiowanie pewnych struktur danych. W przeciwieństwie jednak do znanych już nam typów induktywnych, które automatycznie generują równości między elementami, w wyższych typach induktywnych możemy ręcznie dodać kolejne ścieżki. Oznacza możliwość tworzenia typów posiadających nietrywialne równości między elementami. Tutaj dobrym przykładem będzie okrąg99, który posiada jedną nietrywialną ścieżkę.

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

Kod źródłowy 99: Definicja okręgu w kubicznej Agdzie.

6.2.2. Definicja typów ilorazowych za pomocą wyższych typów induktywnych

Jak możemy zauważyć, definicja multi-zbioru skończonego100 w kubicznej Agdzie jest niezwykle prosta. Wystarczy dodać dodatkową ścieżkę (dowód równości) na listach, które mają dwa początkowe elementy zamienione miejscami. Przechodniość równości jest zapewniona z jej definicji, a więc nie musimy jej explicite definiować. Dodatkowo, równość dla pustego multi-zbioru oraz następnika dostajemy bezpłatnie, z ich definicji. W efekcie, ta definicja 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 100: Definicja multi-zbioru skończonego w kubicznej Agdzie.

W bardzo podobny sposób możemy zdefiniować liczby całkowite w kubicznej Agdzie, za pomocą zera, następnika oraz poprzednika. Aby jednak zagwarantować izomorfizm tej konstrukcji z znanymi nam z matematyki liczbami całkowitymi, musimy dodać dwie dodatkowe równości, które mówią, w jaki sposób następnik i poprzednik nawzajem się niwelują. W ten sposób otrzymujemy prawie liczby całkowite 101. 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. Jednakże, na potrzeby tej pracy, możemy uznać tę definicję za poprawną.

```
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 101: Definicja liczb całkowitych w Agdzie.

Podsumowując, wyższe typy indykatywne doskonale nadają się do definiowania typów ilorazowych. Niestety, nie możemy ich zastosować do definiowania typów ilorazowych w Coqu, ponieważ jego system typów nie wspiera tak zaawansowanych konstrukcji. Próba odtworzenia dodawania równości między elementami za pomocą aksjomatów prowadzi bardzo szybko do wprowadzenia sprzeczności do systemu. Konieczne byłoby wprowadzenie wsparcia dla kubicznej teorii typów do Coqa, na razie jednak nie zanosi się na taki przełom.

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.